Speech and Language Processing

An Introduction to Natural Language Processing,
Computational Linguistics, and Speech Recognition

Third Edition draft

Daniel Jurafsky
Stanford University

James H. Martin
University of Colorado at Boulder

Draft of August 15, 2018. Comments and typos welcome!

Summary of Contents

Contents

CHAPTER

1

Introduction

Placeholder

CHAPTER

2 Regular Expressions, Text Normalization, Edit Distance

```
User:  You are like my father in some ways.
ELIZA: WHAT RESEMBLANCE DO YOU SEE
User:  You are not very aggressive but I think you don't want me to
       notice that.
ELIZA: WHAT MAKES YOU THINK I AM NOT VERY AGGRESSIVE
User:  You don't argue with me.
ELIZA: WHY DO YOU THINK I DON'T ARGUE WITH YOU
User:  You are afraid of me.
ELIZA: DOES IT PLEASE YOU TO BELIEVE I AM AFRAID OF YOU
```

Weizenbaum (1966)

ELIZA

The dialogue above is from **ELIZA**, an early natural language processing system that could carry on a limited conversation with a user by imitating the responses of a Rogerian psychotherapist (Weizenbaum, 1966). ELIZA is a surprisingly simple program that uses pattern matching to recognize phrases like "You are X" and translate them into suitable outputs like "What makes you think I am X?". This simple technique succeeds in this domain because ELIZA doesn't actually need to *know* anything to mimic a Rogerian psychotherapist. As Weizenbaum notes, this is one of the few dialogue genres where listeners can act as if they know nothing of the world. Eliza's mimicry of human conversation was remarkably successful: many people who interacted with ELIZA came to believe that it really *understood* them and their problems, many continued to believe in ELIZA's abilities even after the program's operation was explained to them (Weizenbaum, 1976), and even today such **chatbots** are a fun diversion.

chatbots

Of course modern conversational agents are much more than a diversion; they can answer questions, book flights, or find restaurants, functions for which they rely on a much more sophisticated understanding of the user's intent, as we will see in Chapter 25. Nonetheless, the simple pattern-based methods that powered ELIZA and other chatbots play a crucial role in natural language processing.

We'll begin with the most important tool for describing text patterns: the **regular expression**. Regular expressions can be used to specify strings we might want to extract from a document, from transforming "You are X" in Eliza above, to defining strings like *$199* or *$24.99* for extracting tables of prices from a document.

text normalization

We'll then turn to a set of tasks collectively called **text normalization**, in which regular expressions play an important part. Normalizing text means converting it to a more convenient, standard form. For example, most of what we are going to do with language relies on first separating out or **tokenizing** words from running text, the task of **tokenization**. English words are often separated from each other by whitespace, but whitespace is not always sufficient. *New York* and *rock 'n' roll* are sometimes treated as large words despite the fact that they contain spaces, while sometimes we'll need to separate *I'm* into the two words *I* and *am*. For processing tweets or texts we'll need to tokenize **emoticons** like `:)` or **hashtags** like `#nlproc`. Some languages, like Chinese, don't have spaces between words, so word tokenization becomes more difficult.

tokenization

lemmatization Another part of text normalization is **lemmatization**, the task of determining that two words have the same root, despite their surface differences. For example, the words *sang*, *sung*, and *sings* are forms of the verb *sing*. The word *sing* is the common *lemma* of these words, and a **lemmatizer** maps from all of these to *sing*. Lemmatization is essential for processing morphologically complex languages like Arabic. **stemming** **Stemming** refers to a simpler version of lemmatization in which we mainly just strip suffixes from the end of the word. Text normalization also includes **sentence segmentation** **sentence segmentation**: breaking up a text into individual sentences, using cues like periods or exclamation points.

Finally, we'll need to compare words and other strings. We'll introduce a metric called **edit distance** that measures how similar two strings are based on the number of edits (insertions, deletions, substitutions) it takes to change one string into the other. Edit distance is an algorithm with applications throughout language processing, from spelling correction to speech recognition to coreference resolution.

2.1 Regular Expressions

SIR ANDREW: Her C's, her U's and her T's: why that?
Shakespeare, *Twelfth Night*

regular expression One of the unsung successes in standardization in computer science has been the **regular expression** (**RE**), a language for specifying text search strings. This practical language is used in every computer language, word processor, and text processing tools like the Unix tools grep or Emacs. Formally, a regular expression is an algebraic notation for characterizing a set of strings. They are particularly useful for searching in texts, when we have a **pattern** to search for and a **corpus** of texts to search through. **corpus** A regular expression search function will search through the corpus, returning all texts that match the pattern. The corpus can be a single document or a collection. For example, the Unix command-line tool `grep` takes a regular expression and returns every line of the input document that matches the expression.

A search can be designed to return every match on a line, if there are more than one, or just the first match. In the following examples we generally underline the exact part of the pattern that matches the regular expression and show only the first match. We'll show regular expressions delimited by slashes but note that slashes are *not* part of the regular expressions.

Regular expressions come in many variants. We'll be describing **extended regular expressions**; different regular expression parsers may only recognize subsets of these, or treat some expressions slightly differently. Using an online regular expression tester is a handy way to test out your expressions and explore these variations.

2.1.1 Basic Regular Expression Patterns

The simplest kind of regular expression is a sequence of simple characters. To search for *woodchuck*, we type `/woodchuck/`. The expression `/Buttercup/` matches any string containing the substring *Buttercup*; `grep` with that expression would return the line *I'm called little Buttercup*. The search string can consist of a single character (like `/!/`) or a sequence of characters (like `/urgl/`).

Regular expressions are **case sensitive**; lower case `/s/` is distinct from upper case `/S/` (`/s/` matches a lower case *s* but not an upper case *S*). This means that the pattern `/woodchucks/` will not match the string *Woodchucks*. We can solve this

RE	Example Patterns Matched
`/woodchucks/`	"interesting links to woodchucks and lemurs"
`/a/`	"Mary Ann stopped by Mona's"
`/!/`	"You've left the burglar behind again!" said Nori

Figure 2.1 Some simple regex searches.

problem with the use of the square braces `[` and `]`. The string of characters inside the braces specifies a **disjunction** of characters to match. For example, Fig. 2.2 shows that the pattern `/[wW]/` matches patterns containing either *w* or *W*.

RE	Match	Example Patterns
`/[wW]oodchuck/`	Woodchuck or woodchuck	"Woodchuck"
`/[abc]/`	'a', 'b', *or* 'c'	"In uomini, in soldati"
`/[1234567890]/`	any digit	"plenty of 7 to 5"

Figure 2.2 The use of the brackets `[]` to specify a disjunction of characters.

The regular expression `/[1234567890]/` specified any single digit. While such classes of characters as digits or letters are important building blocks in expressions, they can get awkward (e.g., it's inconvenient to specify

`/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]/`

to mean "any capital letter"). In cases where there is a well-defined sequence associated with a set of characters, the brackets can be used with the dash (`-`) to specify any one character in a **range**. The pattern `/[2-5]/` specifies any one of the characters *2*, *3*, *4*, or *5*. The pattern `/[b-g]/` specifies one of the characters *b*, *c*, *d*, *e*, *f*, or *g*. Some other examples are shown in Fig. 2.3.

range

RE	Match	Example Patterns Matched
`/[A-Z]/`	an upper case letter	"we should call it 'Drenched Blossoms' "
`/[a-z]/`	a lower case letter	"my beans were impatient to be hoed!"
`/[0-9]/`	a single digit	"Chapter 1: Down the Rabbit Hole"

Figure 2.3 The use of the brackets `[]` plus the dash `-` to specify a range.

The square braces can also be used to specify what a single character *cannot* be, by use of the caret `^`. If the caret `^` is the first symbol after the open square brace `[`, the resulting pattern is negated. For example, the pattern `/[^a]/` matches any single character (including special characters) except *a*. This is only true when the caret is the first symbol after the open square brace. If it occurs anywhere else, it usually stands for a caret; Fig. 2.4 shows some examples.

RE	Match (single characters)	Example Patterns Matched
`/[^A-Z]/`	not an upper case letter	"Oyfn pripetchik"
`/[^Ss]/`	neither 'S' nor 's'	"I have no exquisite reason for't"
`/[^\.]/`	not a period	"our resident Djinn"
`/[e^]/`	either 'e' or '^'	"look up ^ now"
`/a^b/`	the pattern 'a^b'	"look up a^ b now"

Figure 2.4 The caret `^` for negation or just to mean `^`. See below re: the backslash for escaping the period.

How can we talk about optional elements, like an optional *s* in *woodchuck* and *woodchucks*? We can't use the square brackets, because while they allow us to say "s or S", they don't allow us to say "s or nothing". For this we use the question mark `/?/`, which means "the preceding character or nothing", as shown in Fig. 2.5.

RE	Match	Example Patterns Matched
`/woodchucks?/`	woodchuck or woodchucks	"woodchuck"
`/colou?r/`	color or colour	"colour"

Figure 2.5 The question mark `?` marks optionality of the previous expression.

We can think of the question mark as meaning "zero or one instances of the previous character". That is, it's a way of specifying how many of something that we want, something that is very important in regular expressions. For example, consider the language of certain sheep, which consists of strings that look like the following:

baa!
baaa!
baaaa!
baaaaa!
...

This language consists of strings with a *b*, followed by at least two *a*'s, followed by an exclamation point. The set of operators that allows us to say things like "some number of *a*s" are based on the asterisk or `*`, commonly called the **Kleene *** (generally pronounced "cleany star"). The Kleene star means "zero or more occurrences of the immediately previous character or regular expression". So `/a*/` means "any string of zero or more *a*s". This will match *a* or *aaaaaa*, but it will also match *Off Minor* since the string *Off Minor* has zero *a*'s. So the regular expression for matching one or more *a* is `/aa*/`, meaning one *a* followed by zero or more *a*s. More complex patterns can also be repeated. So `/[ab]*/` means "zero or more *a*'s or *b*'s" (not "zero or more right square braces"). This will match strings like *aaaa* or *ababab* or *bbbb*.

Kleene *

For specifying multiple digits (useful for finding prices) we can extend `/[0-9]/`, the regular expression for a single digit. An integer (a string of digits) is thus `/[0-9][0-9]*/`. (Why isn't it just `/[0-9]*/`?)

Sometimes it's annoying to have to write the regular expression for digits twice, so there is a shorter way to specify "at least one" of some character. This is the **Kleene +**, which means "one or more occurrences of the immediately preceding character or regular expression". Thus, the expression `/[0-9]+/` is the normal way to specify "a sequence of digits". There are thus two ways to specify the sheep language: `/baaa*!/` or `/baa+!/`.

Kleene +

One very important special character is the period (`/./`), a **wildcard** expression that matches any single character (*except* a carriage return), as shown in Fig. 2.6.

RE	Match	Example Matches
`/beg.n/`	any character between *beg* and *n*	begin, beg'n, begun

Figure 2.6 The use of the period `.` to specify any character.

The wildcard is often used together with the Kleene star to mean "any string of characters". For example, suppose we want to find any line in which a particular word, for example, *aardvark*, appears twice. We can specify this with the regular expression `/aardvark.*aardvark/`.

Anchors

Anchors are special characters that anchor regular expressions to particular places in a string. The most common anchors are the caret ˆ and the dollar sign `$`. The caret ˆ matches the start of a line. The pattern `/^The/` matches the word *The* only at the

start of a line. Thus, the caret `^` has three uses: to match the start of a line, to indicate a negation inside of square brackets, and just to mean a caret. (What are the contexts that allow grep or Python to know which function a given caret is supposed to have?) The dollar sign `$` matches the end of a line. So the pattern `␣$` is a useful pattern for matching a space at the end of a line, and `/^The dog\.$/` matches a line that contains only the phrase *The dog.* (We have to use the backslash here since we want the `.` to mean "period" and not the wildcard.)

There are also two other anchors: `\b` matches a word boundary, and `\B` matches a non-boundary. Thus, `/\bthe\b/` matches the word *the* but not the word *other*. More technically, a "word" for the purposes of a regular expression is defined as any sequence of digits, underscores, or letters; this is based on the definition of "words" in programming languages. For example, `/\b99\b/` will match the string *99* in *There are 99 bottles of beer on the wall* (because 99 follows a space) but not *99* in *There are 299 bottles of beer on the wall* (since 99 follows a number). But it will match *99* in *$99* (since *99* follows a dollar sign ($), which is not a digit, underscore, or letter).

2.1.2 Disjunction, Grouping, and Precedence

Suppose we need to search for texts about pets; perhaps we are particularly interested in cats and dogs. In such a case, we might want to search for either the string *cat* or the string *dog*. Since we can't use the square brackets to search for "cat or dog" (why can't we say `/[catdog]/`?), we need a new operator, the **disjunction** operator, also called the **pipe** symbol `|`. The pattern `/cat|dog/` matches either the string `cat` or the string `dog`.

disjunction

Sometimes we need to use this disjunction operator in the midst of a larger sequence. For example, suppose I want to search for information about pet fish for my cousin David. How can I specify both *guppy* and *guppies*? We cannot simply say `/guppy|ies/`, because that would match only the strings *guppy* and *ies*. This is because sequences like `guppy` take **precedence** over the disjunction operator `|`. To make the disjunction operator apply only to a specific pattern, we need to use the parenthesis operators `(` and `)`. Enclosing a pattern in parentheses makes it act like a single character for the purposes of neighboring operators like the pipe `|` and the Kleene*. So the pattern `/gupp(y|ies)/` would specify that we meant the disjunction only to apply to the suffixes `y` and `ies`.

Precedence

The parenthesis operator `(` is also useful when we are using counters like the Kleene*. Unlike the `|` operator, the Kleene* operator applies by default only to a single character, not to a whole sequence. Suppose we want to match repeated instances of a string. Perhaps we have a line that has column labels of the form *Column 1 Column 2 Column 3*. The expression `/Column␣[0-9]+␣*/` will not match any number of columns; instead, it will match a single column followed by any number of spaces! The star here applies only to the space `␣` that precedes it, not to the whole sequence. With the parentheses, we could write the expression `/(Column␣[0-9]+␣*)*/` to match the word *Column*, followed by a number and optional spaces, the whole pattern repeated any number of times.

This idea that one operator may take precedence over another, requiring us to sometimes use parentheses to specify what we mean, is formalized by the **operator precedence hierarchy** for regular expressions. The following table gives the order of RE operator precedence, from highest precedence to lowest precedence.

operator precedence

Parenthesis	`()`
Counters	`* + ? {}`
Sequences and anchors	`the ^my end$`
Disjunction	`|`

Thus, because counters have a higher precedence than sequences, `/the*/` matches *theeeee* but not *thethe*. Because sequences have a higher precedence than disjunction, `/the|any/` matches *the* or *any* but not *theny*.

Patterns can be ambiguous in another way. Consider the expression `/[a-z]*/` when matching against the text *once upon a time*. Since `/[a-z]*/` matches zero or more letters, this expression could match nothing, or just the first letter *o*, *on*, *onc*, or *once*. In these cases regular expressions always match the *largest* string they can; we say that patterns are **greedy**, expanding to cover as much of a string as they can.

greedy

non-greedy

*?

+?

There are, however, ways to enforce **non-greedy** matching, using another meaning of the ? qualifier. The operator ***?** is a Kleene star that matches as little text as possible. The operator **+?** is a Kleene plus that matches as little text as possible.

2.1.3 A Simple Example

Suppose we wanted to write a RE to find cases of the English article *the*. A simple (but incorrect) pattern might be:

`/the/`

One problem is that this pattern will miss the word when it begins a sentence and hence is capitalized (i.e., *The*). This might lead us to the following pattern:

`/[tT]he/`

But we will still incorrectly return texts with `the` embedded in other words (e.g., *other* or *theology*). So we need to specify that we want instances with a word boundary on both sides:

`/\b[tT]he\b/`

Suppose we wanted to do this without the use of `/\b/`. We might want this since `/\b/` won't treat underscores and numbers as word boundaries; but we might want to find *the* in some context where it might also have underlines or numbers nearby (*the_* or *the25*). We need to specify that we want instances in which there are no alphabetic letters on either side of the *the*:

`/[^a-zA-Z][tT]he[^a-zA-Z]/`

But there is still one more problem with this pattern: it won't find the word *the* when it begins a line. This is because the regular expression `[^a-zA-Z]`, which we used to avoid embedded instances of *the*, implies that there must be some single (although non-alphabetic) character before the *the*. We can avoid this by specifying that before the *the* we require *either* the beginning-of-line or a non-alphabetic character, and the same at the end of the line:

`/(^|[^a-zA-Z])[tT]he([^a-zA-Z]|$)/`

false positives

false negatives

The process we just went through was based on fixing two kinds of errors: **false positives**, strings that we incorrectly matched like *other* or *there*, and **false negatives**, strings that we incorrectly missed, like *The*. Addressing these two kinds of

errors comes up again and again in implementing speech and language processing systems. Reducing the overall error rate for an application thus involves two antagonistic efforts:

- Increasing **precision** (minimizing false positives)
- Increasing **recall** (minimizing false negatives)

2.1.4 A More Complex Example

Let's try out a more significant example of the power of REs. Suppose we want to build an application to help a user buy a computer on the Web. The user might want "any machine with at least 6 GHz and 500 GB of disk space for less than $1000". To do this kind of retrieval, we first need to be able to look for expressions like *6 GHz* or *500 GB* or *Mac* or *$999.99*. In the rest of this section we'll work out some simple regular expressions for this task.

First, let's complete our regular expression for prices. Here's a regular expression for a dollar sign followed by a string of digits:

```
/$[0-9]+/
```

Note that the `$` character has a different function here than the end-of-line function we discussed earlier. Most regular expression parsers are smart enough to realize that `$` here doesn't mean end-of-line. (As a thought experiment, think about how regex parsers might figure out the function of `$` from the context.)

Now we just need to deal with fractions of dollars. We'll add a decimal point and two digits afterwards:

```
/$[0-9]+\.[0-9][0-9]/
```

This pattern only allows *$199.99* but not *$199*. We need to make the cents optional and to make sure we're at a word boundary:

```
/(^|\W)$[0-9]+(\.[0-9][0-9])?\b/
```

One last catch! This pattern allows prices like *$199999.99* which would be far too expensive! We need to limit the dollar

```
/(^|\W)$[0-9]{0,3}(\.[0-9][0-9])?\b/
```

How about specifications for > 6GHz processor speed? Here's a pattern for that:

```
/\b[6-9]+␣*(GHz|[Gg]igahertz)\b/
```

Note that we use `/␣*/` to mean "zero or more spaces" since there might always be extra spaces lying around. For disk space, we'll need to allow for optional fractions again (*5.5 GB*); note the use of `?` for making the final `s` optional:

```
/\b[0-9]+(\.[0-9]+)?␣*(GB|[Gg]igabytes?)\b/
```

Modifying this regular expression so that it only matches more than 500 GB is left as an exercise for the reader.

2.1.5 More Operators

Figure 2.7 shows some aliases for common ranges, which can be used mainly to save typing. Besides the Kleene * and Kleene + we can also use explicit numbers as

counters, by enclosing them in curly brackets. The regular expression /{3}/ means "exactly 3 occurrences of the previous character or expression". So /a\.{24}z/ will match *a* followed by 24 dots followed by *z* (but not *a* followed by 23 or 25 dots followed by a *z*).

RE	Expansion	Match	First Matches
\d	[0-9]	any digit	Party␣of␣5
\D	[^0-9]	any non-digit	Blue␣moon
\w	[a-zA-Z0-9_]	any alphanumeric/underscore	Daiyu
\W	[^\w]	a non-alphanumeric	!!!!
\s	[␣\r\t\n\f]	whitespace (space, tab)	
\S	[^\s]	Non-whitespace	in␣Concord

Figure 2.7 Aliases for common sets of characters.

A range of numbers can also be specified. So /{n,m}/ specifies from *n* to *m* occurrences of the previous char or expression, and /{n,}/ means at least *n* occurrences of the previous expression. REs for counting are summarized in Fig. 2.8.

RE	Match
*	zero or more occurrences of the previous char or expression
+	one or more occurrences of the previous char or expression
?	exactly zero or one occurrence of the previous char or expression
{n}	*n* occurrences of the previous char or expression
{n,m}	from *n* to *m* occurrences of the previous char or expression
{n,}	at least *n* occurrences of the previous char or expression
{,m}	up to *m* occurrences of the previous char or expression

Figure 2.8 Regular expression operators for counting.

Newline

Finally, certain special characters are referred to by special notation based on the backslash (\) (see Fig. 2.9). The most common of these are the **newline** character \n and the **tab** character \t. To refer to characters that are special themselves (like ., *, [, and \), precede them with a backslash, (i.e., /\./, /*/, /\[/, and /\\/).

RE	Match	First Patterns Matched
*	an asterisk "*"	"K*A*P*L*A*N"
\.	a period "."	"Dr. Livingston, I presume"
\?	a question mark	"Why don't they come and lend a hand?"
\n	a newline	
\t	a tab	

Figure 2.9 Some characters that need to be backslashed.

2.1.6 Regular Expression Substitution, Capture Groups, and ELIZA

substitution

An important use of regular expressions is in **substitutions**. For example, the substitution operator `s/regexp1/pattern/` used in Python and in Unix commands like `vim` or `sed` allows a string characterized by a regular expression to be replaced by another string:

```
s/colour/color/
```

It is often useful to be able to refer to a particular subpart of the string matching the first pattern. For example, suppose we wanted to put angle brackets around all

integers in a text, for example, changing *the 35 boxes* to *the <35> boxes*. We'd like a way to refer to the integer we've found so that we can easily add the brackets. To do this, we put parentheses (and) around the first pattern and use the **number** operator \1 in the second pattern to refer back. Here's how it looks:

```
s/([0-9]+)/<\1>/
```

The parenthesis and number operators can also specify that a certain string or expression must occur twice in the text. For example, suppose we are looking for the pattern "the Xer they were, the Xer they will be", where we want to constrain the two X's to be the same string. We do this by surrounding the first X with the parenthesis operator, and replacing the second X with the number operator \1, as follows:

```
/the (.*)er they were, the \1er they will be/
```

Here the \1 will be replaced by whatever string matched the first item in parentheses. So this will match *the bigger they were, the bigger they will be* but not *the bigger they were, the faster they will be*.

capture group

This use of parentheses to store a pattern in memory is called a **capture group**. Every time a capture group is used (i.e., parentheses surround a pattern), the resulting match is stored in a numbered **register**. If you match two different sets of parentheses, \2 means whatever matched the *second* capture group. Thus

register

```
/the (.*)er they (.*), the \1er we \2/
```

will match *the faster they ran, the faster we ran* but not *the faster they ran, the faster we ate*. Similarly, the third capture group is stored in \3, the fourth is \4, and so on.

Parentheses thus have a double function in regular expressions; they are used to group terms for specifying the order in which operators should apply, and they are used to capture something in a register. Occasionally we might want to use parentheses for grouping, but don't want to capture the resulting pattern in a register. In that case we use a **non-capturing group**, which is specified by putting the commands ?: after the open paren, in the form (?: pattern).

non-capturing group

```
/(?:some|a few) (people|cats) like some \1/
```

will match *some cats like some cats* but not *some cats like some a few*.

Substitutions and capture groups are very useful in implementing simple chatbots like ELIZA (Weizenbaum, 1966). Recall that ELIZA simulates a Rogerian psychologist by carrying on conversations like the following:

User$_1$:	Men are all alike.
ELIZA$_1$:	IN WHAT WAY
User$_2$:	They're always bugging us about something or other.
ELIZA$_2$:	CAN YOU THINK OF A SPECIFIC EXAMPLE
User$_3$:	Well, my boyfriend made me come here.
ELIZA$_3$:	YOUR BOYFRIEND MADE YOU COME HERE
User$_4$:	He says I'm depressed much of the time.
ELIZA$_4$:	I AM SORRY TO HEAR YOU ARE DEPRESSED

ELIZA works by having a series or cascade of regular expression substitutions each of which matches and changes some part of the input lines. Input lines are first uppercased. The first substitutions then change all instances of *MY* to *YOUR*, and *I'M* to *YOU ARE*, and so on. The next set of substitutions matches and replaces other patterns in the input. Here are some examples:

```
s/.* I'M (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/
s/.* I AM (depressed|sad) .*/WHY DO YOU THINK YOU ARE \1/
s/.* all .*/IN WHAT WAY/
s/.* always .*/CAN YOU THINK OF A SPECIFIC EXAMPLE/
```

Since multiple substitutions can apply to a given input, substitutions are assigned a rank and applied in order. Creating patterns is the topic of Exercise 2.3, and we return to the details of the ELIZA architecture in Chapter 25.

2.1.7 Lookahead assertions

Finally, there will be times when we need to predict the future: look ahead in the text to see if some pattern matches, but not advance the match cursor, so that we can then deal with the pattern if it occurs.

lookahead These **lookahead** assertions make use of the `(?` syntax that we saw in the previous section for non-capture groups. The operator `(?= pattern)` is true if `pattern` occurs, but is **zero-width**, i.e. the match pointer doesn't advance. The operator `(?! pattern)` only returns true if a pattern does not match, but again is zero-width and doesn't advance the cursor. Negative lookahead is commonly used when we are parsing some complex pattern but want to rule out a special case. For example suppose we want to match, at the beginning of a line, any single word that doesn't start with "Volcano". We can use negative lookahead to do this:

zero-width

`/^(?!Volcano)[A-Za-z]+/`

2.2 Words

corpus **corpora** Before we talk about processing words, we need to decide what counts as a word. Let's start by looking at one particular **corpus** (plural **corpora**), a computer-readable collection of text or speech. For example the Brown corpus is a million-word collection of samples from 500 written English texts from different genres (newspaper, fiction, non-fiction, academic, etc.), assembled at Brown University in 1963–64 (Kučera and Francis, 1967). How many words are in the following Brown sentence?

He stepped out into the hall, was delighted to encounter a water brother.

This sentence has 13 words if we don't count punctuation marks as words, 15 if we count punctuation. Whether we treat period ("."), comma (","), and so on as words depends on the task. Punctuation is critical for finding boundaries of things (commas, periods, colons) and for identifying some aspects of meaning (question marks, exclamation marks, quotation marks). For some tasks, like part-of-speech tagging or parsing or speech synthesis, we sometimes treat punctuation marks as if they were separate words.

The Switchboard corpus of American English telephone conversations between strangers was collected in the early 1990s; it contains 2430 conversations averaging 6 minutes each, totaling 240 hours of speech and about 3 million words (Godfrey et al., 1992). Such corpora of spoken language don't have punctuation but do introduce other complications with regard to defining words. Let's look at one utterance from Switchboard; an **utterance** is the spoken correlate of a sentence:

utterance

I do uh main- mainly business data processing

disfluency
fragment
filled pause

This utterance has two kinds of **disfluencies**. The broken-off word *main-* is called a **fragment**. Words like *uh* and *um* are called **fillers** or **filled pauses**. Should we consider these to be words? Again, it depends on the application. If we are building a speech transcription system, we might want to eventually strip out the disfluencies.

But we also sometimes keep disfluencies around. Disfluencies like *uh* or *um* are actually helpful in speech recognition in predicting the upcoming word, because they may signal that the speaker is restarting the clause or idea, and so for speech recognition they are treated as regular words. Because people use different disfluencies they can also be a cue to speaker identification. In fact Clark and Fox Tree (2002) showed that *uh* and *um* have different meanings. What do you think they are?

Are capitalized tokens like *They* and uncapitalized tokens like *they* the same word? These are lumped together in some tasks (speech recognition), while for part-of-speech or named-entity tagging, capitalization is a useful feature and is retained.

lemma

wordform

How about inflected forms like *cats* versus *cat*? These two words have the same **lemma** *cat* but are different wordforms. A **lemma** is a set of lexical forms having the same stem, the same major part-of-speech, and the same word sense. The **wordform** is the full inflected or derived form of the word. For morphologically complex languages like Arabic, we often need to deal with lemmatization. For many tasks in English, however, wordforms are sufficient.

word type

word token

How many words are there in English? To answer this question we need to distinguish two ways of talking about words. **Types** are the number of distinct words in a corpus; if the set of words in the vocabulary is V, the number of types is the vocabulary size $|V|$. **Tokens** are the total number N of running words. If we ignore punctuation, the following Brown sentence has 16 tokens and 14 types:

They picnicked by the pool, then lay back on the grass and looked at the stars.

When we speak about the number of words in the language, we are generally referring to word types.

Corpus	Tokens = N	Types = $|V|$
Shakespeare	884 thousand	31 thousand
Brown corpus	1 million	38 thousand
Switchboard telephone conversations	2.4 million	20 thousand
COCA	440 million	2 million
Google N-grams	1 trillion	13 million

Figure 2.10 Rough numbers of types and tokens for some English language corpora. The largest, the Google N-grams corpus, contains 13 million types, but this count only includes types appearing 40 or more times, so the true number would be much larger.

Herdan's Law
Heaps' Law

Fig. 2.10 shows the rough numbers of types and tokens computed from some popular English corpora. The larger the corpora we look at, the more word types we find, and in fact this relationship between the number of types $|V|$ and number of tokens N is called **Herdan's Law** (Herdan, 1960) or **Heaps' Law** (Heaps, 1978) after its discoverers (in linguistics and information retrieval respectively). It is shown in Eq. 2.1, where k and β are positive constants, and $0 < \beta < 1$.

$$|V| = kN^{\beta} \tag{2.1}$$

The value of β depends on the corpus size and the genre, but at least for the large corpora in Fig. 2.10, β ranges from .67 to .75. Roughly then we can say that

the vocabulary size for a text goes up significantly faster than the square root of its length in words.

Another measure of the number of words in the language is the number of lemmas instead of wordform types. Dictionaries can help in giving lemma counts; dictionary **entries** or **boldface forms** are a very rough upper bound on the number of lemmas (since some lemmas have multiple boldface forms). The 1989 edition of the Oxford English Dictionary had 615,000 entries.

2.3 Corpora

Words don't appear out of nowhere. Any particular piece of text that we study is produced by one or more specific speakers or writers, in a specific dialect of a specific language, at a specific time, in a specific place, for a specific function.

Perhaps the most important dimension of variation is the language. NLP algorithms are most useful when they apply across many languages. The world has 7097 languages at the time of this writing, according to the online Ethnologue catalog (Simons and Fennig, 2018). Most NLP tools tend to be developed for the official languages of large industrialized nations (Chinese, English, Spanish, Arabic, etc.), but we don't want to limit tools to just these few languages. Furthermore, most languages also have multiple varieties, such as dialects spoken in different regions or by different social groups. Thus, for example, if we're processing text in African American Vernacular English (**AAVE**), a dialect spoken by millions of people in the United States, it's important to make use of NLP tools that function with that dialect. Twitter posts written in AAVE make use of constructions like *iont* (*I don't* in Standard American English (**SAE**)), or *talmbout* corresponding to SAE *talking about*, both examples that influence word segmentation (Blodgett et al. 2016, Jones 2015).

AAVE

SAE

It's also quite common for speakers or writers to use multiple languages in a single communicative act, a phenomenon called **code switching**. Code switching is enormously common across the world; here are examples showing Spanish and (transliterated) Hindi code switching with English (Solorio et al. 2014, Jurgens et al. 2017):

code switching

(2.2) Por primera vez veo a @username actually being hateful! it was beautiful:)
[For the first time I get to see @username actually being hateful! it was beautiful:]

(2.3) dost tha or ra- hega ... dont wory ... but dherya rakhe
["he was and will remain a friend ... don't worry ... but have faith"]

Another dimension of variation is the genre. The text that our algorithms must process might come from newswire, fiction or non-fiction books, scientific articles, Wikipedia, or religious texts. It might come from spoken genres like telephone conversations, business meetings, police body-worn cameras, medical interviews, or transcripts of television shows or movies. It might come from work situations like doctors' notes, legal text, or parliamentary or congressional proceedings.

Text also reflects the demographic characteristics of the writer (or speaker): their age, gender, race, socio-economic class can all influence the linguistic properties of the text we are processing.

And finally, time matters too. Language changes over time, and for some languages we have good corpora of texts from different historical periods.

Because language is so situated, when developing computational models for lan-

guage processing, it's important to consider who produced the language, in what context, for what purpose, and make sure that the models are fit to the data.

2.4 Text Normalization

Before almost any natural language processing of a text, the text has to be normalized. At least three tasks are commonly applied as part of any normalization process:

1. Segmenting/tokenizing words from running text
2. Normalizing word formats
3. Segmenting sentences in running text.

In the next sections we walk through each of these tasks.

2.4.1 Unix tools for crude tokenization and normalization

Let's begin with an easy, if somewhat naive version of word tokenization and normalization (and frequency computation) that can be accomplished for English solely in a single UNIX command-line, inspired by Church (1994). We'll make use of some Unix commands: `tr`, used to systematically change particular characters in the input; `sort`, which sorts input lines in alphabetical order; and `uniq`, which collapses and counts adjacent identical lines.

For example let's begin with the 'complete words' of Shakespeare in one textfile, `sh.txt`. We can use `tr` to tokenize the words by changing every sequence of non-alphabetic characters to a newline ('A-Za-z' means alphabetic, the -c option complements to non-alphabet, and the -s option squeezes all sequences into a single character):

```
tr -sc 'A-Za-z' '\n' < sh.txt
```

The output of this command will be:

```
THE
SONNETS
by
William
Shakespeare
From
fairest
creatures
We
...
```

Now that there is one word per line, we can sort the lines, and pass them to `uniq -c` which will collapse and count them:

```
tr -sc 'A-Za-z' '\n' < sh.txt | sort | uniq -c
```

with the following output:

```
1945 A
72 AARON
19 ABBESS
25 Aaron
```

```
6 Abate
1 Abates
5 Abbess
6 Abbey
3 Abbot
...
```

Alternatively, we can collapse all the upper case to lower case:

```
tr -sc 'A-Za-z' '\n' < sh.txt | tr A-Z a-z | sort | uniq -c
```

whose output is

```
14725 a
  97 aaron
   1 abaissiez
  10 abandon
   2 abandoned
   2 abase
   1 abash
  14 abate
   3 abated
   3 abatement
   ...
```

Now we can sort again to find the frequent words. The `-n` option to `sort` means to sort numerically rather than alphabetically, and the `-r` option means to sort in reverse order (highest-to-lowest):

```
tr -sc 'A-Za-z' '\n' < sh.txt | tr A-Z a-z | sort | uniq -c | sort -n -r
```

The results show that the most frequent words in Shakespeare, as in any other corpus, are the short **function words** like articles, pronouns, prepositions:

```
27378 the
26084 and
22538 i
19771 to
17481 of
14725 a
13826 you
12489 my
11318 that
11112 in
   ...
```

Unix tools of this sort can be very handy in building quick word count statistics for any corpus.

2.4.2 Word Tokenization and Normalization

tokenization
normalization

The simple UNIX tools above were fine for getting rough word statistics but more sophisticated algorithms are generally necessary for **tokenization**, the task of segmenting running text into words, and **normalization**, the task of putting words/tokens in a standard format.

While the Unix command sequence just removed all the numbers and punctuation, for most NLP applications we'll need to keep these in our tokenization. We

often want to break off punctuation as a separate token; commas are a useful piece of information for parsers, periods help indicate sentence boundaries. But we'll often want to keep the punctuation that occurs word internally, in examples like *m.p.h,*, *Ph.D.*, *AT&T*, *cap'n*. Special characters and numbers will need to be kept in prices ($45.55) and dates (`01/02/06`); we don't want to segment that price into separate tokens of "45" and "55". And there are URLs (`http://www.stanford.edu`), Twitter hashtags (`#nlproc`), or email addresses (`someone@cs.colorado.edu`).

Number expressions introduce other complications as well; while commas normally appear at word boundaries, commas are used inside numbers in English, every three digits: `555,500.50`. Languages, and hence tokenization requirements, differ on this; many continental European languages like Spanish, French, and German, by contrast, use a comma to mark the decimal point, and spaces (or sometimes periods) where English puts commas, for example, `555 500,50`.

clitic

A tokenizer can also be used to expand **clitic** contractions that are marked by apostrophes, for example, converting `what're` to the two tokens `what are`, and `we're` to `we are`. A clitic is a part of a word that can't stand on its own, and can only occur when it is attached to another word. Some such contractions occur in other alphabetic languages, including articles and pronouns in French (`j'ai`, `l'homme`).

Depending on the application, tokenization algorithms may also tokenize multiword expressions like `New York` or `rock 'n' roll` as a single token, which requires a multiword expression dictionary of some sort. Tokenization is thus intimately tied up with **named entity detection**, the task of detecting names, dates, and organizations (Chapter 17).

Penn Treebank tokenization

One commonly used tokenization standard is known as the **Penn Treebank tokenization** standard, used for the parsed corpora (treebanks) released by the Linguistic Data Consortium (LDC), the source of many useful datasets. This standard separates out clitics (*doesn't* becomes *does* plus *n't*), keeps hyphenated words together, and separates out all punctuation:

Input: "The San Francisco-based restaurant," they said, "doesn't charge $10".

Output:

“	The	San	Francisco-based	restaurant	,	”	they	
said	,	“	does	n't	charge	$	10	”	.

Tokens can also be **normalized**, in which a single normalized form is chosen for words with multiple forms like `USA` and `US` or `uh-huh` and `uhhuh`. This standardization may be valuable, despite the spelling information that is lost in the normalization process. For information retrieval, we might want a query for `US` to match a document that has `USA`; for information extraction we might want to extract coherent information that is consistent across differently-spelled instances.

case folding

Case folding is another kind of normalization. For tasks like speech recognition and information retrieval, everything is mapped to lower case. For sentiment analysis and other text classification tasks, information extraction, and machine translation, by contrast, case is quite helpful and case folding is generally not done (losing the difference, for example, between *US* the country and *us* the pronoun can outweigh the advantage in generality that case folding provides).

In practice, since tokenization needs to be run before any other language processing, it is important for it to be very fast. The standard method for tokenization/normalization is therefore to use deterministic algorithms based on regular expressions compiled into very efficient finite state automata. Carefully designed deterministic algorithms can deal with the ambiguities that arise, such as the fact that the apostrophe needs to be tokenized differently when used as a genitive marker (as in *the*

book's cover), a quotative as in *'The other class', she said*, or in clitics like *they're*.

2.4.3 Word Segmentation in Chinese: the MaxMatch algorithm

hanzi

maximum matching

Some languages, including written Chinese, Japanese, and Thai, do not use spaces to mark potential word-boundaries, and so require alternative segmentation methods. In Chinese, for example, words are composed of characters known as **hanzi**. Each character generally represents a single morpheme and is pronounceable as a single syllable. Words are about 2.4 characters long on average. A simple algorithm that does remarkably well for segmenting Chinese, and often used as a baseline comparison for more advanced methods, is a version of greedy search called **maximum matching** or sometimes **MaxMatch**. The algorithm requires a dictionary (wordlist) of the language.

The maximum matching algorithm starts by pointing at the beginning of a string. It chooses the longest word in the dictionary that matches the input at the current position. The pointer is then advanced to the end of that word in the string. If no word matches, the pointer is instead advanced one character (creating a one-character word). The algorithm is then iteratively applied again starting from the new pointer position. Fig. 2.11 shows a version of the algorithm.

```
function MAXMATCH(sentence, dictionary) returns word sequence W

    if sentence is empty
        return empty list
    for i ← length(sentence) downto 1
        firstword = first i chars of sentence
        remainder = rest of sentence
        if InDictionary(firstword, dictionary)
            return list(firstword, MaxMatch(remainder,dictionary) )

     # no word was found, so make a one-character word
    firstword = first char of sentence
    remainder = rest of sentence
    return list(firstword, MaxMatch(remainder,dictionary) )
```

Figure 2.11 The MaxMatch algorithm for word segmentation.

MaxMatch works very well on Chinese; the following example shows an application to a simple Chinese sentence using a simple Chinese lexicon available from the Linguistic Data Consortium:

Input:	他特别喜欢北京烤鸭				"He especially likes Peking duck"
Output:	他	特别	喜欢	北京烤鸭	
	He	especially	likes	Peking duck	

MaxMatch doesn't work as well on English. To make the intuition clear, we'll create an example by removing the spaces from the beginning of Turing's famous quote "We can only see a short distance ahead", producing "wecanonlyseeashortdistanceahead". The MaxMatch results are shown below.

Input: `wecanonlyseeashortdistanceahead`
Output: `we canon l y see ash ort distance ahead`

On English the algorithm incorrectly chose `canon` instead of stopping at `can`, which left the algorithm confused and having to create single-character words `l` and

y and use the very rare word ort.

The algorithm works better in Chinese than English, because Chinese has much shorter words than English. We can quantify how well a segmenter works using a metric called **word error rate**. We compare our output segmentation with a perfect hand-segmented ('gold') sentence, seeing how many words differ. The word error rate is then the normalized minimum edit distance in words between our output and the gold: the number of word insertions, deletions, and substitutions divided by the length of the gold sentence in words; we'll see in Section 2.5 how to compute edit distance. Even in Chinese, however, MaxMatch has problems, for example dealing with **unknown words** (words not in the dictionary) or genres that differ a lot from the assumptions made by the dictionary builder.

word error rate

The most accurate Chinese segmentation algorithms generally use statistical **sequence models** trained via supervised machine learning on hand-segmented training sets; we'll introduce sequence models in Chapter 8.

2.4.4 Lemmatization and Stemming

Lemmatization is the task of determining that two words have the same root, despite their surface differences. The words *am*, *are*, and *is* have the shared lemma *be*; the words *dinner* and *dinners* both have the lemma *dinner*. Representing a word by its lemma is important for web search, since we want to find pages mentioning *woodchucks* if we search for *woodchuck*. This is especially important in morphologically complex languages like Russian, where for example the word *Moscow* has different endings in the phrases *Moscow*, *of Moscow*, *from Moscow*, and so on. Lemmatizing each of these forms to the same lemma will let us find all mentions of Moscow. The lemmatized form of a sentence like *He is reading detective stories* would thus be *He be read detective story*.

How is lemmatization done? The most sophisticated methods for lemmatization involve complete **morphological parsing** of the word. **Morphology** is the study of the way words are built up from smaller meaning-bearing units called **morphemes**. Two broad classes of morphemes can be distinguished: **stems**—the central morpheme of the word, supplying the main meaning— and **affixes**—adding "additional" meanings of various kinds. So, for example, the word *fox* consists of one morpheme (the morpheme *fox*) and the word *cats* consists of two: the morpheme *cat* and the morpheme *-s*. A morphological parser takes a word like *cats* and parses it into the two morphemes *cat* and *s*, or a Spanish word like *amaren* ('if in the future they would love') into the morphemes *amar* 'to love', *3PL*, and *future subjunctive*.

morpheme
stem
affix

The Porter Stemmer

While using finite-state transducers to build a full morphological parser is the most general way to deal with morphological variation in word forms, we sometimes make use of simpler but cruder chopping off of affixes. This naive version of morphological analysis is called **stemming**, and one of the most widely used stemming algorithms is the simple and efficient Porter (1980) algorithm. The Porter stemmer applied to the following paragraph:

stemming
Porter stemmer

```
This was not the map we found in Billy Bones's chest, but
an accurate copy, complete in all things-names and heights
and soundings-with the single exception of the red crosses
and the written notes.
```

produces the following stemmed output:

```
Thi wa not the map we found in Billi Bone s chest but an
accur copi complet in all thing name and height and sound
with the singl except of the red cross and the written note
```

cascade

The algorithm is based on series of rewrite rules run in series, as a **cascade**, in which the output of each pass is fed as input to the next pass; here is a sampling of the rules:

$$\begin{aligned}
\text{ATIONAL} &\rightarrow \text{ATE} \quad \text{(e.g., relational} \rightarrow \text{relate)} \\
\text{ING} &\rightarrow \epsilon \quad \text{if stem contains vowel (e.g., motoring} \rightarrow \text{motor)} \\
\text{SSES} &\rightarrow \text{SS} \quad \text{(e.g., grasses} \rightarrow \text{grass)}
\end{aligned}$$

Detailed rule lists for the Porter stemmer, as well as code (in Java, Python, etc.) can be found on Martin Porter's homepage; see also the original paper (Porter, 1980).

Simple stemmers can be useful in cases where we need to collapse across different variants of the same lemma. Nonetheless, they do tend to commit errors of both over- and under-generalizing, as shown in the table below (Krovetz, 1993):

Errors of Commission		**Errors of Omission**	
organization	organ	European	Europe
doing	doe	analysis	analyzes
numerical	numerous	noise	noisy
policy	police	sparse	sparsity

2.4.5 Sentence Segmentation

Sentence segmentation

Sentence segmentation is another important step in text processing. The most useful cues for segmenting a text into sentences are punctuation, like periods, question marks, exclamation points. Question marks and exclamation points are relatively unambiguous markers of sentence boundaries. Periods, on the other hand, are more ambiguous. The period character "." is ambiguous between a sentence boundary marker and a marker of abbreviations like *Mr.* or *Inc.* The previous sentence that you just read showed an even more complex case of this ambiguity, in which the final period of *Inc.* marked both an abbreviation and the sentence boundary marker. For this reason, sentence tokenization and word tokenization may be addressed jointly.

In general, sentence tokenization methods work by building a binary classifier (based on a sequence of rules or on machine learning) that decides if a period is part of the word or is a sentence-boundary marker. In making this decision, it helps to know if the period is attached to a commonly used abbreviation; thus, an abbreviation dictionary is useful.

State-of-the-art methods for sentence tokenization are based on machine learning and are introduced in later chapters.

2.5 Minimum Edit Distance

Much of natural language processing is concerned with measuring how similar two strings are. For example in spelling correction, the user typed some erroneous string—let's say `graffe`–and we want to know what the user meant. The user probably intended a word that is similar to `graffe`. Among candidate similar words,

the word `giraffe`, which differs by only one letter from `graffe`, seems intuitively to be more similar than, say `grail` or `graf`, which differ in more letters. Another example comes from **coreference**, the task of deciding whether two strings such as the following refer to the same entity:

```
Stanford President John Hennessy
Stanford University President John Hennessy
```

Again, the fact that these two strings are very similar (differing by only one word) seems like useful evidence for deciding that they might be coreferent.

minimum edit distance

Edit distance gives us a way to quantify both of these intuitions about string similarity. More formally, the **minimum edit distance** between two strings is defined as the minimum number of editing operations (operations like insertion, deletion, substitution) needed to transform one string into another.

alignment

The gap between *intention* and *execution*, for example, is 5 (delete an `i`, substitute `e` for `n`, substitute `x` for `t`, insert `c`, substitute `u` for `n`). It's much easier to see this by looking at the most important visualization for string distances, an **alignment** between the two strings, shown in Fig. 2.12. Given two sequences, an **alignment** is a correspondence between substrings of the two sequences. Thus, we say `I` **aligns** with the empty string, `N` with `E`, and so on. Beneath the aligned strings is another representation; a series of symbols expressing an **operation list** for converting the top string into the bottom string: **d** for deletion, **s** for substitution, **i** for insertion.

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

Figure 2.12 Representing the minimum edit distance between two strings as an **alignment**. The final row gives the operation list for converting the top string into the bottom string: d for deletion, s for substitution, i for insertion.

We can also assign a particular cost or weight to each of these operations. The **Levenshtein** distance between two sequences is the simplest weighting factor in which each of the three operations has a cost of 1 (Levenshtein, 1966)—we assume that the substitution of a letter for itself, for example, `t` for `t`, has zero cost. The Levenshtein distance between *intention* and *execution* is 5. Levenshtein also proposed an alternative version of his metric in which each insertion or deletion has a cost of 1 and substitutions are not allowed. (This is equivalent to allowing substitution, but giving each substitution a cost of 2 since any substitution can be represented by one insertion and one deletion). Using this version, the Levenshtein distance between *intention* and *execution* is 8.

2.5.1 The Minimum Edit Distance Algorithm

How do we find the minimum edit distance? We can think of this as a search task, in which we are searching for the shortest path—a sequence of edits—from one string to another.

The space of all possible edits is enormous, so we can't search naively. However, lots of distinct edit paths will end up in the same state (string), so rather than recomputing all those paths, we could just remember the shortest path to a state each time

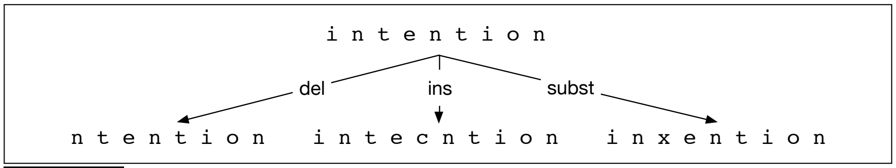


Figure 2.13 Finding the edit distance viewed as a search problem

dynamic programming

we saw it. We can do this by using **dynamic programming**. Dynamic programming is the name for a class of algorithms, first introduced by Bellman (1957), that apply a table-driven method to solve problems by combining solutions to sub-problems. Some of the most commonly used algorithms in natural language processing make use of dynamic programming, such as the **Viterbi** algorithm (Chapter 8) and the **CKY** algorithm for parsing (Chapter 11).

The intuition of a dynamic programming problem is that a large problem can be solved by properly combining the solutions to various sub-problems. Consider the shortest path of transformed words that represents the minimum edit distance between the strings *intention* and *execution* shown in Fig. 2.14.

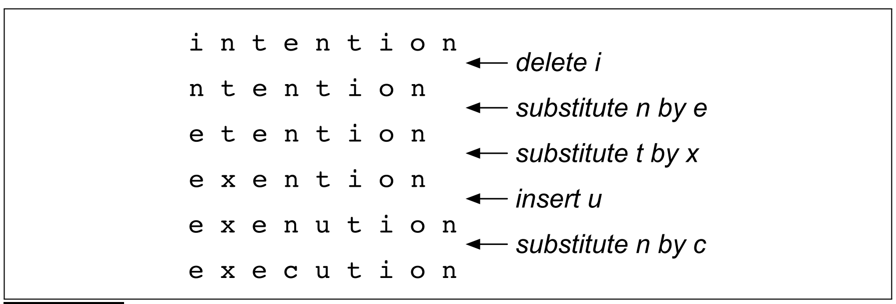


Figure 2.14 Path from *intention* to *execution*.

Imagine some string (perhaps it is *exention*) that is in this optimal path (whatever it is). The intuition of dynamic programming is that if *exention* is in the optimal operation list, then the optimal sequence must also include the optimal path from *intention* to *exention*. Why? If there were a shorter path from *intention* to *exention*, then we could use it instead, resulting in a shorter overall path, and the optimal sequence wouldn't be optimal, thus leading to a contradiction.

minimum edit distance

The **minimum edit distance** algorithm was named by Wagner and Fischer (1974) but independently discovered by many people (see the Historical Notes section of Chapter 8).

Let's first define the minimum edit distance between two strings. Given two strings, the source string X of length n, and target string Y of length m, we'll define $D(i,j)$ as the edit distance between $X[1..i]$ and $Y[1..j]$, i.e., the first i characters of X and the first j characters of Y. The edit distance between X and Y is thus $D(n,m)$.

We'll use dynamic programming to compute $D(n,m)$ bottom up, combining solutions to subproblems. In the base case, with a source substring of length i but an empty target string, going from i characters to 0 requires i deletes. With a target substring of length j but an empty source going from 0 characters to j characters requires j inserts. Having computed $D(i,j)$ for small i,j we then compute larger $D(i,j)$ based on previously computed smaller values. The value of $D(i,j)$ is computed by taking the minimum of the three possible paths through the matrix which arrive there:

$$D[i,j] = \min \begin{cases} D[i-1,j] + \text{del-cost}(\mathit{source}[i]) \\ D[i,j-1] + \text{ins-cost}(\mathit{target}[j]) \\ D[i-1,j-1] + \text{sub-cost}(\mathit{source}[i], \mathit{target}[j]) \end{cases}$$

If we assume the version of Levenshtein distance in which the insertions and deletions each have a cost of 1 (ins-cost(·) = del-cost(·) = 1), and substitutions have a cost of 2 (except substitution of identical letters have zero cost), the computation for $D(i,j)$ becomes:

$$D[i,j] = \min \begin{cases} D[i-1,j] + 1 \\ D[i,j-1] + 1 \\ D[i-1,j-1] + \begin{cases} 2; & \text{if } \mathit{source}[i] \neq \mathit{target}[j] \\ 0; & \text{if } \mathit{source}[i] = \mathit{target}[j] \end{cases} \end{cases} \tag{2.4}$$

The algorithm is summarized in Fig. 2.15; Fig. 2.16 shows the results of applying the algorithm to the distance between *intention* and *execution* with the version of Levenshtein in Eq. 2.4.

```
function MIN-EDIT-DISTANCE(source, target) returns min-distance

  n ← LENGTH(source)
  m ← LENGTH(target)
  Create a distance matrix distance[n+1,m+1]

  # Initialization: the zeroth row and column is the distance from the empty string
      D[0,0] = 0
      for each row i from 1 to n do
          D[i,0] ← D[i-1,0] + del-cost(source[i])
      for each column j from 1 to m do
          D[0,j] ← D[0, j-1] + ins-cost(target[j])

  # Recurrence relation:
  for each row i from 1 to n do
      for each column j from 1 to m do
         D[i, j] ← MIN( D[i−1, j] + del-cost(source[i]),
                        D[i−1, j−1] + sub-cost(source[i], target[j]),
                        D[i, j−1] + ins-cost(target[j]))
  # Termination
  return D[n,m]
```

Figure 2.15 The minimum edit distance algorithm, an example of the class of dynamic programming algorithms. The various costs can either be fixed (e.g., $\forall x, \text{ins-cost}(x) = 1$) or can be specific to the letter (to model the fact that some letters are more likely to be inserted than others). We assume that there is no cost for substituting a letter for itself (i.e., $\text{sub-cost}(x,x) = 0$).

Knowing the minimum edit distance is useful for algorithms like finding potential spelling error corrections. But the edit distance algorithm is important in another way; with a small change, it can also provide the minimum cost **alignment** between two strings. Aligning two strings is useful throughout speech and language processing. In speech recognition, minimum edit distance alignment is used to compute

Src\Tar	#	e	x	e	c	u	t	i	o	n
#	0	1	2	3	4	5	6	7	8	9
i	1	2	3	4	5	6	7	6	7	8
n	2	3	4	5	6	7	8	7	8	7
t	3	4	5	6	7	8	7	8	9	8
e	4	3	4	5	6	7	8	9	10	9
n	5	4	5	6	7	8	9	10	11	10
t	6	5	6	7	8	9	8	9	10	11
i	7	6	7	8	9	10	9	8	9	10
o	8	7	8	9	10	11	10	9	8	9
n	9	8	9	10	11	12	11	10	9	8

Figure 2.16 Computation of minimum edit distance between *intention* and *execution* with the algorithm of Fig. 2.15, using Levenshtein distance with cost of 1 for insertions or deletions, 2 for substitutions.

the word error rate (Chapter 27). Alignment plays a role in machine translation, in which sentences in a parallel corpus (a corpus with a text in two languages) need to be matched to each other.

To extend the edit distance algorithm to produce an alignment, we can start by visualizing an alignment as a path through the edit distance matrix. Figure 2.17 shows this path with the boldfaced cell. Each boldfaced cell represents an alignment of a pair of letters in the two strings. If two boldfaced cells occur in the same row, there will be an insertion in going from the source to the target; two boldfaced cells in the same column indicate a deletion.

Figure 2.17 also shows the intuition of how to compute this alignment path. The computation proceeds in two steps. In the first step, we augment the minimum edit distance algorithm to store backpointers in each cell. The backpointer from a cell points to the previous cell (or cells) that we came from in entering the current cell. We've shown a schematic of these backpointers in Fig. 2.17. Some cells have multiple backpointers because the minimum extension could have come from multiple previous cells. In the second step, we perform a **backtrace**. In a backtrace, we start from the last cell (at the final row and column), and follow the pointers back through the dynamic programming matrix. Each complete path between the final cell and the initial cell is a minimum distance alignment. Exercise 2.7 asks you to modify the minimum edit distance algorithm to store the pointers and compute the backtrace to output an alignment.

backtrace

While we worked our example with simple Levenshtein distance, the algorithm in Fig. 2.15 allows arbitrary weights on the operations. For spelling correction, for example, substitutions are more likely to happen between letters that are next to each other on the keyboard. The **Viterbi** algorithm is a probabilistic extension of minimum edit distance. Instead of computing the "minimum edit distance" between two strings, Viterbi computes the "maximum probability alignment" of one string with another. We'll discuss this more in Chapter 8.

2.6 Summary

This chapter introduced a fundamental tool in language processing, the **regular expression**, and showed how to perform basic **text normalization** tasks including

	#	e	x	e	c	u	t	i	o	n
#	0	← 1	← 2	← 3	← 4	← 5	← 6	← 7	← 8	← 9
i	↑ **1**	↖←↑ 2	↖←↑ 3	↖←↑ 4	↖←↑ 5	↖←↑ 6	↖←↑ 7	↖ 6	← 7	← 8
n	↑ 2	↖←↑ **3**	↖←↑ 4	↖←↑ 5	↖←↑ 6	↖←↑ 7	↖←↑ 8	↑ 7	↖←↑ 8	↖ 7
t	↑ 3	↖←↑ 4	↖←↑ **5**	↖←↑ 6	↖←↑ 7	↖←↑ 8	↖ 7	←↑ 8	↖←↑ 9	↑ 8
e	↑ 4	↖ 3	← 4	↖← **5**	← **6**	← 7	←↑ 8	↖←↑ 9	↖←↑ 10	↑ 9
n	↑ 5	↑ 4	↖←↑ 5	↖←↑ 6	↖←↑ 7	↖←↑ **8**	↖←↑ 9	↖←↑ 10	↖←↑ 11	↖↑ 10
t	↑ 6	↑ 5	↖←↑ 6	↖←↑ 7	↖←↑ 8	↖←↑ 9	↖ **8**	← 9	← 10	←↑ 11
i	↑ 7	↑ 6	↖←↑ 7	↖←↑ 8	↖←↑ 9	↖←↑ 10	↑ 9	↖ **8**	← 9	← 10
o	↑ 8	↑ 7	↖←↑ 8	↖←↑ 9	↖←↑ 10	↖←↑ 11	↑ 10	↑ 9	↖ **8**	← 9
n	↑ 9	↑ 8	↖←↑ 9	↖←↑ 10	↖←↑ 11	↖←↑ 12	↑ 11	↑ 10	↑ 9	↖ **8**

Figure 2.17 When entering a value in each cell, we mark which of the three neighboring cells we came from with up to three arrows. After the table is full we compute an **alignment** (minimum edit path) by using a **backtrace**, starting at the **8** in the lower-right corner and following the arrows back. The sequence of bold cells represents one possible minimum cost alignment between the two strings. Diagram design after Gusfield (1997).

word segmentation and **normalization**, **sentence segmentation**, and **stemming**. We also introduce the important **minimum edit distance** algorithm for comparing strings. Here's a summary of the main points we covered about these ideas:

- The **regular expression** language is a powerful tool for pattern-matching.
- Basic operations in regular expressions include **concatenation** of symbols, **disjunction** of symbols (`[]`, `|`, and `.`), **counters** (`*`, `+`, and `{n,m}`), **anchors** (`^`, `$`) and precedence operators (`(`,`)`).
- **Word tokenization and normalization** are generally done by cascades of simple regular expressions substitutions or finite automata.
- The **Porter algorithm** is a simple and efficient way to do **stemming**, stripping off affixes. It does not have high accuracy but may be useful for some tasks.
- The **minimum edit distance** between two strings is the minimum number of operations it takes to edit one into the other. Minimum edit distance can be computed by **dynamic programming**, which also results in an **alignment** of the two strings.

Bibliographical and Historical Notes

Kleene (1951) and (1956) first defined regular expressions and the finite automaton, based on the McCulloch-Pitts neuron. Ken Thompson was one of the first to build regular expressions compilers into editors for text searching (Thompson, 1968). His editor *ed* included a command "g/regular expression/p", or Global Regular Expression Print, which later became the Unix `grep` utility.

Text normalization algorithms has been applied since the beginning of the field. One of the earliest widely-used stemmers was Lovins (1968). Stemming was also applied early to the digital humanities, by Packard (1973), who built an affix-stripping morphological parser for Ancient Greek. Currently a wide variety of code for tokenization and normalization is available, such as the Stanford Tokenizer (`http://nlp.stanford.edu/software/tokenizer.shtml`) or specialized tokenizers for Twitter (O'Connor et al., 2010), or for sentiment (`http://sentiment.christopherpotts.net/tokenizing.html`). See Palmer (2012) for a survey of text preprocessing. While the max-match algorithm we describe is commonly used as a segmentation

baseline in languages like Chinese, higher accuracy algorithms like the Stanford CRF segmenter, are based on sequence models; see Tseng et al. (2005a) and Chang et al. (2008). NLTK is an essential tool that offers both useful Python libraries (`http://www.nltk.org`) and textbook descriptions (Bird et al., 2009) of many algorithms including text normalization and corpus interfaces.

For more on Herdan's law and Heaps' Law, see Herdan (1960, p. 28), Heaps (1978), Egghe (2007) and Baayen (2001); Yasseri et al. (2012) discuss the relationship with other measures of linguistic complexity. For more on edit distance, see the excellent Gusfield (1997). Our example measuring the edit distance from 'intention' to 'execution' was adapted from Kruskal (1983). There are various publicly available packages to compute edit distance, including Unix `diff` and the NIST `sclite` program (NIST, 2005).

In his autobiography Bellman (1984) explains how he originally came up with the term *dynamic programming*:

> "...The 1950s were not good years for mathematical research. [the] Secretary of Defense ...had a pathological fear and hatred of the word, research... I decided therefore to use the word, "programming". I wanted to get across the idea that this was dynamic, this was multistage... I thought, let's ... take a word that has an absolutely precise meaning, namely dynamic... it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to."

Exercises

2.1 Write regular expressions for the following languages.

1. the set of all alphabetic strings;
2. the set of all lower case alphabetic strings ending in a *b*;
3. the set of all strings from the alphabet a, b such that each a is immediately preceded by and immediately followed by a b;

2.2 Write regular expressions for the following languages. By "word", we mean an alphabetic string separated from other words by whitespace, any relevant punctuation, line breaks, and so forth.

1. the set of all strings with two consecutive repeated words (e.g., "Humbert Humbert" and "the the" but not "the bug" or "the big bug");
2. all strings that start at the beginning of the line with an integer and that end at the end of the line with a word;
3. all strings that have both the word *grotto* and the word *raven* in them (but not, e.g., words like *grottos* that merely *contain* the word *grotto*);
4. write a pattern that places the first word of an English sentence in a register. Deal with punctuation.

2.3 Implement an ELIZA-like program, using substitutions such as those described on page 18. You might want to choose a different domain than a Rogerian psychologist, although keep in mind that you would need a domain in which your program can legitimately engage in a lot of simple repetition.

2.4 Compute the edit distance (using insertion cost 1, deletion cost 1, substitution cost 1) of “leda” to “deal”. Show your work (using the edit distance grid).

2.5 Figure out whether *drive* is closer to *brief* or to *divers* and what the edit distance is to each. You may use any version of *distance* that you like.

2.6 Now implement a minimum edit distance algorithm and use your hand-computed results to check your code.

2.7 Augment the minimum edit distance algorithm to output an alignment; you will need to store pointers and add a stage to compute the backtrace.

2.8 Implement the MaxMatch algorithm.

2.9 To test how well your MaxMatch algorithm works, create a test set by removing spaces from a set of sentences. Implement the Word Error Rate metric (the number of word insertions + deletions + substitutions, divided by the length in words of the correct string) and compute the WER for your test set.

CHAPTER

3 N-gram Language Models

"You are uniformly charming!" cried he, with a smile of associating and now and then I bowed and they perceived a chaise and four to wish for.

Random sentence generated from a Jane Austen trigram model

Being able to predict the future is not always a good thing. Cassandra of Troy had the gift of foreseeing but was cursed by Apollo that no one would believe her predictions. Her warnings of the destruction of Troy were ignored and—well, let's just say that things didn't turn out great for her.

In this chapter we take up the somewhat less fraught topic of predicting words. What word, for example, is likely to follow

```
Please turn your homework ...
```

Hopefully, most of you concluded that a very likely word is *in*, or possibly *over*, but probably not *refrigerator* or *the*. In the following sections we will formalize this intuition by introducing models that assign a **probability** to each possible next word. The same models will also serve to assign a probability to an entire sentence. Such a model, for example, could predict that the following sequence has a much higher probability of appearing in a text:

```
all of a sudden I notice three guys standing on the sidewalk
```

than does this same set of words in a different order:

```
on guys all I of notice sidewalk three a sudden standing the
```

Why would you want to predict upcoming words, or assign probabilities to sentences? Probabilities are essential in any task in which we have to identify words in noisy, ambiguous input, like **speech recognition** or **handwriting recognition**. In the movie *Take the Money and Run*, Woody Allen tries to rob a bank with a sloppily written hold-up note that the teller incorrectly reads as "I have a gub". As Russell and Norvig (2002) point out, a language processing system could avoid making this mistake by using the knowledge that the sequence "I have a gun" is far more probable than the non-word "I have a gub" or even "I have a gull".

In **spelling correction**, we need to find and correct spelling errors like *Their are two midterms in this class*, in which *There* was mistyped as *Their*. A sentence starting with the phrase *There are* will be much more probable than one starting with *Their are*, allowing a spellchecker to both detect and correct these errors.

Assigning probabilities to sequences of words is also essential in **machine translation**. Suppose we are translating a Chinese source sentence:

他	向	记者	介绍了	主要	内容
He	to	reporters	introduced	main	content

As part of the process we might have built the following set of potential rough English translations:

he introduced reporters to the main contents of the statement
he briefed to reporters the main contents of the statement
he briefed reporters on the main contents of the statement

A probabilistic model of word sequences could suggest that *briefed reporters on* is a more probable English phrase than *briefed to reporters* (which has an awkward *to* after *briefed*) or *introduced reporters to* (which uses a verb that is less fluent English in this context), allowing us to correctly select the boldfaced sentence above.

Probabilities are also important for **augmentative communication** (Newell et al., 1998) systems. People like the late physicist Stephen Hawking who are unable to physically talk or sign can instead use simple movements to select words from a menu to be spoken by the system. Word prediction can be used to suggest likely words for the menu.

language model
LM
n-gram

Models that assign probabilities to sequences of words are called **language models** or **LMs**. In this chapter we introduce the simplest model that assigns probabilities to sentences and sequences of words, the **n-gram**. An n-gram is a sequence of N words: a 2-gram (or **bigram**) is a two-word sequence of words like “please turn”, “turn your”, or ”your homework”, and a 3-gram (or **trigram**) is a three-word sequence of words like “please turn your”, or “turn your homework”. We’ll see how to use n-gram models to estimate the probability of the last word of an n-gram given the previous words, and also to assign probabilities to entire sequences. In a bit of terminological ambiguity, we usually drop the word “model”, and thus the term **n-gram** is used to mean either the word sequence itself or the predictive model that assigns it a probability.

3.1 N-Grams

Let’s begin with the task of computing $P(w|h)$, the probability of a word w given some history h. Suppose the history h is “*its water is so transparent that*” and we want to know the probability that the next word is *the*:

$$P(the|its\ water\ is\ so\ transparent\ that). \tag{3.1}$$

One way to estimate this probability is from relative frequency counts: take a very large corpus, count the number of times we see *its water is so transparent that*, and count the number of times this is followed by *the*. This would be answering the question “Out of the times we saw the history h, how many times was it followed by the word w”, as follows:

$$P(the|its\ water\ is\ so\ transparent\ that) = \frac{C(its\ water\ is\ so\ transparent\ that\ the)}{C(its\ water\ is\ so\ transparent\ that)} \tag{3.2}$$

With a large enough corpus, such as the web, we can compute these counts and estimate the probability from Eq. 3.2. You should pause now, go to the web, and compute this estimate for yourself.

While this method of estimating probabilities directly from counts works fine in many cases, it turns out that even the web isn’t big enough to give us good estimates

in most cases. This is because language is creative; new sentences are created all the time, and we won't always be able to count entire sentences. Even simple extensions of the example sentence may have counts of zero on the web (such as "*Walden Pond's water is so transparent that the*").

Similarly, if we wanted to know the joint probability of an entire sequence of words like *its water is so transparent*, we could do it by asking "out of all possible sequences of five words, how many of them are *its water is so transparent*?" We would have to get the count of *its water is so transparent* and divide by the sum of the counts of all possible five word sequences. That seems rather a lot to estimate!

For this reason, we'll need to introduce cleverer ways of estimating the probability of a word w given a history h, or the probability of an entire word sequence W. Let's start with a little formalizing of notation. To represent the probability of a particular random variable X_i taking on the value "the", or $P(X_i = \text{“the”})$, we will use the simplification $P(the)$. We'll represent a sequence of N words either as $w_1 \ldots w_n$ or w_1^n (so the expression w_1^{n-1} means the string $w_1, w_2, ..., w_{n-1}$). For the joint probability of each word in a sequence having a particular value $P(X = w_1, Y = w_2, Z = w_3, ..., W = w_n)$ we'll use $P(w_1, w_2, ..., w_n)$.

Now how can we compute probabilities of entire sequences like $P(w_1, w_2, ..., w_n)$? One thing we can do is decompose this probability using the **chain rule of probability**:

$$\begin{aligned} P(X_1 ... X_n) &= P(X_1)P(X_2|X_1)P(X_3|X_1^2)\ldots P(X_n|X_1^{n-1}) \\ &= \prod_{k=1}^{n} P(X_k|X_1^{k-1}) \end{aligned} \tag{3.3}$$

Applying the chain rule to words, we get

$$\begin{aligned} P(w_1^n) &= P(w_1)P(w_2|w_1)P(w_3|w_1^2)\ldots P(w_n|w_1^{n-1}) \\ &= \prod_{k=1}^{n} P(w_k|w_1^{k-1}) \end{aligned} \tag{3.4}$$

The chain rule shows the link between computing the joint probability of a sequence and computing the conditional probability of a word given previous words. Equation 3.4 suggests that we could estimate the joint probability of an entire sequence of words by multiplying together a number of conditional probabilities. But using the chain rule doesn't really seem to help us! We don't know any way to compute the exact probability of a word given a long sequence of preceding words, $P(w_n|w_1^{n-1})$. As we said above, we can't just estimate by counting the number of times every word occurs following every long string, because language is creative and any particular context might have never occurred before!

The intuition of the n-gram model is that instead of computing the probability of a word given its entire history, we can **approximate** the history by just the last few words.

bigram

The **bigram** model, for example, approximates the probability of a word given all the previous words $P(w_n|w_1^{n-1})$ by using only the conditional probability of the preceding word $P(w_n|w_{n-1})$. In other words, instead of computing the probability

$$P(\text{the}|\text{Walden Pond's water is so transparent that}) \tag{3.5}$$

we approximate it with the probability

$$P(\text{the}|\text{that}) \tag{3.6}$$

When we use a bigram model to predict the conditional probability of the next word, we are thus making the following approximation:

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-1}) \tag{3.7}$$

The assumption that the probability of a word depends only on the previous word is called a **Markov** assumption. Markov models are the class of probabilistic models that assume we can predict the probability of some future unit without looking too far into the past. We can generalize the bigram (which looks one word into the past) to the trigram (which looks two words into the past) and thus to the **n-gram** (which looks $n-1$ words into the past).

Markov

n-gram

Thus, the general equation for this n-gram approximation to the conditional probability of the next word in a sequence is

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-N+1}^{n-1}) \tag{3.8}$$

Given the bigram assumption for the probability of an individual word, we can compute the probability of a complete word sequence by substituting Eq. 3.7 into Eq. 3.4:

$$P(w_1^n) \approx \prod_{k=1}^{n} P(w_k|w_{k-1}) \tag{3.9}$$

How do we estimate these bigram or n-gram probabilities? An intuitive way to estimate probabilities is called **maximum likelihood estimation** or **MLE**. We get the MLE estimate for the parameters of an n-gram model by getting counts from a corpus, and **normalizing** the counts so that they lie between 0 and 1.[1]

maximum likelihood estimation

normalize

For example, to compute a particular bigram probability of a word y given a previous word x, we'll compute the count of the bigram $C(xy)$ and normalize by the sum of all the bigrams that share the same first word x:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)} \tag{3.10}$$

We can simplify this equation, since the sum of all bigram counts that start with a given word w_{n-1} must be equal to the unigram count for that word w_{n-1} (the reader should take a moment to be convinced of this):

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \tag{3.11}$$

Let's work through an example using a mini-corpus of three sentences. We'll first need to augment each sentence with a special symbol <s> at the beginning of the sentence, to give us the bigram context of the first word. We'll also need a special end-symbol. </s>[2]

[1] For probabilistic models, normalizing means dividing by some total count so that the resulting probabilities fall legally between 0 and 1.

[2] We need the end-symbol to make the bigram grammar a true probability distribution. Without an end-symbol, the sentence probabilities for all sentences of a given length would sum to one. This model would define an infinite set of probability distributions, with one distribution per sentence length. See Exercise 3.5.

```
<s> I am Sam </s>
<s> Sam I am </s>
<s> I do not like green eggs and ham </s>
```

Here are the calculations for some of the bigram probabilities from this corpus

$P(\texttt{I}|\texttt{<s>}) = \frac{2}{3} = .67 \qquad P(\texttt{Sam}|\texttt{<s>}) = \frac{1}{3} = .33 \qquad P(\texttt{am}|\texttt{I}) = \frac{2}{3} = .67$
$P(\texttt{</s>}|\texttt{Sam}) = \frac{1}{2} = 0.5 \qquad P(\texttt{Sam}|\texttt{am}) = \frac{1}{2} = .5 \qquad P(\texttt{do}|\texttt{I}) = \frac{1}{3} = .33$

For the general case of MLE n-gram parameter estimation:

$$P(w_n|w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+1}^{n-1}w_n)}{C(w_{n-N+1}^{n-1})} \tag{3.12}$$

relative frequency

Equation 3.12 (like Eq. 3.11) estimates the n-gram probability by dividing the observed frequency of a particular sequence by the observed frequency of a prefix. This ratio is called a **relative frequency**. We said above that this use of relative frequencies as a way to estimate probabilities is an example of maximum likelihood estimation or MLE. In MLE, the resulting parameter set maximizes the likelihood of the training set T given the model M (i.e., $P(T|M)$). For example, suppose the word *Chinese* occurs 400 times in a corpus of a million words like the Brown corpus. What is the probability that a random word selected from some other text of, say, a million words will be the word *Chinese*? The MLE of its probability is $\frac{400}{1000000}$ or .0004. Now .0004 is not the best possible estimate of the probability of *Chinese* occurring in all situations; it might turn out that in some other corpus or context *Chinese* is a very unlikely word. But it is the probability that makes it *most likely* that Chinese will occur 400 times in a million-word corpus. We present ways to modify the MLE estimates slightly to get better probability estimates in Section 3.4.

Let's move on to some examples from a slightly larger corpus than our 14-word example above. We'll use data from the now-defunct Berkeley Restaurant Project, a dialogue system from the last century that answered questions about a database of restaurants in Berkeley, California (Jurafsky et al., 1994). Here are some text-normalized sample user queries (a sample of 9332 sentences is on the website):

can you tell me about any good cantonese restaurants close by
mid priced thai food is what i'm looking for
tell me about chez panisse
can you give me a listing of the kinds of food that are available
i'm looking for a good place to eat breakfast
when is caffe venezia open during the day

Figure 3.1 shows the bigram counts from a piece of a bigram grammar from the Berkeley Restaurant Project. Note that the majority of the values are zero. In fact, we have chosen the sample words to cohere with each other; a matrix selected from a random set of seven words would be even more sparse.

Figure 3.2 shows the bigram probabilities after normalization (dividing each cell in Fig. 3.1 by the appropriate unigram for its row, taken from the following set of unigram probabilities):

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

Here are a few other useful probabilities:

	i	**want**	**to**	**eat**	**chinese**	**food**	**lunch**	**spend**
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

Figure 3.1 Bigram counts for eight of the words (out of $V = 1446$) in the Berkeley Restaurant Project corpus of 9332 sentences. Zero counts are in gray.

	i	**want**	**to**	**eat**	**chinese**	**food**	**lunch**	**spend**
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

Figure 3.2 Bigram probabilities for eight words in the Berkeley Restaurant Project corpus of 9332 sentences. Zero probabilities are in gray.

$P(\texttt{i}|\texttt{<s>}) = 0.25$ $\quad P(\texttt{english}|\texttt{want}) = 0.0011$
$P(\texttt{food}|\texttt{english}) = 0.5$ $\quad P(\texttt{</s>}|\texttt{food}) = 0.68$

Now we can compute the probability of sentences like *I want English food* or *I want Chinese food* by simply multiplying the appropriate bigram probabilities together, as follows:

$$
\begin{aligned}
& P(\texttt{<s> i want english food </s>}) \\
& \quad = P(\texttt{i}|\texttt{<s>})P(\texttt{want}|\texttt{i})P(\texttt{english}|\texttt{want}) \\
& \qquad\qquad P(\texttt{food}|\texttt{english})P(\texttt{</s>}|\texttt{food}) \\
& \quad = .25 \times .33 \times .0011 \times 0.5 \times 0.68 \\
& \quad = .000031
\end{aligned}
$$

We leave it as Exercise 3.2 to compute the probability of *i want chinese food*.

What kinds of linguistic phenomena are captured in these bigram statistics? Some of the bigram probabilities above encode some facts that we think of as strictly **syntactic** in nature, like the fact that what comes after *eat* is usually a noun or an adjective, or that what comes after *to* is usually a verb. Others might be a fact about the personal assistant task, like the high probability of sentences beginning with the words *I*. And some might even be cultural rather than linguistic, like the higher probability that people are looking for Chinese versus English food.

trigram
4-gram
5-gram

Some practical issues: Although for pedagogical purposes we have only described bigram models, in practice it's more common to use **trigram** models, which condition on the previous two words rather than the previous word, or **4-gram** or even **5-gram** models, when there is sufficient training data. Note that for these larger n-grams, we'll need to assume extra context for the contexts to the left and right of the

sentence end. For example, to compute trigram probabilities at the very beginning of the sentence, we can use two pseudo-words for the first trigram (i.e., $P(\texttt{I|<s><s>})$.

log probabilities

We always represent and compute language model probabilities in log format as **log probabilities**. Since probabilities are (by definition) less than or equal to 1, the more probabilities we multiply together, the smaller the product becomes. Multiplying enough n-grams together would result in numerical underflow. By using log probabilities instead of raw probabilities, we get numbers that are not as small. Adding in log space is equivalent to multiplying in linear space, so we combine log probabilities by adding them. The result of doing all computation and storage in log space is that we only need to convert back into probabilities if we need to report them at the end; then we can just take the exp of the logprob:

$$p_1 \times p_2 \times p_3 \times p_4 = \exp(\log p_1 + \log p_2 + \log p_3 + \log p_4) \tag{3.13}$$

3.2 Evaluating Language Models

extrinsic evaluation

The best way to evaluate the performance of a language model is to embed it in an application and measure how much the application improves. Such end-to-end evaluation is called **extrinsic evaluation**. Extrinsic evaluation is the only way to know if a particular improvement in a component is really going to help the task at hand. Thus, for speech recognition, we can compare the performance of two language models by running the speech recognizer twice, once with each language model, and seeing which gives the more accurate transcription.

intrinsic evaluation

Unfortunately, running big NLP systems end-to-end is often very expensive. Instead, it would be nice to have a metric that can be used to quickly evaluate potential improvements in a language model. An **intrinsic evaluation** metric is one that measures the quality of a model independent of any application.

training set

test set

held out

For an intrinsic evaluation of a language model we need a **test set**. As with many of the statistical models in our field, the probabilities of an n-gram model come from the corpus it is trained on, the **training set** or **training corpus**. We can then measure the quality of an n-gram model by its performance on some unseen data called the **test set** or test corpus. We will also sometimes call test sets and other datasets that are not in our training sets **held out** corpora because we hold them out from the training data.

So if we are given a corpus of text and want to compare two different n-gram models, we divide the data into training and test sets, train the parameters of both models on the training set, and then compare how well the two trained models fit the test set.

But what does it mean to "fit the test set"? The answer is simple: whichever model assigns a **higher probability** to the test set—meaning it more accurately predicts the test set—is a better model. Given two probabilistic models, the better model is the one that has a tighter fit to the test data or that better predicts the details of the test data, and hence will assign a higher probability to the test data.

Since our evaluation metric is based on test set probability, it's important not to let the test sentences into the training set. Suppose we are trying to compute the probability of a particular "test" sentence. If our test sentence is part of the training corpus, we will mistakenly assign it an artificially high probability when it occurs in the test set. We call this situation **training on the test set**. Training on the test set introduces a bias that makes the probabilities all look too high, and causes huge

inaccuracies in **perplexity**, the probability-based metric we introduce below.

Sometimes we use a particular test set so often that we implicitly tune to its characteristics. We then need a fresh test set that is truly unseen. In such cases, we call the initial test set the **development** test set or, **devset**. How do we divide our data into training, development, and test sets? We want our test set to be as large as possible, since a small test set may be accidentally unrepresentative, but we also want as much training data as possible. At the minimum, we would want to pick the smallest test set that gives us enough statistical power to measure a statistically significant difference between two potential models. In practice, we often just divide our data into 80% training, 10% development, and 10% test. Given a large corpus that we want to divide into training and test, test data can either be taken from some continuous sequence of text inside the corpus, or we can remove smaller "stripes" of text from randomly selected parts of our corpus and combine them into a test set.

development test

3.2.1 Perplexity

perplexity

In practice we don't use raw probability as our metric for evaluating language models, but a variant called **perplexity**. The **perplexity** (sometimes called *PP* for short) of a language model on a test set is the inverse probability of the test set, normalized by the number of words. For a test set $W = w_1 w_2 \ldots w_N$,:

$$\begin{aligned} \mathrm{PP}(W) &= P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}} \end{aligned} \tag{3.14}$$

We can use the chain rule to expand the probability of W:

$$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1 \ldots w_{i-1})}} \tag{3.15}$$

Thus, if we are computing the perplexity of W with a bigram language model, we get:

$$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}} \tag{3.16}$$

Note that because of the inverse in Eq. 3.15, the higher the conditional probability of the word sequence, the lower the perplexity. Thus, minimizing perplexity is equivalent to maximizing the test set probability according to the language model. What we generally use for word sequence in Eq. 3.15 or Eq. 3.16 is the entire sequence of words in some test set. Since this sequence will cross many sentence boundaries, we need to include the begin- and end-sentence markers `<s>` and `</s>` in the probability computation. We also need to include the end-of-sentence marker `</s>` (but not the beginning-of-sentence marker `<s>`) in the total count of word tokens N.

There is another way to think about perplexity: as the **weighted average branching factor** of a language. The branching factor of a language is the number of possible next words that can follow any word. Consider the task of recognizing the digits

in English (zero, one, two,..., nine), given that each of the 10 digits occurs with equal probability $P = \frac{1}{10}$. The perplexity of this mini-language is in fact 10. To see that, imagine a string of digits of length N. By Eq. 3.15, the perplexity will be

$$\begin{aligned} \mathrm{PP}(W) &= P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} \\ &= (\frac{1}{10}^N)^{-\frac{1}{N}} \\ &= \frac{1}{10}^{-1} \\ &= 10 \end{aligned} \tag{3.17}$$

But suppose that the number zero is really frequent and occurs 10 times more often than other numbers. Now we should expect the perplexity to be lower since most of the time the next number will be zero. Thus, although the branching factor is still 10, the perplexity or weighted branching factor is smaller. We leave this calculation as an exercise to the reader.

We see in Section 3.7 that perplexity is also closely related to the information-theoretic notion of entropy.

Finally, let's look at an example of how perplexity can be used to compare different n-gram models. We trained unigram, bigram, and trigram grammars on 38 million words (including start-of-sentence tokens) from the *Wall Street Journal*, using a 19,979 word vocabulary. We then computed the perplexity of each of these models on a test set of 1.5 million words with Eq. 3.16. The table below shows the perplexity of a 1.5 million word WSJ test set according to each of these grammars.

	Unigram	**Bigram**	**Trigram**
Perplexity	962	170	109

As we see above, the more information the n-gram gives us about the word sequence, the lower the perplexity (since as Eq. 3.15 showed, perplexity is related inversely to the likelihood of the test sequence according to the model).

Note that in computing perplexities, the n-gram model P must be constructed without any knowledge of the test set or any prior knowledge of the vocabulary of the test set. Any kind of knowledge of the test set can cause the perplexity to be artificially low. The perplexity of two language models is only comparable if they use identical vocabularies.

An (intrinsic) improvement in perplexity does not guarantee an (extrinsic) improvement in the performance of a language processing task like speech recognition or machine translation. Nonetheless, because perplexity often correlates with such improvements, it is commonly used as a quick check on an algorithm. But a model's improvement in perplexity should always be confirmed by an end-to-end evaluation of a real task before concluding the evaluation of the model.

3.3 Generalization and Zeros

The n-gram model, like many statistical models, is dependent on the training corpus. One implication of this is that the probabilities often encode specific facts about a given training corpus. Another implication is that n-grams do a better and better job of modeling the training corpus as we increase the value of N.

We can visualize both of these facts by borrowing the technique of Shannon (1951) and Miller and Selfridge (1950) of generating random sentences from different n-gram models. It's simplest to visualize how this works for the unigram case. Imagine all the words of the English language covering the probability space between 0 and 1, each word covering an interval proportional to its frequency. We choose a random value between 0 and 1 and print the word whose interval includes this chosen value. We continue choosing random numbers and generating words until we randomly generate the sentence-final token </s>. We can use the same technique to generate bigrams by first generating a random bigram that starts with <s> (according to its bigram probability). Let's say the second word of that bigram is w. We next chose a random bigram starting with w (again, drawn according to its bigram probability), and so on.

To give an intuition for the increasing power of higher-order n-grams, Fig. 3.3 shows random sentences generated from unigram, bigram, trigram, and 4-gram models trained on Shakespeare's works.

1 gram	–To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have –Hill he late speaks; or! a more to leg less first you enter
2 gram	–Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow. –What means, sir. I confess she? then all sorts, he is trim, captain.
3 gram	–Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done. –This shall forbid it should be branded, if renown made it empty.
4 gram	–King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in; –It cannot be but so.

Figure 3.3 Eight sentences randomly generated from four N-grams computed from Shakespeare's works. All characters were mapped to lower-case and punctuation marks were treated as words. Output is hand-corrected for capitalization to improve readability.

The longer the context on which we train the model, the more coherent the sentences. In the unigram sentences, there is no coherent relation between words or any sentence-final punctuation. The bigram sentences have some local word-to-word coherence (especially if we consider that punctuation counts as a word). The trigram and 4-gram sentences are beginning to look a lot like Shakespeare. Indeed, a careful investigation of the 4-gram sentences shows that they look a little too much like Shakespeare. The words *It cannot be but so* are directly from *King John*. This is because, not to put the knock on Shakespeare, his oeuvre is not very large as corpora go ($N = 884,647, V = 29,066$), and our n-gram probability matrices are ridiculously sparse. There are $V^2 = 844,000,000$ possible bigrams alone, and the number of possible 4-grams is $V^4 = 7 \times 10^{17}$. Thus, once the generator has chosen the first 4-gram (*It cannot be but*), there are only five possible continuations (*that*, *I*, *he*, *thou*, and *so*); indeed, for many 4-grams, there is only one continuation.

To get an idea of the dependence of a grammar on its training set, let's look at an n-gram grammar trained on a completely different corpus: the *Wall Street Journal* (WSJ) newspaper. Shakespeare and the *Wall Street Journal* are both English, so we might expect some overlap between our n-grams for the two genres. Fig. 3.4

shows sentences generated by unigram, bigram, and trigram grammars trained on 40 million words from WSJ.

1 gram	Months the my and issue of year foreign new exchange's september were recession exchange new endorsed a acquire to six executives
2 gram	Last December through the way to preserve the Hudson corporation N. B. E. C. Taylor would seem to complete the major central planners one point five percent of U. S. E. has already old M. X. corporation of living on information such as more frequently fishing to keep her
3 gram	They also point to ninety nine point six billion dollars from two hundred four oh six three percent of the rates of interest stores as Mexico and Brazil on market conditions

Figure 3.4 Three sentences randomly generated from three n-gram models computed from 40 million words of the *Wall Street Journal*, lower-casing all characters and treating punctuation as words. Output was then hand-corrected for capitalization to improve readability.

Compare these examples to the pseudo-Shakespeare in Fig. 3.3. While they both model "English-like sentences", there is clearly no overlap in generated sentences, and little overlap even in small phrases. Statistical models are likely to be pretty useless as predictors if the training sets and the test sets are as different as Shakespeare and WSJ.

How should we deal with this problem when we build n-gram models? One step is to be sure to use a training corpus that has a similar **genre** to whatever task we are trying to accomplish. To build a language model for translating legal documents, we need a training corpus of legal documents. To build a language model for a question-answering system, we need a training corpus of questions.

It is equally important to get training data in the appropriate **dialect**, especially when processing social media posts or spoken transcripts. Thus tweets in AAVE (African American Vernacular English) often use words like *finna*—an auxiliary verb that markes immediate future tense —that don't occur in other dialects, or spellings like *den* for *then*, in tweets like this one (Blodgett and O'Connor, 2017):

(3.18) Bored af den my phone finna die!!!

while tweets from varieties like Nigerian English have markedly different vocabulary and n-gram patterns from American English (Jurgens et al., 2017):

(3.19) @username R u a wizard or wat gan sef: in d mornin - u tweet, afternoon - u tweet, nyt gan u dey tweet. beta get ur IT placement wiv twitter

Matching genres and dialects is still not sufficient. Our models may still be subject to the problem of **sparsity**. For any n-gram that occurred a sufficient number of times, we might have a good estimate of its probability. But because any corpus is limited, some perfectly acceptable English word sequences are bound to be missing from it. That is, we'll have many cases of putative "zero probability n-grams" that should really have some non-zero probability. Consider the words that follow the bigram *denied the* in the WSJ Treebank3 corpus, together with their counts:

denied the allegations:	5
denied the speculation:	2
denied the rumors:	1
denied the report:	1

But suppose our test set has phrases like:

denied the offer
denied the loan

Our model will incorrectly estimate that the $P(\text{offer}|\text{denied the})$ is 0!

zeros

These **zeros**— things that don't ever occur in the training set but do occur in the test set—are a problem for two reasons. First, their presence means we are underestimating the probability of all sorts of words that might occur, which will hurt the performance of any application we want to run on this data.

Second, if the probability of any word in the test set is 0, the entire probability of the test set is 0. By definition, perplexity is based on the inverse probability of the test set. Thus if some words have zero probability, we can't compute perplexity at all, since we can't divide by 0!

3.3.1 Unknown Words

The previous section discussed the problem of words whose bigram probability is zero. But what about words we simply have never seen before?

closed vocabulary

Sometimes we have a language task in which this can't happen because we know all the words that can occur. In such a **closed vocabulary** system the test set can only contain words from this lexicon, and there will be no unknown words. This is a reasonable assumption in some domains, such as speech recognition or machine translation, where we have a pronunciation dictionary or a phrase table that are fixed in advance, and so the language model can only use the words in that dictionary or phrase table.

OOV
open vocabulary

In other cases we have to deal with words we haven't seen before, which we'll call **unknown** words, or **out of vocabulary** (**OOV**) words. The percentage of OOV words that appear in the test set is called the **OOV rate**. An **open vocabulary** system is one in which we model these potential unknown words in the test set by adding a pseudo-word called `<UNK>`.

There are two common ways to train the probabilities of the unknown word model `<UNK>`. The first one is to turn the problem back into a closed vocabulary one by choosing a fixed vocabulary in advance:

1. **Choose a vocabulary** (word list) that is fixed in advance.
2. **Convert** in the training set any word that is not in this set (any OOV word) to the unknown word token `<UNK>` in a text normalization step.
3. **Estimate** the probabilities for `<UNK>` from its counts just like any other regular word in the training set.

The second alternative, in situations where we don't have a prior vocabulary in advance, is to create such a vocabulary implicitly, replacing words in the training data by `<UNK>` based on their frequency. For example we can replace by `<UNK>` all words that occur fewer than n times in the training set, where n is some small number, or equivalently select a vocabulary size V in advance (say 50,000) and choose the top V words by frequency and replace the rest by UNK. In either case we then proceed to train the language model as before, treating `<UNK>` like a regular word.

The exact choice of `<UNK>` model does have an effect on metrics like perplexity. A language model can achieve low perplexity by choosing a small vocabulary and assigning the unknown word a high probability. For this reason, perplexities should only be compared across language models with the same vocabularies (Buck et al., 2014).

3.4 Smoothing

What do we do with words that are in our vocabulary (they are not unknown words) but appear in a test set in an unseen context (for example they appear after a word they never appeared after in training)? To keep a language model from assigning zero probability to these unseen events, we'll have to shave off a bit of probability mass from some more frequent events and give it to the events we've never seen. This modification is called **smoothing** or **discounting**. In this section and the following ones we'll introduce a variety of ways to do smoothing: **add-1 smoothing**, **add-k smoothing**, **stupid backoff**, and **Kneser-Ney smoothing**.

smoothing
discounting

3.4.1 Laplace Smoothing

The simplest way to do smoothing is to add one to all the bigram counts, before we normalize them into probabilities. All the counts that used to be zero will now have a count of 1, the counts of 1 will be 2, and so on. This algorithm is called **Laplace smoothing**. Laplace smoothing does not perform well enough to be used in modern n-gram models, but it usefully introduces many of the concepts that we see in other smoothing algorithms, gives a useful baseline, and is also a practical smoothing algorithm for other tasks like **text classification** (Chapter 4).

Laplace smoothing

Let's start with the application of Laplace smoothing to unigram probabilities. Recall that the unsmoothed maximum likelihood estimate of the unigram probability of the word w_i is its count c_i normalized by the total number of word tokens N:

$$P(w_i) = \frac{c_i}{N}$$

Laplace smoothing merely adds one to each count (hence its alternate name **add-one** smoothing). Since there are V words in the vocabulary and each one was incremented, we also need to adjust the denominator to take into account the extra V observations. (What happens to our P values if we don't increase the denominator?)

add-one

$$P_{\text{Laplace}}(w_i) = \frac{c_i+1}{N+V} \tag{3.20}$$

Instead of changing both the numerator and denominator, it is convenient to describe how a smoothing algorithm affects the numerator, by defining an **adjusted count** c^*. This adjusted count is easier to compare directly with the MLE counts and can be turned into a probability like an MLE count by normalizing by N. To define this count, since we are only changing the numerator in addition to adding 1 we'll also need to multiply by a normalization factor $\frac{N}{N+V}$:

$$c_i^* = (c_i+1)\frac{N}{N+V} \tag{3.21}$$

We can now turn c_i^* into a probability P_i^* by normalizing by N.

A related way to view smoothing is as **discounting** (lowering) some non-zero counts in order to get the probability mass that will be assigned to the zero counts. Thus, instead of referring to the discounted counts c^*, we might describe a smoothing algorithm in terms of a relative **discount** d_c, the ratio of the discounted counts to the original counts:

discounting
discount

$$d_c = \frac{c^*}{c}$$

Now that we have the intuition for the unigram case, let's smooth our Berkeley Restaurant Project bigrams. Figure 3.5 shows the add-one smoothed counts for the bigrams in Fig. 3.1.

	i	want	to	eat	chinese	food	lunch	spend
i	6	828	1	10	1	1	1	3
want	3	1	609	2	7	7	6	2
to	3	1	5	687	3	1	7	212
eat	1	1	3	1	17	3	43	1
chinese	2	1	1	1	1	83	2	1
food	16	1	16	1	2	5	1	1
lunch	3	1	1	1	1	2	1	1
spend	2	1	2	1	1	1	1	1

Figure 3.5 Add-one smoothed bigram counts for eight of the words (out of $V = 1446$) in the Berkeley Restaurant Project corpus of 9332 sentences. Previously-zero counts are in gray.

Figure 3.6 shows the add-one smoothed probabilities for the bigrams in Fig. 3.2. Recall that normal bigram probabilities are computed by normalizing each row of counts by the unigram count:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \tag{3.22}$$

For add-one smoothed bigram counts, we need to augment the unigram count by the number of total word types in the vocabulary V:

$$P^*_{\text{Laplace}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+1}{\sum_w (C(w_{n-1}w)+1)} = \frac{C(w_{n-1}w_n)+1}{C(w_{n-1})+V} \tag{3.23}$$

Thus, each of the unigram counts given in the previous section will need to be augmented by $V = 1446$. The result is the smoothed bigram probabilities in Fig. 3.6.

	i	want	to	eat	chinese	food	lunch	spend
i	0.0015	0.21	0.00025	0.0025	0.00025	0.00025	0.00025	0.00075
want	0.0013	0.00042	0.26	0.00084	0.0029	0.0029	0.0025	0.00084
to	0.00078	0.00026	0.0013	0.18	0.00078	0.00026	0.0018	0.055
eat	0.00046	0.00046	0.0014	0.00046	0.0078	0.0014	0.02	0.00046
chinese	0.0012	0.00062	0.00062	0.00062	0.00062	0.052	0.0012	0.00062
food	0.0063	0.00039	0.0063	0.00039	0.00079	0.002	0.00039	0.00039
lunch	0.0017	0.00056	0.00056	0.00056	0.00056	0.0011	0.00056	0.00056
spend	0.0012	0.00058	0.0012	0.00058	0.00058	0.00058	0.00058	0.00058

Figure 3.6 Add-one smoothed bigram probabilities for eight of the words (out of $V = 1446$) in the BeRP corpus of 9332 sentences. Previously-zero probabilities are in gray.

It is often convenient to reconstruct the count matrix so we can see how much a smoothing algorithm has changed the original counts. These adjusted counts can be computed by Eq. 3.24. Figure 3.7 shows the reconstructed counts.

$$c^*(w_{n-1}w_n) = \frac{[C(w_{n-1}w_n)+1] \times C(w_{n-1})}{C(w_{n-1})+V} \tag{3.24}$$

	i	want	to	eat	chinese	food	lunch	spend
i	3.8	527	0.64	6.4	0.64	0.64	0.64	1.9
want	1.2	0.39	238	0.78	2.7	2.7	2.3	0.78
to	1.9	0.63	3.1	430	1.9	0.63	4.4	133
eat	0.34	0.34	1	0.34	5.8	1	15	0.34
chinese	0.2	0.098	0.098	0.098	0.098	8.2	0.2	0.098
food	6.9	0.43	6.9	0.43	0.86	2.2	0.43	0.43
lunch	0.57	0.19	0.19	0.19	0.19	0.38	0.19	0.19
spend	0.32	0.16	0.32	0.16	0.16	0.16	0.16	0.16

Figure 3.7 Add-one reconstituted counts for eight words (of $V = 1446$) in the BeRP corpus of 9332 sentences. Previously-zero counts are in gray.

Note that add-one smoothing has made a very big change to the counts. $C(want\ to)$ changed from 608 to 238! We can see this in probability space as well: $P(to|want)$ decreases from .66 in the unsmoothed case to .26 in the smoothed case. Looking at the discount d (the ratio between new and old counts) shows us how strikingly the counts for each prefix word have been reduced; the discount for the bigram *want to* is .39, while the discount for *Chinese food* is .10, a factor of 10!

The sharp change in counts and probabilities occurs because too much probability mass is moved to all the zeros.

3.4.2 Add-k smoothing

One alternative to add-one smoothing is to move a bit less of the probability mass from the seen to the unseen events. Instead of adding 1 to each count, we add a fractional count k (.5? .05? .01?). This algorithm is therefore called **add-k smoothing**.

add-k

$$P^{*}_{\text{Add-k}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+k}{C(w_{n-1})+kV} \tag{3.25}$$

Add-k smoothing requires that we have a method for choosing k; this can be done, for example, by optimizing on a **devset**. Although add-k is useful for some tasks (including text classification), it turns out that it still doesn't work well for language modeling, generating counts with poor variances and often inappropriate discounts (Gale and Church, 1994).

3.4.3 Backoff and Interpolation

The discounting we have been discussing so far can help solve the problem of zero frequency n-grams. But there is an additional source of knowledge we can draw on. If we are trying to compute $P(w_n|w_{n-2}w_{n-1})$ but we have no examples of a particular trigram $w_{n-2}w_{n-1}w_n$, we can instead estimate its probability by using the bigram probability $P(w_n|w_{n-1})$. Similarly, if we don't have counts to compute $P(w_n|w_{n-1})$, we can look to the unigram $P(w_n)$.

In other words, sometimes using **less context** is a good thing, helping to generalize more for contexts that the model hasn't learned much about. There are two ways to use this n-gram "hierarchy". In **backoff**, we use the trigram if the evidence is sufficient, otherwise we use the bigram, otherwise the unigram. In other words, we only "back off" to a lower-order n-gram if we have zero evidence for a higher-order n-gram. By contrast, in **interpolation**, we always mix the probability estimates from all the n-gram estimators, weighing and combining the trigram, bigram, and unigram counts.

backoff

interpolation

In simple linear interpolation, we combine different order n-grams by linearly interpolating all the models. Thus, we estimate the trigram probability $P(w_n|w_{n-2}w_{n-1})$ by mixing together the unigram, bigram, and trigram probabilities, each weighted by a λ:

$$\begin{aligned}\hat{P}(w_n|w_{n-2}w_{n-1}) &= \lambda_1 P(w_n|w_{n-2}w_{n-1}) \\ &\quad +\lambda_2 P(w_n|w_{n-1}) \\ &\quad +\lambda_3 P(w_n)\end{aligned} \tag{3.26}$$

such that the λs sum to 1:

$$\sum_i \lambda_i = 1 \tag{3.27}$$

In a slightly more sophisticated version of linear interpolation, each λ weight is computed by conditioning on the context. This way, if we have particularly accurate counts for a particular bigram, we assume that the counts of the trigrams based on this bigram will be more trustworthy, so we can make the λs for those trigrams higher and thus give that trigram more weight in the interpolation. Equation 3.28 shows the equation for interpolation with context-conditioned weights:

$$\begin{aligned}\hat{P}(w_n|w_{n-2}w_{n-1}) &= \lambda_1(w_{n-2}^{n-1}) P(w_n|w_{n-2}w_{n-1}) \\ &\quad +\lambda_2(w_{n-2}^{n-1}) P(w_n|w_{n-1}) \\ &\quad +\lambda_3(w_{n-2}^{n-1}) P(w_n)\end{aligned} \tag{3.28}$$

How are these λ values set? Both the simple interpolation and conditional interpolation λs are learned from a **held-out** corpus. A held-out corpus is an additional training corpus that we use to set hyperparameters like these λ values, by choosing the λ values that maximize the likelihood of the held-out corpus. That is, we fix the n-gram probabilities and then search for the λ values that—when plugged into Eq. 3.26—give us the highest probability of the held-out set. There are various ways to find this optimal set of λs. One way is to use the **EM** algorithm, an iterative learning algorithm that converges on locally optimal λs (Jelinek and Mercer, 1980).

held-out

In a **backoff** n-gram model, if the n-gram we need has zero counts, we approximate it by backing off to the (N-1)-gram. We continue backing off until we reach a history that has some counts.

In order for a backoff model to give a correct probability distribution, we have to **discount** the higher-order n-grams to save some probability mass for the lower order n-grams. Just as with add-one smoothing, if the higher-order n-grams aren't discounted and we just used the undiscounted MLE probability, then as soon as we replaced an n-gram which has zero probability with a lower-order n-gram, we would be adding probability mass, and the total probability assigned to all possible strings by the language model would be greater than 1! In addition to this explicit discount factor, we'll need a function α to distribute this probability mass to the lower order n-grams.

discount

This kind of backoff with discounting is also called **Katz backoff**. In Katz backoff we rely on a discounted probability P^* if we've seen this n-gram before (i.e., if we have non-zero counts). Otherwise, we recursively back off to the Katz probability for the shorter-history (N-1)-gram. The probability for a backoff n-gram P_{BO} is

Katz backoff

thus computed as follows:

$$P_{\mathrm{BO}}(w_n|w_{n-N+1}^{n-1}) = \begin{cases} P^*(w_n|w_{n-N+1}^{n-1}), & \text{if } C(w_{n-N+1}^{n}) > 0 \\ \alpha(w_{n-N+1}^{n-1})P_{\mathrm{BO}}(w_n|w_{n-N+2}^{n-1}), & \text{otherwise.} \end{cases} \tag{3.29}$$

Good-Turing Katz backoff is often combined with a smoothing method called **Good-Turing**. The combined **Good-Turing backoff** algorithm involves quite detailed computation for estimating the Good-Turing smoothing and the P^* and α values.

3.5 Kneser-Ney Smoothing

Kneser-Ney One of the most commonly used and best performing n-gram smoothing methods is the interpolated **Kneser-Ney** algorithm (Kneser and Ney 1995, Chen and Goodman 1998).

Kneser-Ney has its roots in a method called **absolute discounting**. Recall that **discounting** of the counts for frequent n-grams is necessary to save some probability mass for the smoothing algorithm to distribute to the unseen n-grams.

To see this, we can use a clever idea from Church and Gale (1991). Consider an n-gram that has count 4. We need to discount this count by some amount. But how much should we discount it? Church and Gale's clever idea was to look at a held-out corpus and just see what the count is for all those bigrams that had count 4 in the training set. They computed a bigram grammar from 22 million words of AP newswire and then checked the counts of each of these bigrams in another 22 million words. On average, a bigram that occurred 4 times in the first 22 million words occurred 3.23 times in the next 22 million words. The following table from Church and Gale (1991) shows these counts for bigrams with c from 0 to 9:

Bigram count in training set	Bigram count in heldout set
0	0.0000270
1	0.448
2	1.25
3	2.24
4	3.23
5	4.21
6	5.23
7	6.21
8	7.21
9	8.26

Figure 3.8 For all bigrams in 22 million words of AP newswire of count 0, 1, 2,...,9, the counts of these bigrams in a held-out corpus also of 22 million words.

Absolute discounting The astute reader may have noticed that except for the held-out counts for 0 and 1, all the other bigram counts in the held-out set could be estimated pretty well by just subtracting 0.75 from the count in the training set! **Absolute discounting** formalizes this intuition by subtracting a fixed (absolute) discount d from each count. The intuition is that since we have good estimates already for the very high counts, a small discount d won't affect them much. It will mainly modify the smaller counts,

for which we don't necessarily trust the estimate anyway, and Fig. 3.8 suggests that in practice this discount is actually a good one for bigrams with counts 2 through 9. The equation for interpolated absolute discounting applied to bigrams:

$$P_{\text{AbsoluteDiscounting}}(w_i|w_{i-1}) = \frac{C(w_{i-1}w_i) - d}{\sum_v C(w_{i-1}v)} + \lambda(w_{i-1})P(w_i) \tag{3.30}$$

The first term is the discounted bigram, and the second term is the unigram with an interpolation weight λ. We could just set all the d values to .75, or we could keep a separate discount value of 0.5 for the bigrams with counts of 1.

Kneser-Ney discounting (Kneser and Ney, 1995) augments absolute discounting with a more sophisticated way to handle the lower-order unigram distribution. Consider the job of predicting the next word in this sentence, assuming we are interpolating a bigram and a unigram model.

I can't see without my reading ________________ .

The word *glasses* seems much more likely to follow here than, say, the word *Kong*, so we'd like our unigram model to prefer *glasses*. But in fact it's *Kong* that is more common, since *Hong Kong* is a very frequent word. A standard unigram model will assign *Kong* a higher probability than *glasses*. We would like to capture the intuition that although *Kong* is frequent, it is mainly only frequent in the phrase *Hong Kong*, that is, after the word *Hong*. The word *glasses* has a much wider distribution.

In other words, instead of $P(w)$, which answers the question "How likely is w?", we'd like to create a unigram model that we might call $P_{\text{CONTINUATION}}$, which answers the question "How likely is w to appear as a novel continuation?". How can we estimate this probability of seeing the word w as a novel continuation, in a new unseen context? The Kneser-Ney intuition is to base our estimate of $P_{\text{CONTINUATION}}$ on the *number of different contexts word w has appeared in*, that is, the number of bigram types it completes. Every bigram type was a novel continuation the first time it was seen. We hypothesize that words that have appeared in more contexts in the past are more likely to appear in some new context as well. The number of times a word w appears as a novel continuation can be expressed as:

$$P_{\text{CONTINUATION}}(w) \propto |\{v : C(vw) > 0\}| \tag{3.31}$$

To turn this count into a probability, we normalize by the total number of word bigram types. In summary:

$$P_{\text{CONTINUATION}}(w) = \frac{|\{v : C(vw) > 0\}|}{|\{(u',w') : C(u'w') > 0\}|} \tag{3.32}$$

An alternative metaphor for an equivalent formulation is to use the number of word types seen to precede w (Eq. 3.31 repeated):

$$P_{\text{CONTINUATION}}(w) \propto |\{v : C(vw) > 0\}| \tag{3.33}$$

normalized by the number of words preceding all words, as follows:

$$P_{\text{CONTINUATION}}(w) = \frac{|\{v : C(vw) > 0\}|}{\sum_{w'} |\{v : C(vw') > 0\}|} \tag{3.34}$$

A frequent word (Kong) occurring in only one context (Hong) will have a low continuation probability.

Interpolated Kneser-Ney

The final equation for **Interpolated Kneser-Ney** smoothing for bigrams is then:

$$P_{\text{KN}}(w_i|w_{i-1}) = \frac{\max(C(w_{i-1}w_i) - d, 0)}{C(w_{i-1})} + \lambda(w_{i-1})P_{\text{CONTINUATION}}(w_i) \quad (3.35)$$

The λ is a normalizing constant that is used to distribute the probability mass we've discounted.:

$$\lambda(w_{i-1}) = \frac{d}{\sum_v C(w_{i-1}v)} |\{w : C(w_{i-1}w) > 0\}| \quad (3.36)$$

The first term $\frac{d}{\sum_v C(w_{i-1}v)}$ is the normalized discount. The second term $|\{w : C(w_{i-1}w) > 0\}|$ is the number of word types that can follow w_{i-1} or, equivalently, the number of word types that we discounted; in other words, the number of times we applied the normalized discount.

The general recursive formulation is as follows:

$$P_{\text{KN}}(w_i|w_{i-n+1}^{i-1}) = \frac{\max(c_{KN}(w_{i-n+1}^{i}) - d, 0)}{\sum_v c_{KN}(w_{i-n+1}^{i-1}v)} + \lambda(w_{i-n+1}^{i-1})P_{KN}(w_i|w_{i-n+2}^{i-1}) \quad (3.37)$$

where the definition of the count c_{KN} depends on whether we are counting the highest-order n-gram being interpolated (for example trigram if we are interpolating trigram, bigram, and unigram) or one of the lower-order n-grams (bigram or unigram if we are interpolating trigram, bigram, and unigram):

$$c_{KN}(\cdot) = \begin{cases} \text{count}(\cdot) & \text{for the highest order} \\ \text{continuationcount}(\cdot) & \text{for lower orders} \end{cases} \quad (3.38)$$

The continuation count is the number of unique single word contexts for $\cdot$.

At the termination of the recursion, unigrams are interpolated with the uniform distribution, where the parameter ϵ is the empty string:

$$P_{\text{KN}}(w) = \frac{\max(c_{KN}(w) - d, 0)}{\sum_{w'} c_{KN}(w')} + \lambda(\epsilon)\frac{1}{V} \quad (3.39)$$

If we want to include an unknown word <UNK>, it's just included as a regular vocabulary entry with count zero, and hence its probability will be a lambda-weighted uniform distribution $\frac{\lambda(\epsilon)}{V}$.

modified Kneser-Ney

The best-performing version of Kneser-Ney smoothing is called **modified Kneser-Ney** smoothing, and is due to Chen and Goodman (1998). Rather than use a single fixed discount d, modified Kneser-Ney uses three different discounts d_1, d_2, and d_{3+} for n-grams with counts of 1, 2 and three or more, respectively. See Chen and Goodman (1998, p. 19) or Heafield et al. (2013) for the details.

3.6 The Web and Stupid Backoff

By using text from the web, it is possible to build extremely large language models. In 2006 Google released a very large set of N-gram counts, including n-grams (1-grams through 5-grams) from all the five-word sequences that appear at least 40 times from 1,024,908,267,229 words of running text on the web; this includes

1,176,470,663 five-word sequences using over 13 million unique words types (Franz and Brants, 2006). Some examples:

4-gram	Count
serve as the incoming	92
serve as the incubator	99
serve as the independent	794
serve as the index	223
serve as the indication	72
serve as the indicator	120
serve as the indicators	45
serve as the indispensable	111
serve as the indispensible	40
serve as the individual	234

Efficiency considerations are important when building language models that use such large sets of n-grams. Rather than store each word as a string, it is generally represented in memory as a 64-bit hash number, with the words themselves stored on disk. Probabilities are generally quantized using only 4-8 bits (instead of 8-byte floats), and n-grams are stored in reverse tries.

N-grams can also be shrunk by pruning, for example only storing n-grams with counts greater than some threshold (such as the count threshold of 40 used for the Google n-gram release) or using entropy to prune less-important n-grams (Stolcke, 1998). Another option is to build approximate language models using techniques

Bloom filters

like **Bloom filters** (Talbot and Osborne 2007, Church et al. 2007). Finally, efficient language model toolkits like KenLM (Heafield 2011, Heafield et al. 2013) use sorted arrays, efficiently combine probabilities and backoffs in a single value, and use merge sorts to efficiently build the probability tables in a minimal number of passes through a large corpus.

Although with these toolkits it is possible to build web-scale language models using full Kneser-Ney smoothing, Brants et al. (2007) show that with very large language models a much simpler algorithm may be sufficient. The algorithm is called

stupid backoff

stupid backoff. Stupid backoff gives up the idea of trying to make the language model a true probability distribution. There is no discounting of the higher-order probabilities. If a higher-order n-gram has a zero count, we simply backoff to a lower order n-gram, weighed by a fixed (context-independent) weight. This algorithm does not produce a probability distribution, so we'll follow Brants et al. (2007) in referring to it as S:

$$S(w_i|w_{i-k+1}^{i-1}) = \begin{cases} \frac{\text{count}(w_{i-k+1}^{i})}{\text{count}(w_{i-k+1}^{i-1})} & \text{if count}(w_{i-k+1}^{i}) > 0 \\ \lambda S(w_i|w_{i-k+2}^{i-1}) & \text{otherwise} \end{cases} \tag{3.40}$$

The backoff terminates in the unigram, which has probability $S(w) = \frac{count(w)}{N}$. Brants et al. (2007) find that a value of 0.4 worked well for λ.

3.7 Advanced: Perplexity's Relation to Entropy

We introduced perplexity in Section 3.2.1 as a way to evaluate n-gram models on a test set. A better n-gram model is one that assigns a higher probability to the

test data, and perplexity is a normalized version of the probability of the test set. The perplexity measure actually arises from the information-theoretic concept of cross-entropy, which explains otherwise mysterious properties of perplexity (why the inverse probability, for example?) and its relationship to entropy. **Entropy** is a measure of information. Given a random variable X ranging over whatever we are predicting (words, letters, parts of speech, the set of which we'll call χ) and with a particular probability function, call it $p(x)$, the entropy of the random variable X is:

Entropy

$$H(X) = -\sum_{x\in\chi} p(x)\log_2 p(x) \tag{3.41}$$

The log can, in principle, be computed in any base. If we use log base 2, the resulting value of entropy will be measured in **bits**.

One intuitive way to think about entropy is as a lower bound on the number of bits it would take to encode a certain decision or piece of information in the optimal coding scheme.

Consider an example from the standard information theory textbook Cover and Thomas (1991). Imagine that we want to place a bet on a horse race but it is too far to go all the way to Yonkers Racetrack, so we'd like to send a short message to the bookie to tell him which of the eight horses to bet on. One way to encode this message is just to use the binary representation of the horse's number as the code; thus, horse 1 would be `001`, horse 2 `010`, horse 3 `011`, and so on, with horse 8 coded as `000`. If we spend the whole day betting and each horse is coded with 3 bits, on average we would be sending 3 bits per race.

Can we do better? Suppose that the spread is the actual distribution of the bets placed and that we represent it as the prior probability of each horse as follows:

Horse 1	$\frac{1}{2}$	Horse 5	$\frac{1}{64}$
Horse 2	$\frac{1}{4}$	Horse 6	$\frac{1}{64}$
Horse 3	$\frac{1}{8}$	Horse 7	$\frac{1}{64}$
Horse 4	$\frac{1}{16}$	Horse 8	$\frac{1}{64}$

The entropy of the random variable X that ranges over horses gives us a lower bound on the number of bits and is

$$\begin{aligned} H(X) &= -\sum_{i=1}^{i=8} p(i)\log p(i) \\ &= -\tfrac{1}{2}\log\tfrac{1}{2} - \tfrac{1}{4}\log\tfrac{1}{4} - \tfrac{1}{8}\log\tfrac{1}{8} - \tfrac{1}{16}\log\tfrac{1}{16} - 4(\tfrac{1}{64}\log\tfrac{1}{64}) \\ &= 2 \text{ bits} \end{aligned} \tag{3.42}$$

A code that averages 2 bits per race can be built with short encodings for more probable horses, and longer encodings for less probable horses. For example, we could encode the most likely horse with the code `0`, and the remaining horses as `10`, then `110`, `1110`, `111100`, `111101`, `111110`, and `111111`.

What if the horses are equally likely? We saw above that if we used an equal-length binary code for the horse numbers, each horse took 3 bits to code, so the average was 3. Is the entropy the same? In this case each horse would have a probability of $\frac{1}{8}$. The entropy of the choice of horses is then

$$H(X) = -\sum_{i=1}^{i=8} \frac{1}{8}\log\frac{1}{8} = -\log\frac{1}{8} = 3 \text{ bits} \tag{3.43}$$

Until now we have been computing the entropy of a single variable. But most of what we will use entropy for involves *sequences*. For a grammar, for example, we will be computing the entropy of some sequence of words $W = \{w_0, w_1, w_2, \ldots, w_n\}$. One way to do this is to have a variable that ranges over sequences of words. For example we can compute the entropy of a random variable that ranges over all finite sequences of words of length n in some language L as follows:

$$H(w_1, w_2, \ldots, w_n) = -\sum_{W_1^n \in L} p(W_1^n) \log p(W_1^n) \tag{3.44}$$

entropy rate

We could define the **entropy rate** (we could also think of this as the **per-word entropy**) as the entropy of this sequence divided by the number of words:

$$\frac{1}{n} H(W_1^n) = -\frac{1}{n} \sum_{W_1^n \in L} p(W_1^n) \log p(W_1^n) \tag{3.45}$$

But to measure the true entropy of a language, we need to consider sequences of infinite length. If we think of a language as a stochastic process L that produces a sequence of words, and allow W to represent the sequence of words $w_1, \ldots, w_n$, then L's entropy rate $H(L)$ is defined as

$$\begin{aligned} H(L) &= \lim_{n\to\infty} \frac{1}{n} H(w_1, w_2, \ldots, w_n) \\ &= -\lim_{n\to\infty} \frac{1}{n} \sum_{W \in L} p(w_1, \ldots, w_n) \log p(w_1, \ldots, w_n) \end{aligned} \tag{3.46}$$

The Shannon-McMillan-Breiman theorem (Algoet and Cover 1988, Cover and Thomas 1991) states that if the language is regular in certain ways (to be exact, if it is both stationary and ergodic),

$$H(L) = \lim_{n\to\infty} -\frac{1}{n} \log p(w_1 w_2 \ldots w_n) \tag{3.47}$$

That is, we can take a single sequence that is long enough instead of summing over all possible sequences. The intuition of the Shannon-McMillan-Breiman theorem is that a long-enough sequence of words will contain in it many other shorter sequences and that each of these shorter sequences will reoccur in the longer sequence according to their probabilities.

Stationary

A stochastic process is said to be **stationary** if the probabilities it assigns to a sequence are invariant with respect to shifts in the time index. In other words, the probability distribution for words at time t is the same as the probability distribution at time $t+1$. Markov models, and hence n-grams, are stationary. For example, in a bigram, P_i is dependent only on P_{i-1}. So if we shift our time index by x, P_{i+x} is still dependent on P_{i+x-1}. But natural language is not stationary, since as we show in Chapter 10, the probability of upcoming words can be dependent on events that were arbitrarily distant and time dependent. Thus, our statistical models only give an approximation to the correct distributions and entropies of natural language.

To summarize, by making some incorrect but convenient simplifying assumptions, we can compute the entropy of some stochastic process by taking a very long sample of the output and computing its average log probability.

cross-entropy

Now we are ready to introduce **cross-entropy**. The cross-entropy is useful when we don't know the actual probability distribution p that generated some data. It

allows us to use some m, which is a model of p (i.e., an approximation to p). The cross-entropy of m on p is defined by

$$H(p,m) = \lim_{n\to\infty} -\frac{1}{n}\sum_{W\in L} p(w_1,\ldots,w_n)\log m(w_1,\ldots,w_n) \tag{3.48}$$

That is, we draw sequences according to the probability distribution p, but sum the log of their probabilities according to m.

Again, following the Shannon-McMillan-Breiman theorem, for a stationary ergodic process:

$$H(p,m) = \lim_{n\to\infty} -\frac{1}{n}\log m(w_1 w_2 \ldots w_n) \tag{3.49}$$

This means that, as for entropy, we can estimate the cross-entropy of a model m on some distribution p by taking a single sequence that is long enough instead of summing over all possible sequences.

What makes the cross-entropy useful is that the cross-entropy $H(p,m)$ is an upper bound on the entropy $H(p)$. For any model m:

$$H(p) \leq H(p,m) \tag{3.50}$$

This means that we can use some simplified model m to help estimate the true entropy of a sequence of symbols drawn according to probability p. The more accurate m is, the closer the cross-entropy $H(p,m)$ will be to the true entropy $H(p)$. Thus, the difference between $H(p,m)$ and $H(p)$ is a measure of how accurate a model is. Between two models m_1 and m_2, the more accurate model will be the one with the lower cross-entropy. (The cross-entropy can never be lower than the true entropy, so a model cannot err by underestimating the true entropy.)

We are finally ready to see the relation between perplexity and cross-entropy as we saw it in Eq. 3.49. Cross-entropy is defined in the limit, as the length of the observed word sequence goes to infinity. We will need an approximation to cross-entropy, relying on a (sufficiently long) sequence of fixed length. This approximation to the cross-entropy of a model $M = P(w_i|w_{i-N+1}...w_{i-1})$ on a sequence of words W is

$$H(W) = -\frac{1}{N}\log P(w_1 w_2 \ldots w_N) \tag{3.51}$$

perplexity

The **perplexity** of a model P on a sequence of words W is now formally defined as the exp of this cross-entropy:

$$\begin{aligned}
\text{Perplexity}(W) &= 2^{H(W)} \\
&= P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} \\
&= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}} \\
&= \sqrt[N]{\prod_{i=1}^{N}\frac{1}{P(w_i|w_1\ldots w_{i-1})}}
\end{aligned} \tag{3.52}$$

3.8 Summary

This chapter introduced language modeling and the n-gram, one of the most widely used tools in language processing.

- Language models offer a way to assign a probability to a sentence or other sequence of words, and to predict a word from preceding words.
- n-grams are Markov models that estimate words from a fixed window of previous words. n-gram probabilities can be estimated by counting in a corpus and normalizing (the **maximum likelihood estimate**).
- n-gram **language models** are evaluated extrinsically in some task, or intrinsically using **perplexity.**
- The **perplexity** of a test set according to a language model is the geometric mean of the inverse test set probability computed by the model.
- **Smoothing** algorithms provide a more sophisticated way to estimate the probability of n-grams. Commonly used smoothing algorithms for n-grams rely on lower-order n-gram counts through **backoff** or **interpolation**.
- Both backoff and interpolation require **discounting** to create a probability distribution.
- **Kneser-Ney** smoothing makes use of the probability of a word being a novel **continuation**. The interpolated **Kneser-Ney** smoothing algorithm mixes a discounted probability with a lower-order continuation probability.

Bibliographical and Historical Notes

The underlying mathematics of the n-gram was first proposed by Markov (1913), who used what are now called **Markov chains** (bigrams and trigrams) to predict whether an upcoming letter in Pushkin's *Eugene Onegin* would be a vowel or a consonant. Markov classified 20,000 letters as V or C and computed the bigram and trigram probability that a given letter would be a vowel given the previous one or two letters. Shannon (1948) applied n-grams to compute approximations to English word sequences. Based on Shannon's work, Markov models were commonly used in engineering, linguistic, and psychological work on modeling word sequences by the 1950s. In a series of extremely influential papers starting with Chomsky (1956) and including Chomsky (1957) and Miller and Chomsky (1963), Noam Chomsky argued that "finite-state Markov processes", while a possibly useful engineering heuristic, were incapable of being a complete cognitive model of human grammatical knowledge. These arguments led many linguists and computational linguists to ignore work in statistical modeling for decades.

The resurgence of n-gram models came from Jelinek and colleagues at the IBM Thomas J. Watson Research Center, who were influenced by Shannon, and Baker at CMU, who was influenced by the work of Baum and colleagues. Independently these two labs successfully used n-grams in their speech recognition systems (Baker 1990, Jelinek 1976, Baker 1975, Bahl et al. 1983, Jelinek 1990). A trigram model was used in the IBM TANGORA speech recognition system in the 1970s, but the idea was not written up until later.

Add-one smoothing derives from Laplace's 1812 law of succession and was first applied as an engineering solution to the zero-frequency problem by Jeffreys (1948)

based on an earlier Add-K suggestion by Johnson (1932). Problems with the add-one algorithm are summarized in Gale and Church (1994).

A wide variety of different language modeling and smoothing techniques were proposed in the 80s and 90s, including Good-Turing discounting—first applied to the n-gram smoothing at IBM by Katz (Nádas 1984, Church and Gale 1991)—Witten-Bell discounting (Witten and Bell, 1991), and varieties of **class-based n-gram** models that used information about word classes.

class-based n-gram

Starting in the late 1990s, Chen and Goodman produced a highly influential series of papers with a comparison of different language models (Chen and Goodman 1996, Chen and Goodman 1998, Chen and Goodman 1999, Goodman 2006). They performed a number of carefully controlled experiments comparing different discounting algorithms, cache models, class-based models, and other language model parameters. They showed the advantages of **Modified Interpolated Kneser-Ney**, which has since become the standard baseline for language modeling, especially because they showed that caches and class-based models provided only minor additional improvement. These papers are recommended for any reader with further interest in language modeling.

Two commonly used toolkits for building language models are SRILM (Stolcke, 2002) and KenLM (Heafield 2011, Heafield et al. 2013). Both are publicly available. SRILM offers a wider range of options and types of discounting, while KenLM is optimized for speed and memory size, making it possible to build web-scale language models.

neural nets

The highest accuracy language models at the time of this writing make use of **neural nets**. The problem with standard language models is that the number of parameters increases exponentially as the n-gram order increases, and n-grams have no way to generalize from training to test set. Neural networks instead project words into a **continuous** space in which words with similar contexts have similar representations. Both **feedforward** nets Bengio et al. 2006, Schwenk 2007 and **recurrent nets** (Mikolov, 2012) are used.

maximum entropy

Other important classes of language models are **maximum entropy language models** (Rosenfeld, 1996), based on logistic regression classifiers that use lots of features to help predict upcoming words. These classifiers can use the standard features presented in this chapter (i.e., the previous words) but also lots of other useful predictors, as can other kinds of discriminative language models (Roark et al., 2007). We'll introduce logistic regression language modeling when we introduce classification in Chapter 4.

adaptation

Another important technique is language model **adaptation**, where we want to combine data from multiple domains (for example we might have less in-domain training data but more general data that we then need to adapt) (Bulyko et al. 2003, Bacchiani et al. 2004, Bellegarda 2004, Bacchiani et al. 2006, Hsu 2007, Liu et al. 2013).

Exercises

3.1 Write out the equation for trigram probability estimation (modifying Eq. 3.11). Now write out all the non-zero trigram probabilities for the `I am Sam` corpus on page 39.

3.2 Calculate the probability of the sentence `i want chinese food`. Give two probabilities, one using Fig. 3.2, and another using the add-1 smoothed table in Fig. 3.6.

3.3 Which of the two probabilities you computed in the previous exercise is higher, unsmoothed or smoothed? Explain why.

3.4 We are given the following corpus, modified from the one in the chapter:

```
<s> I am Sam </s>
<s> Sam I am </s>
<s> I am Sam </s>
<s> I do not like green eggs and Sam </s>
```

Using a bigram language model with add-one smoothing, what is P(Sam | am)? Include <s> and </s> in your counts just like any other token.

3.5 Suppose we didn't use the end-symbol </s>. Train an unsmoothed bigram grammar on the following training corpus without using the end-symbol </s>:

```
<s> a b
<s> b b
<s> b a
<s> a a
```

Demonstrate that your bigram model does not assign a single probability distribution across all sentence lengths by showing that the sum of the probability of the four possible 2 word sentences over the alphabet {a,b} is 1.0, and the sum of the probability of all possible 3 word sentences over the alphabet {a,b} is also 1.0.

3.6 Suppose we train a trigram language model with add-one smoothing on a given corpus. The corpus contains V word types. Express a formula for estimating P(w3|w1,w2), where w3 is a word which follows the bigram (w1,w2), in terms of various N-gram counts and V. Use the notation c(w1,w2,w3) to denote the number of times that trigram (w1,w2,w3) occurs in the corpus, and so on for bigrams and unigrams.

3.7 We are given the following corpus, modified from the one in the chapter:

```
<s> I am Sam </s>
<s> Sam I am </s>
<s> I am Sam </s>
<s> I do not like green eggs and Sam </s>
```

If we use linear interpolation smoothing between a maximum-likelihood bigram model and a maximum-likelihood unigram model with $\lambda_1 = \frac{1}{2}$ and $\lambda_2 = \frac{1}{2}$, what is P(Sam|am)? Include <s> and </s>\verb in your counts just like any other token.

3.8 Write a program to compute unsmoothed unigrams and bigrams.

3.9 Run your *N*-gram program on two different small corpora of your choice (you might use email text or newsgroups). Now compare the statistics of the two corpora. What are the differences in the most common unigrams between the two? How about interesting differences in bigrams?

3.10 Add an option to your program to generate random sentences.

3.11 Add an option to your program to compute the perplexity of a test set.

CHAPTER

4 Naive Bayes and Sentiment Classification

Classification lies at the heart of both human and machine intelligence. Deciding what letter, word, or image has been presented to our senses, recognizing faces or voices, sorting mail, assigning grades to homeworks; these are all examples of assigning a category to an input. The potential challenges of this task are highlighted by the fabulist Jorge Luis Borges (1964), who imagined classifying animals into:

> *(a) those that belong to the Emperor, (b) embalmed ones, (c) those that are trained, (d) suckling pigs, (e) mermaids, (f) fabulous ones, (g) stray dogs, (h) those that are included in this classification, (i) those that tremble as if they were mad, (j) innumerable ones, (k) those drawn with a very fine camel's hair brush, (l) others, (m) those that have just broken a flower vase, (n) those that resemble flies from a distance.*

Many language processing tasks involve classification, although luckily our classes are much easier to define than those of Borges. In this chapter we introduce the naive Bayes algorithm and apply it to **text categorization**, the task of assigning a label or category to an entire text or document.

text categorization

sentiment analysis

We focus on one common text categorization task, **sentiment analysis**, the extraction of **sentiment**, the positive or negative orientation that a writer expresses toward some object. A review of a movie, book, or product on the web expresses the author's sentiment toward the product, while an editorial or political text expresses sentiment toward a candidate or political action. Extracting consumer or public sentiment is thus relevant for fields from marketing to politics.

The simplest version of sentiment analysis is a binary classification task, and the words of the review provide excellent cues. Consider, for example, the following phrases extracted from positive and negative reviews of movies and restaurants,. Words like *great*, *richly*, *awesome*, and *pathetic*, and *awful* and *ridiculously* are very informative cues:

+ *...zany characters and richly applied satire, and some great plot twists*

− *It was pathetic. The worst part about it was the boxing scenes...*

+ *...awesome caramel sauce and sweet toasty almonds. I love this place!*

− *...awful pizza and ridiculously overpriced...*

spam detection

Spam detection is another important commercial application, the binary classification task of assigning an email to one of the two classes *spam* or *not-spam*. Many lexical and other features can be used to perform this classification. For example you might quite reasonably be suspicious of an email containing phrases like "online pharmaceutical" or "WITHOUT ANY COST" or "Dear Winner".

language id

authorship attribution

Another thing we might want to know about a text is the language it's written in. Texts on social media, for example, can be in any number of languages and we'll need to apply different processing. The task of **language id** is thus the first step in most language processing pipelines. Related tasks like determining a text's author, (**authorship attribution**), or author characteristics like gender, age, and native

language are text classification tasks that are also relevant to the digital humanities, social sciences, and forensic linguistics.

Finally, one of the oldest tasks in text classification is assigning a library subject category or topic label to a text. Deciding whether a research paper concerns epidemiology or instead, perhaps, embryology, is an important component of information retrieval. Various sets of subject categories exist, such as the MeSH (Medical Subject Headings) thesaurus. In fact, as we will see, subject category classification is the task for which the naive Bayes algorithm was invented in 1961.

Classification is essential for tasks below the level of the document as well. We've already seen period disambiguation (deciding if a period is the end of a sentence or part of a word), and word tokenization (deciding if a character should be a word boundary). Even language modeling can be viewed as classification: each word can be thought of as a class, and so predicting the next word is classifying the context-so-far into a class for each next word. A part-of-speech tagger (Chapter 8) classifies each occurrence of a word in a sentence as, e.g., a noun or a verb.

The goal of classification is to take a single observation, extract some useful features, and thereby **classify** the observation into one of a set of discrete classes. One method for classifying text is to use hand-written rules. There are many areas of language processing where hand-written rule-based classifiers constitute a state-of-the-art system, or at least part of it.

Rules can be fragile, however, as situations or data change over time, and for some tasks humans aren't necessarily good at coming up with the rules. Most cases of classification in language processing are instead done via **supervised machine learning**, and this will be the subject of the remainder of this chapter. In supervised learning, we have a data set of input observations, each associated with some correct output (a 'supervision signal'). The goal of the algorithm is to learn how to map from a new observation to a correct output.

supervised machine learning

Formally, the task of supervised classification is to take an input x and a fixed set of output classes $Y = y_1, y_2, ..., y_M$ and return a predicted class $y \in Y$. For text classification, we'll sometimes talk about c (for "class") instead of y as our output variable, and d (for "document") instead of x as our input variable. In the supervised situation we have a training set of N documents that have each been hand-labeled with a class: $(d_1, c_1), ..., (d_N, c_N)$. Our goal is to learn a classifier that is capable of mapping from a new document d to its correct class $c \in C$. A **probabilistic classifier** additionally will tell us the probability of the observation being in the class. This full distribution over the classes can be useful information for downstream decisions; avoiding making discrete decisions early on can be useful when combining systems.

Many kinds of machine learning algorithms are used to build classifiers. This chapter introduces naive Bayes; the following one introduces logistic regression. These exemplify two ways of doing classification. **Generative** classifiers like naive Bayes build a model of how a class could generate some input data. Given an observation, they return the class most likely to have generated the observation. **Discriminative** classifiers like logistic regression instead learn what features from the input are most useful to discriminate between the different possible classes. While discriminative systems are often more accurate and hence more commonly used, generative classifiers still have a role.

4.1 Naive Bayes Classifiers

naive Bayes classifier

In this section we introduce the **multinomial naive Bayes classifier**, so called because it is a Bayesian classifier that makes a simplifying (naive) assumption about how the features interact.

bag-of-words

The intuition of the classifier is shown in Fig. 4.1. We represent a text document as if it were a **bag-of-words**, that is, an unordered set of words with their position ignored, keeping only their frequency in the document. In the example in the figure, instead of representing the word order in all the phrases like "I love this movie" and "I would recommend it", we simply note that the word *I* occurred 5 times in the entire excerpt, the word *it* 6 times, the words *love*, *recommend*, and *movie* once, and so on.

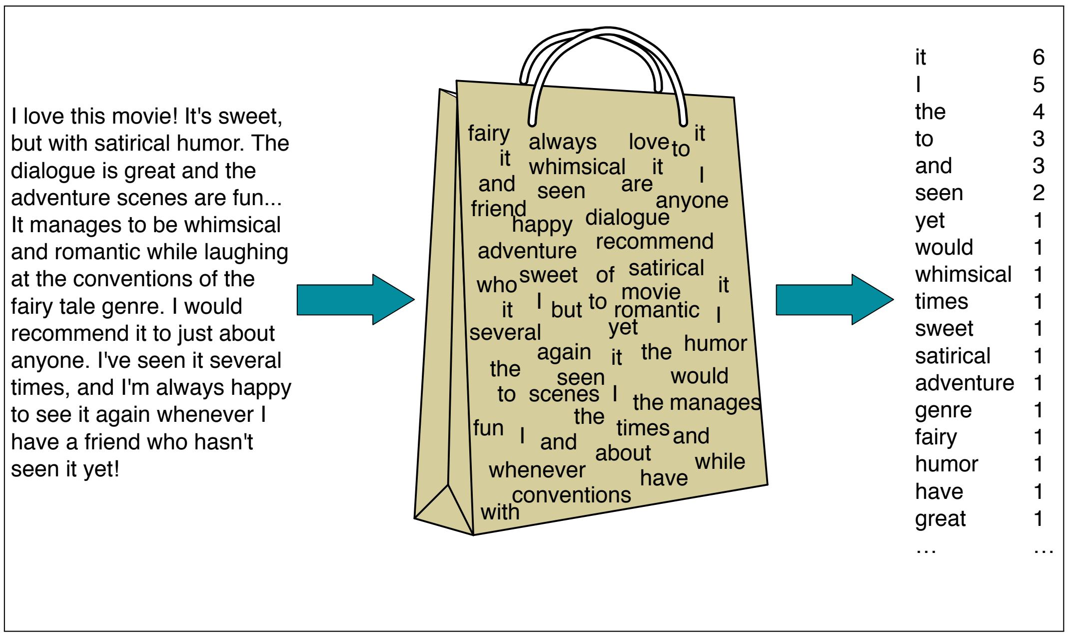


Figure 4.1 Intuition of the multinomial naive Bayes classifier applied to a movie review. The position of the words is ignored (the *bag of words* assumption) and we make use of the frequency of each word.

ˆ

Naive Bayes is a probabilistic classifier, meaning that for a document d, out of all classes $c \in C$ the classifier returns the class $\hat{c}$ which has the maximum posterior probability given the document. In Eq. 4.1 we use the hat notation ˆ to mean "our estimate of the correct class".

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} P(c|d) \tag{4.1}$$

Bayesian inference

This idea of **Bayesian inference** has been known since the work of Bayes (1763), and was first applied to text classification by Mosteller and Wallace (1964). The intuition of Bayesian classification is to use Bayes' rule to transform Eq. 4.1 into other probabilities that have some useful properties. Bayes' rule is presented in Eq. 4.2; it gives us a way to break down any conditional probability $P(x|y)$ into three other

probabilities:

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} \tag{4.2}$$

We can then substitute Eq. 4.2 into Eq. 4.1 to get Eq. 4.3:

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}}\, P(c|d) = \underset{c \in C}{\operatorname{argmax}}\, \frac{P(d|c)P(c)}{P(d)} \tag{4.3}$$

We can conveniently simplify Eq. 4.3 by dropping the denominator $P(d)$. This is possible because we will be computing $\frac{P(d|c)P(c)}{P(d)}$ for each possible class. But $P(d)$ doesn't change for each class; we are always asking about the most likely class for the same document d, which must have the same probability $P(d)$. Thus, we can choose the class that maximizes this simpler formula:

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}}\, P(c|d) = \underset{c \in C}{\operatorname{argmax}}\, P(d|c)P(c) \tag{4.4}$$

prior probability
likelihood

We thus compute the most probable class $\hat{c}$ given some document d by choosing the class which has the highest product of two probabilities: the **prior probability** of the class $P(c)$ and the **likelihood** of the document $P(d|c)$:

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}}\ \overbrace{P(d|c)}^{\text{likelihood}}\ \overbrace{P(c)}^{\text{prior}} \tag{4.5}$$

Without loss of generalization, we can represent a document d as a set of features $f_1, f_2, ..., f_n$:

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}}\, \overbrace{P(f_1, f_2, ..., f_n|c)}^{\text{likelihood}}\ \overbrace{P(c)}^{\text{prior}} \tag{4.6}$$

Unfortunately, Eq. 4.6 is still too hard to compute directly: without some simplifying assumptions, estimating the probability of every possible combination of features (for example, every possible set of words and positions) would require huge numbers of parameters and impossibly large training sets. Naive Bayes classifiers therefore make two simplifying assumptions.

The first is the *bag of words* assumption discussed intuitively above: we assume position doesn't matter, and that the word "love" has the same effect on classification whether it occurs as the 1st, 20th, or last word in the document. Thus we assume that the features $f_1, f_2, ..., f_n$ only encode word identity and not position.

naive Bayes assumption

The second is commonly called the **naive Bayes assumption**: this is the conditional independence assumption that the probabilities $P(f_i|c)$ are independent given the class c and hence can be 'naively' multiplied as follows:

$$P(f_1, f_2, ..., f_n|c) = P(f_1|c) \cdot P(f_2|c) \cdot ... \cdot P(f_n|c) \tag{4.7}$$

The final equation for the class chosen by a naive Bayes classifier is thus:

$$c_{NB} = \underset{c \in C}{\operatorname{argmax}}\, P(c) \prod_{f \in F} P(f|c) \tag{4.8}$$

To apply the naive Bayes classifier to text, we need to consider word positions, by simply walking an index through every word position in the document:

$$\begin{aligned} \text{positions} &\leftarrow \text{all word positions in test document} \\ c_{NB} &= \underset{c \in C}{\operatorname{argmax}} P(c) \prod_{i \in positions} P(w_i|c) \end{aligned} \tag{4.9}$$

Naive Bayes calculations, like calculations for language modeling, are done in log space, to avoid underflow and increase speed. Thus Eq. 4.9 is generally instead expressed as

$$c_{NB} = \underset{c \in C}{\operatorname{argmax}} \log P(c) + \sum_{i \in positions} \log P(w_i|c) \tag{4.10}$$

By considering features in log space Eq. 4.10 computes the predicted class as a linear function of input features. Classifiers that use a linear combination of the inputs to make a classification decision —like naive Bayes and also logistic regression— are called **linear classifiers**.

linear classifiers

4.2 Training the Naive Bayes Classifier

How can we learn the probabilities $P(c)$ and $P(f_i|c)$? Let's first consider the maximum likelihood estimate. We'll simply use the frequencies in the data. For the document prior $P(c)$ we ask what percentage of the documents in our training set are in each class c. Let N_c be the number of documents in our training data with class c and N_{doc} be the total number of documents. Then:

$$\hat{P}(c) = \frac{N_c}{N_{doc}} \tag{4.11}$$

To learn the probability $P(f_i|c)$, we'll assume a feature is just the existence of a word in the document's bag of words, and so we'll want $P(w_i|c)$, which we compute as the fraction of times the word w_i appears among all words in all documents of topic c. We first concatenate all documents with category c into one big "category c" text. Then we use the frequency of w_i in this concatenated document to give a maximum likelihood estimate of the probability:

$$\hat{P}(w_i|c) = \frac{count(w_i, c)}{\sum_{w \in V} count(w, c)} \tag{4.12}$$

Here the vocabulary V consists of the union of all the word types in all classes, not just the words in one class c.

There is a problem, however, with maximum likelihood training. Imagine we are trying to estimate the likelihood of the word "fantastic" given class *positive*, but suppose there are no training documents that both contain the word "fantastic" and are classified as *positive*. Perhaps the word "fantastic" happens to occur (sarcastically?) in the class *negative*. In such a case the probability for this feature will be zero:

$$\hat{P}(\text{“fantastic”}|\text{positive}) = \frac{count(\text{“fantastic”}, \text{positive})}{\sum_{w \in V} count(w, \text{positive})} = 0 \tag{4.13}$$

But since naive Bayes naively multiplies all the feature likelihoods together, zero probabilities in the likelihood term for any class will cause the probability of the class to be zero, no matter the other evidence!

The simplest solution is the add-one (Laplace) smoothing introduced in Chapter 3. While Laplace smoothing is usually replaced by more sophisticated smoothing algorithms in language modeling, it is commonly used in naive Bayes text categorization:

$$\hat{P}(w_i|c) = \frac{count(w_i, c) + 1}{\sum_{w \in V} (count(w, c) + 1)} = \frac{count(w_i, c) + 1}{\left(\sum_{w \in V} count(w, c)\right) + |V|} \tag{4.14}$$

Note once again that it is crucial that the vocabulary V consists of the union of all the word types in all classes, not just the words in one class c (try to convince yourself why this must be true; see the exercise at the end of the chapter).

What do we do about words that occur in our test data but are not in our vocabulary at all because they did not occur in any training document in any class? The solution for such **unknown words** is to ignore them—remove them from the test document and not include any probability for them at all.

unknown word

Finally, some systems choose to completely ignore another class of words: **stop words**, very frequent words like *the* and *a*. This can be done by sorting the vocabulary by frequency in the training set, and defining the top 10–100 vocabulary entries as stop words, or alternatively by using one of the many pre-defined stop word list available online. Then every instance of these stop words are simply removed from both training and test documents as if they had never occurred. In most text classification applications, however, using a stop word list doesn't improve performance, and so it is more common to make use of the entire vocabulary and not use a stop word list.

stop words

Fig. 4.2 shows the final algorithm.

4.3 Worked example

Let's walk through an example of training and testing naive Bayes with add-one smoothing. We'll use a sentiment analysis domain with the two classes positive (+) and negative (-), and take the following miniature training and test documents simplified from actual movie reviews.

	Cat	Documents
Training	-	just plain boring
	-	entirely predictable and lacks energy
	-	no surprises and very few laughs
	+	very powerful
	+	the most fun film of the summer
Test	?	predictable with no fun

The prior $P(c)$ for the two classes is computed via Eq. 4.11 as $\frac{N_c}{N_{doc}}$:

function TRAIN NAIVE BAYES(D, C) **returns** log $P(c)$ and log $P(w|c)$

for each class $c \in C$ # Calculate $P(c)$ terms
  N_{doc} = number of documents in D
  N_c = number of documents from D in class c
  logprior[c] $\leftarrow$ log $\frac{N_c}{N_{doc}}$
  $V \leftarrow$ vocabulary of D
  bigdoc[*c*] $\leftarrow$ **append**(d) **for** d $\in$ D **with** class *c*
  for each word *w* in V # Calculate $P(w|c)$ terms
    count(w,c) $\leftarrow$ # of occurrences of *w* in *bigdoc*[*c*]
    loglikelihood[w,c] $\leftarrow$ log $\frac{count(w,c)\ +\ 1}{\sum_{w' \ in\ V}\ (count\ (w',c)\ +\ 1)}$
return *logprior*, *loglikelihood*, *V*

function TEST NAIVE BAYES(*testdoc*, *logprior*, *loglikelihood*, C, V) **returns** best *c*

for each class $c \in C$
  sum[*c*] $\leftarrow$ *logprior*[*c*]
  for each position *i* in *testdoc*
    word $\leftarrow$ *testdoc[i]*
    if *word* $\in V$
      sum[*c*] $\leftarrow$ *sum*[*c*]+ *loglikelihood*[*word*,*c*]
return argmax$_c$ *sum*[*c*]

Figure 4.2 The naive Bayes algorithm, using add-1 smoothing. To use add-α smoothing instead, change the $+1$ to $+\alpha$ for loglikelihood counts in training.

$$P(-) = \frac{3}{5} \qquad P(+) = \frac{2}{5}$$

The word *with* doesn't occur in the training set, so we drop it completely (as mentioned above, we don't use unknown word models for naive Bayes). The likelihoods from the training set for the remaining three words "predictable", "no", and "fun", are as follows, from Eq. 4.14 (computing the probabilities for the remainder of the words in the training set is left as Exercise 4.**??** (TBD)).

$$\begin{aligned} P(\text{"predictable"}|-) &= \frac{1+1}{14+20} & P(\text{"predictable"}|+) &= \frac{0+1}{9+20} \\ P(\text{"no"}|-) &= \frac{1+1}{14+20} & P(\text{"no"}|+) &= \frac{0+1}{9+20} \\ P(\text{"fun"}|-) &= \frac{0+1}{14+20} & P(\text{"fun"}|+) &= \frac{1+1}{9+20} \end{aligned}$$

For the test sentence S = "predictable with no fun", after removing the word 'with', the chosen class, via Eq. 4.9, is therefore computed as follows:

$$\begin{aligned} P(-)P(S|-) &= \frac{3}{5} \times \frac{2 \times 2 \times 1}{34^3} = 6.1 \times 10^{-5} \\ P(+)P(S|+) &= \frac{2}{5} \times \frac{1 \times 1 \times 2}{29^3} = 3.2 \times 10^{-5} \end{aligned}$$

The model thus predicts the class *negative* for the test sentence.

4.4 Optimizing for Sentiment Analysis

While standard naive Bayes text classification can work well for sentiment analysis, some small changes are generally employed that improve performance.

First, for sentiment classification and a number of other text classification tasks, whether a word occurs or not seems to matter more than its frequency. Thus it often improves performance to clip the word counts in each document at 1 (see the end of the chapter for pointers to these results). This variant is called **binary multinomial naive Bayes** or **binary NB**. The variant uses the same Eq. 4.10 except that for each document we remove all duplicate words before concatenating them into the single big document. Fig. 4.3 shows an example in which a set of four documents (shortened and text-normalized for this example) are remapped to binary, with the modified counts shown in the table on the right. The example is worked without add-1 smoothing to make the differences clearer. Note that the results counts need not be 1; the word *great* has a count of 2 even for Binary NB, because it appears in multiple documents.

binary NB

Four original documents:

- − it was pathetic the worst part was the boxing scenes
- − no plot twists or great scenes
- + and satire and great plot twists
- + great scenes great film

After per-document binarization:

- − it was pathetic the worst part boxing scenes
- − no plot twists or great scenes
- + and satire great plot twists
- + great scenes film

	NB Counts		Binary Counts	
	+	−	+	−
and	2	0	1	0
boxing	0	1	0	1
film	1	0	1	0
great	3	1	2	1
it	0	1	0	1
no	0	1	0	1
or	0	1	0	1
part	0	1	0	1
pathetic	0	1	0	1
plot	1	1	1	1
satire	1	0	1	0
scenes	1	2	1	2
the	0	2	0	1
twists	1	1	1	1
was	0	2	0	1
worst	0	1	0	1

Figure 4.3 An example of binarization for the binary naive Bayes algorithm.

A second important addition commonly made when doing text classification for sentiment is to deal with negation. Consider the difference between *I really like this movie* (positive) and *I didn't like this movie* (negative). The negation expressed by *didn't* completely alters the inferences we draw from the predicate *like*. Similarly, negation can modify a negative word to produce a positive review (*don't dismiss this film*, *doesn't let us get bored*).

A very simple baseline that is commonly used in sentiment to deal with negation is during text normalization to prepend the prefix *NOT_* to every word after a token of logical negation (*n't, not, no, never*) until the next punctuation mark. Thus the phrase

```
didn't like this movie , but I
```

becomes

```
didn't NOT_like NOT_this NOT_movie , but I
```

Newly formed 'words' like *NOT_like*, *NOT_recommend* will thus occur more often in negative document and act as cues for negative sentiment, while words like *NOT_bored*, *NOT_dismiss* will acquire positive associations. We will return in Chapter 15 to the use of parsing to deal more accurately with the scope relationship between these negation words and the predicates they modify, but this simple baseline works quite well in practice.

Finally, in some situations we might have insufficient labeled training data to train accurate naive Bayes classifiers using all words in the training set to estimate positive and negative sentiment. In such cases we can instead derive the positive and negative word features from **sentiment lexicons**, lists of words that are pre-annotated with positive or negative sentiment. Four popular lexicons are the **General Inquirer** (Stone et al., 1966), **LIWC** (Pennebaker et al., 2007), the opinion lexicon of Hu and Liu (2004a) and the MPQA Subjectivity Lexicon (Wilson et al., 2005).

sentiment lexicons

General Inquirer

LIWC

For example the MPQA subjectivity lexicon has 6885 words, 2718 positive and 4912 negative, each marked for whether it is strongly or weakly biased. (Chapter 20 will discuss how these lexicons can be learned automatically.) Some samples of positive and negative words from the MPQA lexicon include:

+ : *admirable, beautiful, confident, dazzling, ecstatic, favor, glee, great*
− : *awful, bad, bias, catastrophe, cheat, deny, envious, foul, harsh, hate*

A common way to use lexicons in a naive Bayes classifier is to add a feature that is counted whenever a word from that lexicon occurs. Thus we might add a feature called 'this word occurs in the positive lexicon', and treat all instances of words in the lexicon as counts for that one feature, instead of counting each word separately. Similarly, we might add as a second feature 'this word occurs in the negative lexicon' of words in the negative lexicon. If we have lots of training data, and if the test data matches the training data, using just two features won't work as well as using all the words. But when training data is sparse or not representative of the test set, using dense lexicon features instead of sparse individual-word features may generalize better.

4.5 Naive Bayes for other text classification tasks

In the previous section we pointed out that naive Bayes doesn't require that our classifier use all the words in the training data as features. In fact features in naive Bayes can express any property of the input text we want.

spam detection

Consider the task of **spam detection**, deciding if a particular piece of email is an example of spam (unsolicited bulk email) — and one of the first applications of naive Bayes to text classification (Sahami et al., 1998).

A common solution here, rather than using all the words as individual features, is to predefine likely sets of words or phrases as features, combined these with features that are not purely linguistic. For example the open-source SpamAssassin tool[1] predefines features like the phrase "one hundred percent guaranteed", or the feature *mentions millions of dollars*, which is a regular expression that matches suspiciously large sums of money. But it also includes features like *HTML has a low ratio of*

[1] https://spamassassin.apache.org

text to image area, that isn't purely linguistic and might require some sophisticated computation, or totally non-linguistic features about, say, the path that the email took to arrive. More sample SpamAssassin features:

- Email subject line is all capital letters
- Contains phrases of urgency like "urgent reply"
- Email subject line contains "online pharmaceutical"
- HTML has unbalanced "head" tags
- Claims you can be removed from the list

language ID

For other tasks, like **language ID**—determining what language a given piece of text is written in—the most effective naive Bayes features are not words at all, but **byte n-grams**, 2-grams ('zw') 3-grams ('nya', ' Vo'), or 4-grams ('ie z', 'thei'). Because spaces count as a byte, byte n-grams can model statistics about the beginning or ending of words. [2] A widely used naive Bayes system, `langid.py` (Lui and Baldwin, 2012) begins with all possible n-grams of lengths 1-4, using **feature selection** to winnow down to the most informative 7000 final features.

Language ID systems are trained on multilingual text, such as Wikipedia (Wikipedia text in 68 different languages were used in (Lui and Baldwin, 2011)), or newswire. To make sure that this multilingual text correctly reflects different regions, dialects, and socio-economic classes, systems also add Twitter text in many languages geotagged to many regions (important for getting world English dialects from countries with large Anglophone populations like Nigeria or India), Bible and Quran translations, slang websites like Urban Dictionary, corpora of African American Vernacular English (Blodgett et al., 2016), and so on (Jurgens et al., 2017).

4.6 Naive Bayes as a Language Model

As we saw in the previous section, naive Bayes classifiers can use any sort of feature: dictionaries, URLs, email addresses, network features, phrases, and so on. But if, as in the previous section, we use only individual word features, and we use all of the words in the text (not a subset), then naive Bayes has an important similarity to language modeling. Specifically, a naive Bayes model can be viewed as a set of class-specific unigram language models, in which the model for each class instantiates a unigram language model.

Since the likelihood features from the naive Bayes model assign a probability to each word $P(word|c)$, the model also assigns a probability to each sentence:

$$P(s|c) = \prod_{i \in positions} P(w_i|c) \tag{4.15}$$

Thus consider a naive Bayes model with the classes *positive* (+) and *negative* (-) and the following model parameters:

[2] It's also possible to use codepoints, which are multi-byte Unicode representations of characters in character sets, but simply using bytes seems to work better.

w	P(w|+)	P(w|-)
I	0.1	0.2
love	0.1	0.001
this	0.01	0.01
fun	0.05	0.005
film	0.1	0.1
...	...	...

Each of the two columns above instantiates a language model that can assign a probability to the sentence "I love this fun film":

$$P(\text{"I love this fun film"}|+) = 0.1 \times 0.1 \times 0.01 \times 0.05 \times 0.1 = 0.0000005$$
$$P(\text{"I love this fun film"}|-) = 0.2 \times 0.001 \times 0.01 \times 0.005 \times 0.1 = .0000000010$$

As it happens, the positive model assigns a higher probability to the sentence: $P(s|pos) > P(s|neg)$. Note that this is just the likelihood part of the naive Bayes model; once we multiply in the prior a full naive Bayes model might well make a different classification decision.

4.7 Evaluation: Precision, Recall, F-measure

To introduce the methods for evaluating text classification, let's first consider some simple binary *detection* tasks. For example, in spam detection, our goal is to label every text as being in the spam category ("positive") or not in the spam category ("negative"). For each item (email document) we therefore need to know whether our system called it spam or not. We also need to know whether the email is actually spam or not, i.e. the human-defined labels for each document that we are trying to match. We will refer to these human labels as the **gold labels**.

gold labels

Or imagine you're the CEO of the *Delicious Pie Company* and you need to know what people are saying about your pies on social media, so you build a system that detects tweets concerning Delicious Pie. Here the positive class is tweets about Delicious Pie and the negative class is all other tweets.

In both cases, we need a metric for knowing how well our spam detector (or pie-tweet-detector) is doing. To evaluate any system for detecting things, we start by building a **contingency table** like the one shown in Fig. 4.4. Each cell labels a set of possible outcomes. In the spam detection case, for example, true positives are documents that are indeed spam (indicated by human-created gold labels) and our system said they were spam. False negatives are documents that are indeed spam but our system labeled as non-spam.

contingency table

To the bottom right of the table is the equation for *accuracy*, which asks what percentage of all the observations (for the spam or pie examples that means all emails or tweets) our system labeled correctly. Although accuracy might seem a natural metric, we generally don't use it. That's because accuracy doesn't work well when the classes are unbalanced (as indeed they are with spam, which is a large majority of email, or with tweets, which are mainly not about pie).

To make this more explicit, imagine that we looked at a million tweets, and let's say that only 100 of them are discussing their love (or hatred) for our pie, while the other 999,900 are tweets about something completely unrelated. Imagine a

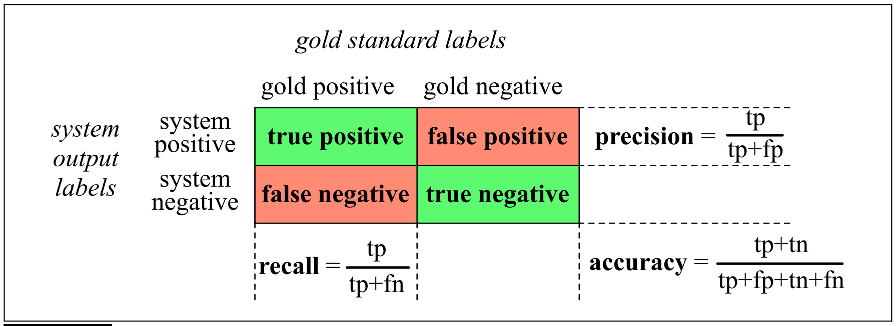


Figure 4.4 Contingency table

simple classifier that stupidly classified every tweet as "not about pie". This classifier would have 999,900 true negatives and only 100 false negatives for an accuracy of 999,900/1,000,000 or 99.99%! What an amazing accuracy level! Surely we should be happy with this classifier? But of course this fabulous 'no pie' classifier would be completely useless, since it wouldn't find a single one of the customer comments we are looking for. In other words, accuracy is not a good metric when the goal is to discover something that is rare, or at least not completely balanced in frequency, which is a very common situation in the world.

precision

That's why instead of accuracy we generally turn to two other metrics: **precision** and **recall**. **Precision** measures the percentage of the items that the system detected (i.e., the system labeled as positive) that are in fact positive (i.e., are positive according to the human gold labels). Precision is defined as

$$\textbf{Precision} = \frac{\text{true positives}}{\text{true positives + false positives}}$$

recall

Recall measures the percentage of items actually present in the input that were correctly identified by the system. Recall is defined as

$$\textbf{Recall} = \frac{\text{true positives}}{\text{true positives + false negatives}}$$

Precision and recall will help solve the problem with the useless "nothing is pie" classifier. This classifier, despite having a fabulous accuracy of 99.99%, has a terrible recall of 0 (since there are no true positives, and 100 false negatives, the recall is 0/100). You should convince yourself that the precision at finding relevant tweets is equally problematic. Thus precision and recall, unlike accuracy, emphasize true positives: finding the things that we are supposed to be looking for.

F-measure

There are many ways to define a single metric that incorporates aspects of both precision and recall. The simplest of these combinations is the ***F*-measure** (van Rijsbergen, 1975) , defined as:

$$F_\beta = \frac{(\beta^2+1)PR}{\beta^2 P + R}$$

F1

The β parameter differentially weights the importance of recall and precision, based perhaps on the needs of an application. Values of $\beta > 1$ favor recall, while values of $\beta < 1$ favor precision. When $\beta = 1$, precision and recall are equally balanced; this is the most frequently used metric, and is called $F_{\beta=1}$ or just F_1:

$$F_1 = \frac{2PR}{P+R} \tag{4.16}$$

F-measure comes from a weighted harmonic mean of precision and recall. The harmonic mean of a set of numbers is the reciprocal of the arithmetic mean of reciprocals:

$$\text{HarmonicMean}(\mathrm{a}_1, \mathrm{a}_2, \mathrm{a}_3, \mathrm{a}_4, ..., \mathrm{a}_\mathrm{n}) = \frac{\mathrm{n}}{\frac{1}{\mathrm{a}_1} + \frac{1}{\mathrm{a}_2} + \frac{1}{\mathrm{a}_3} + ... + \frac{1}{\mathrm{a}_\mathrm{n}}} \tag{4.17}$$

and hence F-measure is

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} \quad \text{or} \left(\text{with } \beta^2 = \frac{1-\alpha}{\alpha}\right) \quad F = \frac{(\beta^2+1)PR}{\beta^2 P + R} \tag{4.18}$$

Harmonic mean is used because it is a conservative metric; the harmonic mean of two values is closer to the minimum of the two values than the arithmetic mean is. Thus it weighs the lower of the two numbers more heavily.

4.7.1 More than two classes

Up to now we have been assuming text classification tasks with only two classes. But lots of classification tasks in language processing have more than two classes. For sentiment analysis we generally have 3 classes (positive, negative, neutral) and even more classes are common for tasks like part-of-speech tagging, word sense disambiguation, semantic role labeling, emotion detection, and so on.

any-of

There are two kinds of multi-class classification tasks. In **any-of** or **multi-label classification**, each document or item can be assigned more than one label. We can solve *any-of* classification by building separate binary classifiers for each class c, trained on positive examples labeled c and negative examples not labeled c. Given a test document or item d, then each classifier makes their decision independently, and we may assign multiple labels to d.

one-of
multinomial classification

More common in language processing is **one-of** or **multinomial classification**, in which the classes are mutually exclusive and each document or item appears in exactly one class. Here we again build a separate binary classifier trained on positive examples from c and negative examples from all other classes. Now given a test document or item d, we run all the classifiers and choose the label from the classifier with the highest score. Consider the sample confusion matrix for a hypothetical 3-way *one-of* email categorization decision (urgent, normal, spam) shown in Fig. 4.5.

macroaveraging
microaveraging

The matrix shows, for example, that the system mistakenly labeled 1 spam document as urgent, and we have shown how to compute a distinct precision and recall value for each class. In order to derive a single metric that tells us how well the system is doing, we can combine these values in two ways. In **macroaveraging**, we compute the performance for each class, and then average over classes. In **microaveraging**, we collect the decisions for all classes into a single contingency table, and then compute precision and recall from that table. Fig. 4.6 shows the contingency table for each class separately, and shows the computation of microaveraged and macroaveraged precision.

As the figure shows, a microaverage is dominated by the more frequent class (in this case spam), since the counts are pooled. The macroaverage better reflects the statistics of the smaller classes, and so is more appropriate when performance on all the classes is equally important.

		gold labels: urgent	gold labels: normal	gold labels: spam	
system output	urgent	8	10	1	**precision**$_u$ = $\frac{8}{8+10+1}$
	normal	5	60	50	**precision**$_n$ = $\frac{60}{5+60+50}$
	spam	3	30	200	**precision**$_s$ = $\frac{200}{3+30+200}$
		recall$_u$ = $\frac{8}{8+5+3}$	**recall**$_n$ = $\frac{60}{10+60+30}$	**recall**$_s$ = $\frac{200}{1+50+200}$	

Figure 4.5 Confusion matrix for a three-class categorization task, showing for each pair of classes (c_1, c_2), how many documents from c_1 were (in)correctly assigned to c_2

Class 1: Urgent

	true urgent	true not
system urgent	8	11
system not	8	340

$$\text{precision} = \frac{8}{8+11} = .42$$

Class 2: Normal

	true normal	true not
system normal	60	55
system not	40	212

$$\text{precision} = \frac{60}{60+55} = .52$$

Class 3: Spam

	true spam	true not
system spam	200	33
system not	51	83

$$\text{precision} = \frac{200}{200+33} = .86$$

Pooled

	true yes	true no
system yes	268	99
system no	99	635

$$\text{microaverage precision} = \frac{268}{268+99} = \mathbf{.73}$$

$$\text{macroaverage precision} = \frac{.42+.52+.86}{3} = \mathbf{.60}$$

Figure 4.6 Separate contingency tables for the 3 classes from the previous figure, showing the pooled contingency table and the microaveraged and macroaveraged precision.

4.8 Test sets and Cross-validation

development test set
devset

The training and testing procedure for text classification follows what we saw with language modeling (Section 3.2): we use the training set to train the model, then use the **development test set** (also called a **devset**) to perhaps tune some parameters, and in general decide what the best model is. Once we come up with what we think is the best model, we run it on the (hitherto unseen) test set to report its performance.

cross-validation

While the use of a devset avoids overfitting the test set, having a fixed training set, devset, and test set creates another problem: in order to save lots of data for training, the test set (or devset) might not be large enough to be representative. It would be better if we could somehow use **all** our data both for training and test. We do this by **cross-validation**: we randomly choose a training and test set division of our data, train our classifier, and then compute the error rate on the test set. Then we repeat with a different randomly selected training set and test set. We do this sampling process 10 times and average these 10 runs to get an average error rate. This is called **10-fold cross-validation**.

10-fold cross-validation

The only problem with cross-validation is that because all the data is used for testing, we need the whole corpus to be blind; we can't examine any of the data to suggest possible features and in general see what's going on. But looking at the corpus is often important for designing the system. For this reason, it is common

to create a fixed training set and test set, then do 10-fold cross-validation inside the training set, but compute error rate the normal way in the test set, as shown in Fig. 4.7.

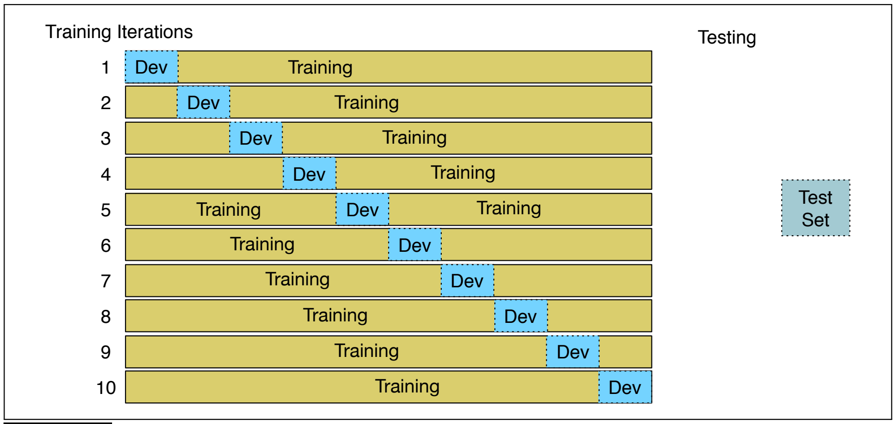


Figure 4.7 10-fold cross-validation

4.9 Statistical Significance Testing

In building systems we are constantly comparing the performance of systems. Often we have added some new bells and whistles to our algorithm and want to compare the new version of the system to the unaugmented version. Or we want to compare our algorithm to a previously published one to know which is better.

We might imagine that to compare the performance of two classifiers A and B all we have to do is look at A and B's score on the same test set—for example we might choose to compare macro-averaged F1— and see whether it's A or B that has the higher score. But just looking at this one difference isn't good enough, because A might have a better performance than B on a particular test set just by chance.

Let's say we have a test set x of n observations $x = x_1, x_2, .., x_n$ on which A's performance is better than B by $\delta(x)$. How can we know if A is really better than B? To do so we'd need to reject the **null hypothesis** that A isn't really better than B and this difference $\delta(x)$ occurred purely by chance. If the null hypothesis was correct, we would expect that if we had many test sets of size n and we measured A and B's performance on all of them, that on average A might accidentally still be better than B by this amount $\delta(x)$ just by chance.

null hypothesis

More formally, if we had a random variable X ranging over test sets, the null hypothesis H_0 expects $P(\delta(X) > \delta(x)|H_0)$, the probability that we'll see similarly big differences just by chance, to be high.

If we had all these test sets we could just measure all the $\delta(x')$ for all the x'. If we found that those deltas didn't seem to be bigger than $\delta(x)$, that is, that p-value(x) was sufficiently small, less than the standard thresholds of 0.05 or 0.01, then we might reject the null hypothesis and agree that $\delta(x)$ was a sufficiently surprising difference and A is really a better algorithm than B. Following Berg-Kirkpatrick et al. (2012) we'll refer to $P(\delta(X) > \delta(x)|H_0)$ as p-value(x).

In language processing we don't generally use traditional statistical approaches like paired t-tests to compare system outputs because most metrics are not normally

distributed, violating the assumptions of the tests. The standard approach to computing p-value(x) in natural language processing is to use non-parametric tests like the **bootstrap test** (Efron and Tibshirani, 1993)— which we will describe below—or a similar test, **approximate randomization** (Noreen, 1989). The advantage of these tests is that they can apply to any metric; from precision, recall, or F1 to the BLEU metric used in machine translation.

bootstrap test

approximate randomization

bootstrapping

The word **bootstrapping** refers to repeatedly drawing large numbers of smaller samples with replacement (called **bootstrap samples**) from an original larger sample. The intuition of the bootstrap test is that we can create many virtual test sets from an observed test set by repeatedly sampling from it. The method only makes the assumption that the sample is representative of the population.

Consider a tiny text classification example with a test set x of 10 documents. The first row of Fig. 4.8 shows the results of two classifiers (A and B) on this test set, with each document labeled by one of the four possibilities: (A and B both right, both wrong, A right and B wrong, A wrong and B right); a slash through a letter ($\cancel{\text{B}}$) means that that classifier got the answer wrong. On the first document both A and B get the correct class (AB), while on the second document A got it right but B got it wrong ($\text{A}\cancel{\text{B}}$). If we assume for simplicity that our metric is accuracy, A has an accuracy of .70 and B of .50, so $\delta(x)$ is .20. To create each virtual test set of size $N = 10$, we repeatedly (10 times) select a cell from row x with replacement. Fig. 4.8 shows a few examples.

	1	2	3	4	5	6	7	8	9	10	A%	B%	$\delta()$
x	AB	$\text{A}\cancel{\text{B}}$	AB	$\cancel{\text{A}}\text{B}$	$\text{A}\cancel{\text{B}}$	$\cancel{\text{A}}\text{B}$	$\text{A}\cancel{\text{B}}$	AB	$\cancel{\text{A}}\cancel{\text{B}}$	$\text{A}\cancel{\text{B}}$	.70	.50	.20
$x^{*(1)}$	$\text{A}\cancel{\text{B}}$	AB	$\text{A}\cancel{\text{B}}$	$\cancel{\text{A}}\text{B}$	$\cancel{\text{A}}\text{B}$	$\text{A}\cancel{\text{B}}$	$\cancel{\text{A}}\text{B}$	AB	$\cancel{\text{A}}\cancel{\text{B}}$	AB	.60	.60	.00
$x^{*(2)}$	$\text{A}\cancel{\text{B}}$	AB	$\cancel{\text{A}}\cancel{\text{B}}$	$\cancel{\text{A}}\text{B}$	$\cancel{\text{A}}\text{B}$	AB	$\cancel{\text{A}}\text{B}$	$\text{A}\cancel{\text{B}}$	AB	AB	.60	.70	-.10
...													
$x^{*(b)}$													

Figure 4.8 The bootstrap: Examples of b pseudo test sets being created from an initial true test set x. Each pseudo test set is created by sampling $n = 10$ times with replacement; thus an individual sample is a single cell, a document with its gold label and the correct or incorrect performance of classifiers A and B.

Now that we have a sampling distribution, we can do statistics on how how often A has an accidental advantage. There are various ways to compute this advantage; here we follow the version laid out in Berg-Kirkpatrick et al. (2012). We might think that we should just ask, for each bootstrap sample $x^{*(i)}$, whether A beats B by more than $\delta(x)$. But there's a problem: we didn't draw these samples from a distribution with 0 mean. The $x^{*(i)}$ were sampled from x, and so the expected value of $\delta(x^{*(i)})$ lies very close to $\delta(x)$. That is, about half the time A will be better than B, so we *expect* A to beat B by $\delta(x)$. Instead, we want to know how often A beats these expectations by more than $\delta(x)$. To correct for the expected success, we need to zero-center, subtracting $\delta(x)$ from each pseudo test set. Thus we'll be comparing for each $x^{*(i)}$ whether $\delta(x^{*(i)}) > 2\delta(x)$. The full algorithm for the bootstrap is shown in Fig. 4.9. It is given a test set x, a number of samples b, and counts the percentage of the b bootstrap test sets in which $delta(x^{*(i)}) > 2\delta(x)$. This percentage then acts as a one-sided empirical p-value (more sophisticated ways to get p-values from confidence intervals also exist).

```
function BOOTSTRAP(test set x, num of samples b) returns p-value(x)

Calculate δ(x)   # how much better does algorithm A do than B on x
for i = 1 to b do
   for j = 1 to n do    # Draw a bootstrap sample x*(i) of size n
      Select a member of x at random and add it to x*(i)
   Calculate δ(x*(i))   # how much better does algorithm A do than B on x*(i)
for each x*(i)
   s←s + 1 if δ(x*(i)) > 2δ(x)
p-value(x) ≈ s/b   # on what % of the b samples did algorithm A beat expectations?
return p-value(x)
```

Figure 4.9 A version of the bootstrap algorithm after Berg-Kirkpatrick et al. (2012).

4.10 Advanced: Feature Selection

Feature selection

The regularization technique introduced in the previous section is **feature selection** is a method of removing features that are unlikely to generalize well. The basis of feature selection is to assign some metric of goodness to each feature, rank the features, and keep the best ones. The number of features to keep is a meta-parameter that can be optimized on a dev set.

information gain

Features are generally ranked by how informative they are about the classification decision. A very common metric is **information gain**. Information gain tells us how many bits of information the presence of the word gives us for guessing the class, and can be computed as follows (where c_i is the ith class and $\bar{w}$ means that a document does not contain the word w):

$$\begin{aligned} G(w) = \quad & -\sum_{i=1}^{C} P(c_i)\log P(c_i) \\ & +P(w)\sum_{i=1}^{C} P(c_i|w)\log P(c_i|w) \\ & +P(\bar{w})\sum_{i=1}^{C} P(c_i|\bar{w})\log P(c_i|\bar{w}) \end{aligned} \tag{4.19}$$

4.11 Summary

This chapter introduced the **naive Bayes** model for **classification** and applied it to the **text categorization** task of **sentiment analysis**.

- Many language processing tasks can be viewed as tasks of **classification**. learn to model the class given the observation.
- Text categorization, in which an entire text is assigned a class from a finite set, includes such tasks as **sentiment analysis**, **spam detection**, language identification, and authorship attribution.
- Sentiment analysis classifies a text as reflecting the positive or negative orientation (**sentiment**) that a writer expresses toward some object.

- Naive Bayes is a **generative** model that make the bag of words assumption (position doesn't matter) and the conditional independence assumption (words are conditionally independent of each other given the class)
- Naive Bayes with binarized features seems to work better for many text classification tasks.
- Feature selection can be used to automatically remove features that aren't helpful.
- Classifiers are evaluated based on **precision** and **recall**.
- Classifiers are trained using distinct training, dev, and test sets, including the use of **cross-validation** in the training set.

Bibliographical and Historical Notes

Multinomial naive Bayes text classification was proposed by Maron (1961) at the RAND Corporation for the task of assigning subject categories to journal abstracts. His model introduced most of the features of the modern form presented here, approximating the classification task with one-of categorization, and implementing add-δ smoothing and information-based feature selection.

The conditional independence assumptions of naive Bayes and the idea of Bayesian analysis of text seem to have been arisen multiple times. The same year as Maron's paper, Minsky (1961) proposed a naive Bayes classifier for vision and other artificial intelligence problems, and Bayesian techniques were also applied to the text classification task of authorship attribution by Mosteller and Wallace (1963). It had long been known that Alexander Hamilton, John Jay, and James Madison wrote the anonymously-published *Federalist* papers. in 1787–1788 to persuade New York to ratify the United States Constitution. Yet although some of the 85 essays were clearly attributable to one author or another, the authorship of 12 were in dispute between Hamilton and Madison. Mosteller and Wallace (1963) trained a Bayesian probabilistic model of the writing of Hamilton and another model on the writings of Madison, then computed the maximum-likelihood author for each of the disputed essays. Naive Bayes was first applied to spam detection in Heckerman et al. (1998).

Metsis et al. (2006), Pang et al. (2002), and Wang and Manning (2012) show that using boolean attributes with multinomial naive Bayes works better than full counts. Binary multinomial naive Bayes is sometimes confused with another variant of naive Bayes that also use a binary representation of whether a term occurs in a document: **Multivariate Bernoulli naive Bayes**. The Bernoulli variant instead estimates $P(w|c)$ as the fraction of documents that contain a term, and includes a probability for whether a term is *not* in a document. McCallum and Nigam (1998) and Wang and Manning (2012) show that the multivariate Bernoulli variant of naive Bayes doesn't work as well as the multinomial algorithm for sentiment or other text tasks.

There are a variety of sources covering the many kinds of text classification tasks. For sentiment analysis see Pang and Lee (2008), and Liu and Zhang (2012). Stamatatos (2009) surveys authorship attribute algorithms. On language identification see Jauhiainen et al. (2018); Jaech et al. (2016) is an important early neural system. The task of newswire indexing was often used as a test case for text classification algorithms, based on the Reuters-21578 collection of newswire articles.

See Manning et al. (2008) and Aggarwal and Zhai (2012) on text classification; classification in general is covered in machine learning textbooks (Hastie et al. 2001,

Witten and Frank 2005, Bishop 2006, Murphy 2012).

Non-parametric methods for computing statistical significance were used first in NLP in the MUC competition (Chinchor et al., 1993), and even earlier in speech recognition (Gillick and Cox 1989, Bisani and Ney 2004). Our description of the bootstrap draws on the description in Berg-Kirkpatrick et al. (2012). Recent work has focused on issues including multiple test sets and multiple metrics (Søgaard et al. 2014, Dror et al. 2017).

Metrics besides information gain for feature selection include χ^2, pointwise mutual information, and GINI index; see Yang and Pedersen (1997) for a comparison and Guyon and Elisseeff (2003) for a broad introduction survey of feature selection.

Exercises

4.1 Assume the following likelihoods for each word being part of a positive or negative movie review, and equal prior probabilities for each class.

	pos	neg
I	0.09	0.16
always	0.07	0.06
like	0.29	0.06
foreign	0.04	0.15
films	0.08	0.11

What class will Naive bayes assign to the sentence “I always like foreign films.”?

4.2 Given the following short movie reviews, each labeled with a genre, either comedy or action:

1. fun, couple, love, love **comedy**
2. fast, furious, shoot **action**
3. couple, fly, fast, fun, fun **comedy**
4. furious, shoot, shoot, fun **action**
5. fly, fast, shoot, love **action**

and a new document D:

fast, couple, shoot, fly

compute the most likely class for D. Assume a naive Bayes classifier and use add-1 smoothing for the likelihoods.

4.3 Train two models, multinominal naive Bayes and binarized naive Bayes, both with add-1 smoothing, on the following document counts for key sentiment words, with positive or negative class assigned as noted.

doc	“good”	“poor”	“great”	(class)
d1.	3	0	3	pos
d2.	0	1	2	pos
d3.	1	3	0	neg
d4.	1	5	2	neg
d5.	0	2	0	neg

Use both naive Bayes models to assign a class (pos or neg) to this sentence:

A good, good plot and great characters, but poor acting.

Do the two models agree or disagree?

CHAPTER 5

Logistic Regression

"And how do you know that these fine begonias are not of equal importance?"
Hercule Poirot, in Agatha Christie's *The Mysterious Affair at Styles*

Detective stories are as littered with clues as texts are with words. Yet for the poor reader it can be challenging to know how to weigh the author's clues in order to make the crucial classification task: deciding whodunnit.

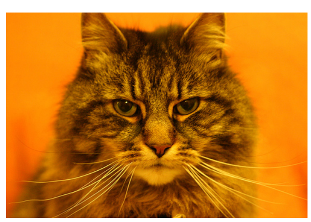
logistic regression

In this chapter we introduce an algorithm that is admirably suited for discovering the link between features or cues and some particular outcome: **logistic regression**. Indeed, logistic regression is one of the most important analytic tool in the social and natural sciences. In natural language processing, logistic regression is the baseline supervised machine learning algorithm for classification, and also has a very close relationship with neural networks. As we will see in Chapter 7, a neural network can be viewed as a series of logistic regression classifiers stacked on top of each other. Thus the classification and machine learning techniques introduced here will play an important role throughout the book.

Logistic regression can be used to classify an observation into one of two classes (like 'positive sentiment' and 'negative sentiment'), or into one of many classes. Because the mathematics for the two-class case is simpler, we'll describe this special case of logistic regression first in the next few sections, and then briefly summarize the use of **multinomial logistic regression** for more than two classes in Section 5.6.

We'll introduce the mathematics of logistic regression in the next few sections. But let's begin with some high-level issues.

Generative and Discriminative Classifiers: The most important difference between naive Bayes and logistic regression is that logistic regression is a **discriminative** classifier while naive Bayes is a **generative** classifier.



These are two very different frameworks for how to build a machine learning model. Consider a visual metaphor: imagine we're trying to distinguish dog images from cat images. A generative model would have the goal of understanding what dogs look like and what cats look like. You might literally ask such a model to 'generate', i.e. draw, a dog. Given a test image, the system then asks whether it's the cat model or the dog model that better fits (is less surprised by) the image, and chooses that as its label.

A discriminative model, by contrast, is only trying to learn to distinguish the classes (perhaps without learning much about them). So maybe all the dogs in the training data are wearing collars and the cats aren't. If that one feature neatly separates the classes, the model is satisfied. If you ask such a model what it knows about cats all it can say is that they don't wear collars.

More formally, recall that the naive Bayes assigns a class c to a document d not by directly computing $P(c|d)$ but by computing a likelihood and a prior

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \; \overbrace{P(d|c)}^{\text{likelihood}} \; \overbrace{P(c)}^{\text{prior}} \tag{5.1}$$

generative model A **generative model** like naive Bayes makes use of this **likelihood** term, which expresses how to generate the features of a document *if we knew it was of class c*.

discriminative model By contrast a **discriminative model** in this text categorization scenario attempts to **directly** compute $P(c|d)$. Perhaps it will learn to assign high weight to document features that directly improve its ability to *discriminate* between possible classes, even if it couldn't generate an example of one of the classes.

Components of a probabilistic machine learning classifier: Like naive Bayes, logistic regression is a probabilistic classifier that makes use of supervised machine learning. Machine learning classifiers require a training corpus of M observations input/output pairs $(x^{(i)}, y^{(i)})$. (We'll use superscripts in parentheses to refer to individual instances in the training set—for sentiment classification each instance might be an individual document to be classified). A machine learning system for classification then has four components:

1. A **feature representation** of the input. For each input observation $x^{(i)}$, this will be a vector of features $[x_1, x_2, ..., x_n]$. We will generally refer to feature i for input $x^{(j)}$ as $x_i^{(j)}$, sometimes simplified as x_i, but we will also see the notation f_i, $f_i(x)$, or, for multiclass classification, $f_i(c,x)$.
2. A classification function that computes $\hat{y}$, the estimated class, via $p(y|x)$. In the next section we will introduce the **sigmoid** and **softmax** tools for classification.
3. An objective function for learning, usually involving minimizing error on training examples. We will introduce the **cross-entropy loss function**
4. An algorithm for optimizing the objective function. We introduce the **stochastic gradient descent** algorithm.

Logistic regression has two phases:

training: we train the system (specifically the weights w and b) using stochastic gradient descent and the cross-entropy loss.

test: Given a test example x we compute $p(y|x)$ and return the higher probability label $y = 1$ or $y = 0$.

5.1 Classification: the sigmoid

The goal of binary logistic regression is to train a classifier that can make a binary decision about the class of a new input observation. Here we introduce the **sigmoid** classifier that will help us make this decision.

Consider a single input observation x, which we will represent by a vector of features $[x_1, x_2, ..., x_n]$ (we'll show sample features in the next subsection). The classifier output y can be 1 (meaning the observation is a member of the class) or 0 (the observation is not a member of the class). We want to know the probability $P(y = 1|x)$ that this observation is a member of the class. So perhaps the decision

is "positive sentiment" versus "negative sentiment", the features represent counts of words in a document, and $P(y=1|x)$ is the probability that the document has positive sentiment, while and $P(y=0|x)$ is the probability that the document has negative sentiment.

Logistic regression solves this task by learning, from a training set, a vector of **weights** and a **bias term**. Each weight w_i is a real number, and is associated with one of the input features x_i. The weight w_i represents how important that input feature is to the classification decision, and can be positive (meaning the feature is associated with the class) or negative (meaning the feature is not associated with the class). Thus we might expect in a sentiment task the word *awesome* to have a high positive weight, and *abysmal* to have a very negative weight. The **bias term**, also called the **intercept**, is another real number that's added to the weighted inputs.

bias term

intercept

To make a decision on a test instance— after we've learned the weights in training— the classifier first multiplies each x_i by its weight w_i, sums up the weighted features, and adds the bias term b. The resulting single number z expresses the weighted sum of the evidence for the class.

$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b \tag{5.2}$$

dot product

In the rest of the book we'll represent such sums using the **dot product** notation from linear algebra. The dot product of two vectors a and b, written as $a \cdot b$ is the sum of the products of the corresponding elements of each vector. Thus the following is an equivalent formation to Eq. 5.2:

$$z = w \cdot x + b \tag{5.3}$$

But note that nothing in Eq. 5.3 forces z to be a legal probability, that is, to lie between 0 and 1. In fact, since weights are real-valued, the output might even be negative; z ranges from $-\infty$ to ∞.

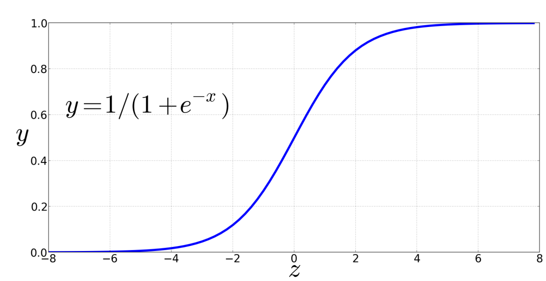


Figure 5.1 The sigmoid function $y = \frac{1}{1+e^{-z}}$ takes a real value and maps it to the range $[0,1]$. Because it is nearly linear around 0 but has a sharp slope toward the ends, it tends to squash outlier values toward 0 or 1.

sigmoid

logistic function

To create a probability, we'll pass z through the **sigmoid** function, $\sigma(z)$. The sigmoid function (named because it looks like an s) is also called the **logistic function**, and gives logistic regression its name. The sigmoid has the following equation, shown graphically in Fig. 5.1:

$$y = \sigma(z) = \frac{1}{1+e^{-z}} \tag{5.4}$$

The sigmoid has a number of advantages; it take a real-valued number and maps it into the range $[0,1]$, which is just what we want for a probability. Because it is nearly linear around 0 but has a sharp slope toward the ends, it tends to squash outlier values toward 0 or 1. And it's differentiable, which as we'll see in Section 5.8 will be handy for learning.

We're almost there. If we apply the sigmoid to the sum of the weighted features, we get a number between 0 and 1. To make it a probability, we just need to make sure that the two cases, $p(y=1)$ and $p(y=0)$, sum to 1. We can do this as follows:

$$\begin{aligned} P(y=1) &= \sigma(w \cdot x + b) \\ &= \frac{1}{1+e^{-(w \cdot x+b)}} \\ P(y=0) &= 1-\sigma(w \cdot x+b) \\ &= 1-\frac{1}{1+e^{-(w \cdot x+b)}} \\ &= \frac{e^{-(w \cdot x+b)}}{1+e^{-(w \cdot x+b)}} \end{aligned} \tag{5.5}$$

Now we have an algorithm that given an instance x computes the probability $P(y=1|x)$. How do we make a decision? For a test instance x, we say yes if the probability $P(y=1|x)$ is more than .5, and yes otherwise. We call .5 the **decision boundary**:

decision boundary

$$\hat{y} = \begin{cases} 1 & \text{if } P(y=1|x) > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

5.1.1 Example: sentiment classification

Let's have an example. Suppose we are doing binary sentiment classification on movie review text, and we would like to know whether to assign the sentiment class $+$ or $-$ to a review document *doc*. We'll represent each input observation by the following 6 features $x_1 \dots x_6$ of the input; Fig. 5.2 shows the features in a sample mini test document.

Var	Definition	Value in Fig. 5.2
x_1	count(positive lexicon) $\in$ doc)	3
x_2	count(negative lexicon) $\in$ doc)	2
x_3	$\begin{cases} 1 & \text{if "no"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	1
x_4	count(1st and 2nd pronouns $\in$ doc)	3
x_5	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	0
x_6	log(word count of doc)	$\ln(64) = 4.15$

Let's assume for the moment that we've already learned a real-valued weight for each of these features, and that the 6 weights corresponding to the 6 features are $[2.5, -5.0, -1.2, 0.5, 2.0, 0.7]$, while b = 0.1. (We'll discuss in the next section how the weights are learned.) The weight w_1, for example indicates how important

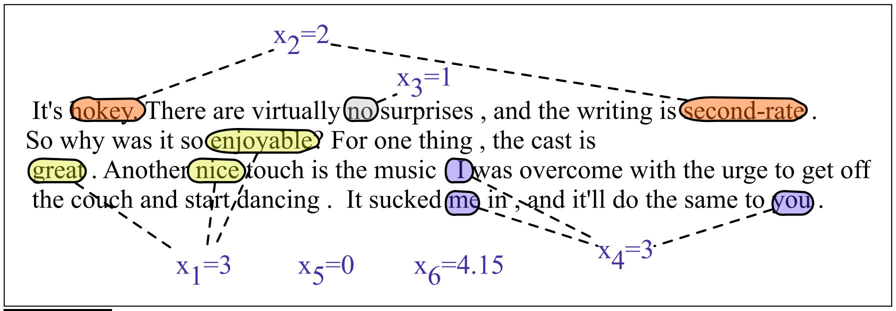


Figure 5.2 A sample mini test document showing the extracted features in the vector x.

a feature the number of positive lexicon words (*great*, *nice*, *enjoyable*, etc.) is to a positive sentiment decision, while w_2 tells us the importance of negative lexicon words. Note that $w_1 = 2.5$ is positive, while $w_2 = -5.0$, meaning that negative words are negatively associated with a positive sentiment decision, and are about twice as important as positive words.

Given these 6 features and the input review x, $P(+|x)$ and $P(-|x)$ can be computed using Eq. 5.5:

$$\begin{aligned}
p(+|x) = P(Y=1|x) &= \sigma(w\cdot x+b)\\
&= \sigma([2.5,-5.0,-1.2,0.5,2.0,0.7]\cdot[3,2,1,3,0,4.15]+0.1)\\
&= \sigma(1.805)\\
&= 0.86\\
p(-|x) = P(Y=0|x) &= 1-\sigma(w\cdot x+b)\\
&= 0.14
\end{aligned}$$

Logistic regression is commonly applied to all sorts of NLP tasks, and any property of the input can be a feature. Consider the task of **period disambiguation**: deciding if a period is the end of a sentence or part of a word, by classifying each period into one of two classes EOS (end-of-sentence) and not-EOS. We might use features like x_1 below expressing that the current word is lower case and the class is EOS (perhaps with a positive weight), or that the current word is in our abbreviations dictionary ("Prof.") and the class is EOS (perhaps with a negative weight). A feature can also express a quite complex combination of properties. For example a period following a upper cased word is a likely to be an EOS, but if the word itself is *St.* and the previous word is capitalized, then the period is likely part of a shortening of the word *street*.

$$\begin{aligned}
x_1 &= \begin{cases} 1 & \text{if “}Case(w_i) = \text{Lower”} \\ 0 & \text{otherwise} \end{cases}\\
x_2 &= \begin{cases} 1 & \text{if “}w_i \in \text{AcronymDict”} \\ 0 & \text{otherwise} \end{cases}\\
x_3 &= \begin{cases} 1 & \text{if “}w_i = \text{St. \& }Case(w_{i-1}) = \text{Cap”} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

Designing features: Features are generally designed by examining the training set with an eye to linguistic intuitions and the linguistic literature on the domain. A careful error analysis on the training or dev set. of an early version of a system often provides insights into features.

For some tasks it is especially helpful to build complex features that are combinations of more primitive features. We saw such a feature for period disambiguation above, where a period on the word *St.* was less likely to be the end of sentence if the previous word was capitalized. For logistic regression and naive Bayes these combination features or **feature interactions** have to be designed by hand.

feature interactions

For many tasks (especially when feature values can reference specific words) we'll need large numbers of features. Often these are created automatically via **feature templates**, abstract specifications of features. For example a bigram template for period disambiguation might create a feature for every pair of words that occurs before a period in the training set. Thus the feature space is sparse, since we only have to create a feature if that n-gram exists in that position in the training set. The feature is generally created as a hash from the string descriptions. A user description of a feature as, "bigram(American breakfast)" is hashed into a unique integer i that becomes the feature number f_i.

feature templates

In order to avoid the extensive human effort of feature design, recent research in NLP has focused on **representation learning**: ways to learn features automatically in an unsupervised way from the input. We'll introduce methods for representation learning in Chapter 6 and Chapter 7.

Choosing a classifier Logistic regression has a number of advantages over naive Bayes. Naive Bayes has overly strong conditional independence assumptions. Consider two features which are strongly correlated; in fact, imagine that we just add the same feature f_1 twice. Naive Bayes will treat both copies of f_1 as if they were separate, multiplying them both in, overestimating the evidence. By contrast, logistic regression is much more robust to correlated features; if two features f_1 and f_2 are perfectly correlated, regression will simply assign part of the weight to w_1 and part to w_2. Thus when there are many correlated features, logistic regression will assign a more accurate probability than naive Bayes. So logistic regression generally works better on larger documents or datasets and is a common default.

Despite the less accurate probabilities, naive Bayes still often makes the correct classification decision. Furthermore, naive Bayes works extremely well (even better than logistic regression) on very small datasets (Ng and Jordan, 2002) or short documents (Wang and Manning, 2012). Furthermore, naive Bayes is easy to implement and very fast to train (there's no optimization step). So it's still a reasonable approach to use in some situations.

5.2 Learning in Logistic Regression

How are the parameters of the model, the weights w and bias b, learned?

Logistic regression is an instance of supervised classification in which we know the correct label y (either 0 or 1) for each observation x. What the system produces, via Eq. 5.5 is $\hat{y}$, the system's estimate of the true y. We want to learn parameters (meaning w and b) that make $\hat{y}$ for each training observation as close as possible to the true y .

This requires 2 components that we foreshadowed in the introduction to the chapter. The first is a metric for how close the current label ($\hat{y}$) is to the true gold label y. Rather than measure similarity, we usually talk about the opposite of this: the *distance* between the system output and the gold output, and we call this distance the **loss** function or the **cost function**. In the next section we'll introduce the loss

loss

function that is commonly used for logistic regression and also for neural networks, the **cross-entropy loss**.

The second thing we need is an optimization algorithm for iteratively updating the weights so as to minimize this loss function. The standard algorithm for this is **gradient descent**; we'll introduce the **stochastic gradient descent** algorithm in the following section.

5.3 The cross-entropy loss function

We need a loss function that expresses, for an observation x, how close the classifier output ($\hat{y} = \sigma(w \cdot x + b)$) is to the correct output (y, which is 0 or 1). We'll call this:

$$L(\hat{y}, y) \;=\; \text{How much } \hat{y} \text{ differs from the true } y \tag{5.6}$$

You could imagine using a simple loss function that just takes the mean squared error between $\hat{y}$ and y.

$$L_{\text{MSE}}(\hat{y}, y) \;=\; \frac{1}{2}(\hat{y} - y)^2 \tag{5.7}$$

It turns out that this MSE loss, which is very useful for some algorithms like linear regression, becomes harder to optimize (technically, non-convex), when it's applied to probabilistic classification.

Instead, we use a loss function that prefers the correct class labels of the training example to be *more likely*. This is called **conditional maximum likelihood estimation**: we choose the parameters w, b that **maximize the log probability of the true** y **labels in the training data** given the observations x. The resulting loss function is the negative log likelihood loss, generally called the **cross entropy loss**.

cross entropy loss

Let's derive this loss function, applied to a single observation x. We'd like to learn weights that maximize the probability of the correct label $p(y|x)$. Since there are only two discrete outcomes (1 or 0), this is a Bernoulli distribution, and we can express the probability $p(y|x)$ that our classifier produces for one observation as the following (keeping in mind that if y=1, Eq. 5.8 simplifies to $\hat{y}$; if y=0, Eq. 5.8 simplifies to $1 - \hat{y}$):

$$p(y|x) \;=\; \hat{y}^{\,y}\,(1-\hat{y})^{1-y} \tag{5.8}$$

Now we take the log of both sides. This will turn out to be handy mathematically, and doesn't hurt us; whatever values maximize a probability will also maximize the log of the probability:

$$\begin{aligned}\log p(y|x) &= \log\left[\hat{y}^{\,y}\,(1-\hat{y})^{1-y}\right]\\ &= y\log\hat{y} + (1-y)\log(1-\hat{y})\end{aligned} \tag{5.9}$$

Eq. 5.9 describes a log likelihood that should be maximized. In order to turn this into loss function (something that we need to minimize), we'll just flip the sign on Eq. 5.9. The result is the cross-entropy loss L_{CE}:

$$L_{CE}(\hat{y}, y) = -\log p(y|x) \;=\; -\left[y\log\hat{y} + (1-y)\log(1-\hat{y})\right] \tag{5.10}$$

Finally, we can plug in the definition of $\hat{y} = \sigma(w \cdot x) + b$:

$$L_{CE}(w, b) \;=\; -\left[y\log\sigma(w\cdot x+b) + (1-y)\log\left(1-\sigma(w\cdot x+b)\right)\right] \tag{5.11}$$

Why does minimizing this negative log probability do what we want? A perfect classifier would assign probability 1 to the correct outcome (y=1 or y=0) and probability 0 to the incorrect outcome. That means the higher $\hat{y}$ (the closer it is to 1), the better the classifier; the lower $\hat{y}$ is (the closer it is to 0), the worse the classifier. The negative log of this probability is a convenient loss metric since it goes from 0 (negative log of 1, no loss) to infinity (negative log of 0, infinite loss). This loss function also insures that as probability of the correct answer is maximized, the probability of the incorrect answer is minimized; since the two sum to one, any increase in the probability of the correct answer is coming at the expense of the incorrect answer. It's called the cross-entropy loss, because Eq. 5.9 is also the formula for the **cross-entropy** between the true probability distribution y and our estimated distribution $\hat{y}$.

Let's now extend Eq. 5.10 from one example to the whole training set: we'll continue to use the notation that $x^{(i)}$ and $y^{(i)}$ mean the ith training features and training label, respectively. We make the assumption that the training examples are independent:

$$\begin{aligned}
\log p(\text{training labels}) &= \log \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}) && (5.12)\\
&= \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}) && (5.13)\\
&= -\sum_{i=1}^{m} L_{CE}(\hat{y}^{(i)}, y^{(i)}) && (5.14)
\end{aligned}$$

We'll define the cost function for the whole dataset as the average loss for each example:

$$\begin{aligned}
Cost(w,b) &= \frac{1}{m}\sum_{i=1}^{m} L_{CE}(\hat{y}^{(i)}, y^{(i)})\\
&= -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \sigma(w \cdot x^{(i)} + b) + (1-y^{(i)}) \log\left(1-\sigma(w \cdot x^{(i)} + b)\right)
\end{aligned} \tag{5.15}$$

Now we know what we want to minimize; in the next section, we'll see how to find the minimum.

5.4 Gradient Descent

Our goal with gradient descent is to find the optimal weights: minimize the loss function we've defined for the model. In Eq. 5.16 below, we'll explicitly represent the fact that the loss function L is parameterized by the weights, which we'll refer to in machine learning in general as θ (in the case of logistic regression $\theta = w, b$):

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{m}\sum_{i=1}^{m} L_{CE}(y^{(i)}, x^{(i)}; \theta) \tag{5.16}$$

How shall we find the minimum of this (or any) loss function? Gradient descent is a method that finds a minimum of a function by figuring out in which direction (in the space of the parameters θ) the function's slope is rising the most steeply, and moving in the opposite direction. The intuition is that if you are hiking in a canyon and trying to descend most quickly down to the river at the bottom, you might look around yourself 360 degrees, find the direction where the ground is sloping the steepest, and walk downhill in that direction.

convex

For logistic regression, this loss function is conveniently **convex**. A convex function has just one minimum; there are no local minima to get stuck in, so gradient descent starting from any point is guaranteed to find the minimum.

Although the algorithm (and the concept of gradient) are designed for direction *vectors*, let's first consider a visualization of the the case where the parameter of our system, is just a single scalar w, shown in Fig. 5.3.

Given a random initialization of w at some value w_1, and assuming the loss function L happened to have the shape in Fig. 5.3, we need the algorithm to tell us whether at the next iteration, we should move left (making w^2 smaller than w^1) or right (making w^2 bigger than w^1) to reach the minimum.

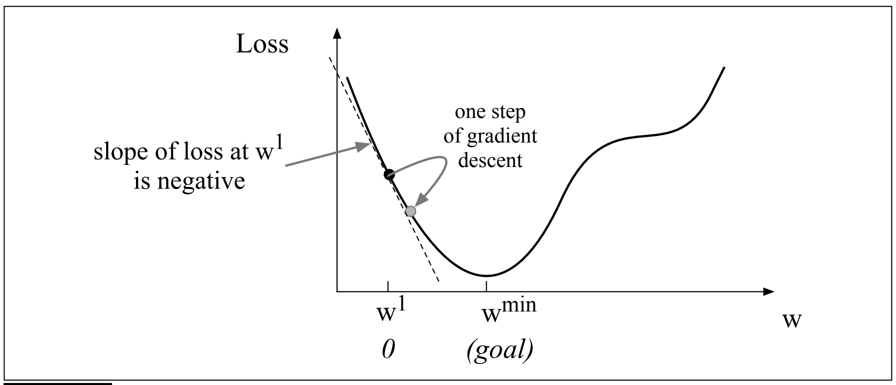


Figure 5.3 The first step in iteratively finding the minimum of this loss function, by moving w in the reverse direction from the slope of the function. Since the slope is negative, we need to move w in a positive direction, to the right. Here superscripts are used for learning steps, so w^1 means the initial value of w (which is 0), w^2 at the second step, and so on.

gradient

The gradient descent algorithm answers this question by finding the **gradient** of the loss function at the current point and moving in the opposite direction. The gradient of a function of many variables is a vector pointing in the direction the greatest increase in a function. The gradient is a multi-variable generalization of the slope, so for a function of one variable like the one in Fig. 5.3, we can informally think of the gradient as the slope. The dotted line in Fig. 5.3 shows the slope of this hypothetical loss function at point $w = w^1$. You can see that the slope of this dotted line is negative. Thus to find the minimum, gradient descent tells us to go in the opposite direction: moving w in a positive direction.

learning rate

The magnitude of the amount to move in gradient descent is the value of the slope $\frac{d}{dw}f(x;w)$ weighted by a **learning rate** η. A higher (faster) learning rate means that we should move w more on each step. The change we make in our parameter is the learning rate times the gradient (or the slope, in our single-variable example):

$$w^{t+1} = w^t - \eta \frac{d}{dw} f(x;w) \tag{5.17}$$

Now let's extend the intuition from a function of one scalar variable w to many

variables, because we don't just want to move left or right, we want to know where in the N-dimensional space (of the N parameters that make up θ) we should move. The **gradient** is just such a vector; it expresses the directional components of the sharpest slope along each of those N dimensions. If we're just imagining two weight dimension (say for one weight w and one bias b), the gradient might be a vector with two orthogonal components, each of which tells us how much the ground slopes in the w dimension and in the b dimension. Fig. 5.4 shows a visualization:

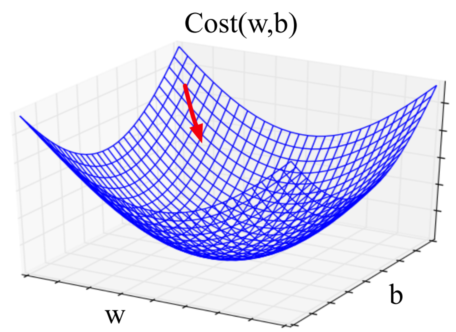


Figure 5.4 Visualization of the gradient vector in two dimensions w and b.

In an actual logistic regression, the parameter vector w is much longer than 1 or 2, since the input feature vector x can be quite long, and we need a weight w_i for each x_i For each dimension/variable w_i in w (plus the bias b), the gradient will have a component that tells us the slope with respect to that variable. Essentially we're asking: "How much would a small change in that variable w_i influence the total loss function L?"

In each dimension w_i, we express the slope as a partial derivative $\frac{\partial}{\partial w_i}$ of the loss function. The gradient is then defined as a vector of these partials. We'll represent $\hat{y}$ as $f(x;\theta)$ to make the dependence on θ more obvious:

$$\nabla_\theta L(f(x;\theta),y)) = \begin{bmatrix} \frac{\partial}{\partial w_1}L(f(x;\theta),y) \\ \frac{\partial}{\partial w_2}L(f(x;\theta),y) \\ \vdots \\ \frac{\partial}{\partial w_n}L(f(x;\theta),y) \end{bmatrix} \tag{5.18}$$

The final equation for updating θ based on the gradient is thus

$$\theta_{t+1} = \theta_t - \eta\nabla L(f(x;\theta),y) \tag{5.19}$$

5.4.1 The Gradient for Logistic Regression

In order to update θ, we need a definition for the gradient $\nabla L(f(x;\theta),y)$. Recall that for logistic regression, the cross-entropy loss function is:

$$L_{CE}(w,b) = -[y\log\sigma(w\cdot x+b)+(1-y)\log(1-\sigma(w\cdot x+b))] \tag{5.20}$$

It turns out that the derivative of this function for one observation vector x is Eq. 5.21 (the interested reader can see Section 5.8 for the derivation of this equation):

$$\frac{\partial L_{CE}(w,b)}{\partial w_j} = [\sigma(w\cdot x+b)-y]x_j \tag{5.21}$$

Note in Eq. 5.21 that the gradient with respect to a single weight w_j represents a very intuitive value: the difference between the true y and our estimated $\hat{y} = \sigma(w \cdot x + b)$ for that observation, multiplied by the corresponding input value x_j.

The loss for a batch of data or an entire dataset is just the average loss over the m examples:

$$Cost(w,b) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \sigma(w \cdot x^{(i)} + b) + (1 - y^{(i)}) \log \left(1 - \sigma(w \cdot x^{(i)} + b)\right) \tag{5.22}$$

And the gradient for multiple data points is the sum of the individual gradients::

$$\frac{\partial Cost(w,b)}{\partial w_j} = \sum_{i=1}^{m} \left[\sigma(w \cdot x^{(i)} + b) - y^{(i)}\right] x_j^{(i)} \tag{5.23}$$

5.4.2 The Stochastic Gradient Descent Algorithm

Stochastic gradient descent is an online algorithm that minimizes the loss function by computing its gradient after each training example, and nudging θ in the right direction (the opposite direction of the gradient). Fig. 5.5 shows the algorithm.

function STOCHASTIC GRADIENT DESCENT($L()$, $f()$, x, y) **returns** θ
  # where: L is the loss function
  #   f is a function parameterized by θ
  #   x is the set of training inputs $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$
  #   y is the set of training outputs (labels) $y^{(1)}, y^{(2)}, \ldots, y^{(n)}$

$\theta \leftarrow 0$
repeat T times
  For each training tuple $(x^{(i)}, y^{(i)})$ (in random order)
  Compute $\hat{y}^{(i)} = f(x^{(i)};\theta)$ # What is our estimated output $\hat{y}$?
  Compute the loss $L(\hat{y}^{(i)}, y^{(i)})$ # How far off is $\hat{y}^{(i)})$ from the true output $y^{(i)}$?
  $g \leftarrow \nabla_\theta L(f(x^{(i)};\theta), y^{(i)})$ # How should we move θ to maximize loss ?
  $\theta \leftarrow \theta - \eta\, g$ # go the other way instead
return θ

Figure 5.5 The stochastic gradient descent algorithm

Stochastic gradient descent is called stochastic because it chooses a single random example at a time, moving the weights so as to improve performance on that single example. That can result in very choppy movements, so it's also common to do **minibatch** gradient descent, which computes the gradient over batches of training instances rather than a single instance.

minibatch

The learning rate η is a parameter that must be adjusted. If it's too high, the learner will take steps that are too large, overshooting the minimum of the loss function. If it's too low, the learner will take steps that are too small, and take too long to get to the minimum. It is most common to begin the learning rate at a higher value, and then slowly decrease it, so that it is a function of the iteration k of training; you will sometimes see the notation η_k to mean the value of the learning rate at iteration k.

5.4.3 Working through an example

Let's walk though a single step of the gradient descent algorithm. We'll use a simplified version of the example in Fig. 5.2 as it sees a single observation x, whose correct value is $y = 1$ (this is a positive review), and with only two features:

$$\begin{aligned} x_1 &= 3 \quad \text{(count of positive lexicon words)} \\ x_2 &= 2 \quad \text{(count of negative lexicon words)} \end{aligned}$$

Let's assume the initial weights and bias in θ^0 are all set to 0, and the initial learning rate η is 0.1:

$$\begin{aligned} w_1 = w_2 = b &= 0 \\ \eta &= 0.1 \end{aligned}$$

The single update step requires that we compute the gradient, multiplied by the learning rate

$$\theta^{t+1} = \theta^t - \eta \nabla_\theta L(f(x^{(i)};\theta), y^{(i)})$$

In our mini example there are three parameters, so the gradient vector has 3 dimensions, for w_1, w_2, and b. We can compute the first gradient as follows:

$$\nabla_{w,b} = \begin{bmatrix} \frac{\partial L_{CE}(w,b)}{\partial w_1} \\ \frac{\partial L_{CE}(w,b)}{\partial w_2} \\ \frac{\partial L_{CE}(w,b)}{\partial b} \end{bmatrix} = \begin{bmatrix} (\sigma(w\cdot x+b)-y)x_1 \\ (\sigma(w\cdot x+b)-y)x_2 \\ \sigma(w\cdot x+b)-y \end{bmatrix} = \begin{bmatrix} (\sigma(0)-1)x_1 \\ (\sigma(0)-1)x_2 \\ \sigma(0)-1 \end{bmatrix} = \begin{bmatrix} -0.5x_1 \\ -0.5x_2 \\ -0.5 \end{bmatrix} = \begin{bmatrix} -1.5 \\ -1.0 \\ -0.5 \end{bmatrix}$$

Now that we have a gradient, we compute the new parameter vector θ^2 by moving θ^1 in the opposite direction from the gradient:

$$\theta^2 = \begin{bmatrix} w_1 \\ w_2 \\ b \end{bmatrix} - \eta \begin{bmatrix} -1.5 \\ -1.0 \\ -0.5 \end{bmatrix} = \begin{bmatrix} .15 \\ .1 \\ .05 \end{bmatrix}$$

So after one step of gradient descent, the weights have shifted to be: $w_1 = .15$, $w_2 = .1$, and $b = .05$.

Note that this observation x happened to be a positive example. We would expect that after seeing more negative examples with high counts of negative words, that the weight w_2 would shift to have a negative value.

5.5 Regularization

Numquam ponenda est pluralitas sine necessitate
'Plurality should never be proposed unless needed'
William of Occam

There is a problem with learning weights that make the model perfectly match the training data. If a feature is perfectly predictive of the outcome because it happens to only occur in one class, it will be assigned a very high weight. The weights for features will attempt to perfectly fit details of the training set, in fact too perfectly, modeling noisy factors that just accidentally correlate with the class. This problem is called **overfitting**. A good model should be able to **generalize** well from the training data to the unseen test set, but a model that overfits will have poor generalization.

overfitting

generalize

regularization

To avoid overfitting, a **regularization** term is added to the objective function in Eq. 5.16, resulting in the following objective:

$$\hat{w} = \underset{w}{\operatorname{argmax}} \sum_{1=1}^{m} \log P(y^{(i)}|x^{(i)}) - \alpha R(w) \tag{5.24}$$

The new component, $R(w)$ is called a regularization term, and is used to penalize large weights. Thus a setting of the weights that matches the training data perfectly, but uses many weights with high values to do so, will be penalized more than a setting that matches the data a little less well, but does so using smaller weights.

L2 regularization

There are two common regularization terms $R(w)$. **L2 regularization** is a quadratic function of the weight values, named because it uses the (square of the) L2 norm of the weight values. The L2 norm, $||W||_2$, is the same as the **Euclidean distance**:

$$R(W) = ||W||_2^2 = \sum_{j=1}^{N} w_j^2 \tag{5.25}$$

The L2 regularized objective function becomes:

$$\hat{w} = \underset{w}{\operatorname{argmax}} \left[\sum_{1=i}^{m} \log P(y^{(i)}|x^{(i)}) \right] - \alpha \sum_{j=1}^{n} w_j^2 \tag{5.26}$$

L1 regularization

L1 regularization is a linear function of the weight values, named after the L1 norm $||W||_1$, the sum of the absolute values of the weights, or **Manhattan distance** (the Manhattan distance is the distance you'd have to walk between two points in a city with a street grid like New York):

$$R(W) = ||W||_1 = \sum_{i=1}^{N} |w_i| \tag{5.27}$$

The L1 regularized objective function becomes:

$$\hat{w} = \underset{w}{\operatorname{argmax}} \left[\sum_{1=i}^{m} \log P(y^{(i)}|x^{(i)}) \right] - \alpha \sum_{j=1}^{n} |w_j| \tag{5.28}$$

These kinds of regularization come from statistics, where L1 regularization is called the **'lasso'** or **lasso regression** (Tibshirani, 1996) and L2 regression is called

ridge regression, and both are commonly used in language processing. L2 regularization is easier to optimize because of its simple derivative (the derivative of w^2 is just $2w$), while L1 regularization is more complex (the derivative of $|w|$ is non-continuous at zero). But where L2 prefers weight vectors with many small weights, L1 prefers sparse solutions with some larger weights but many more weights set to zero. Thus L1 regularization leads to much sparser weight vectors, that is, far fewer features.

Both L1 and L2 regularization have Bayesian interpretations as constraints on the prior of how weights should look. L1 regularization can be viewed as a Laplace prior on the weights. L2 regularization corresponds to assuming that weights are distributed according to a gaussian distribution with mean $\mu = 0$. In a gaussian or normal distribution, the further away a value is from the mean, the lower its probability (scaled by the variance σ). By using a gaussian prior on the weights, we are saying that weights prefer to have the value 0. A gaussian for a weight w_j is

$$\frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(-\frac{(w_j - \mu_j)^2}{2\sigma_j^2}\right) \tag{5.29}$$

If we multiply each weight by a gaussian prior on the weight, we are thus maximizing the following constraint:

$$\hat{w} = \underset{w}{\operatorname{argmax}} \prod_{i=1}^{M} P(y^{(i)}|x^{(i)}) \times \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(-\frac{(w_j - \mu_j)^2}{2\sigma_j^2}\right) \tag{5.30}$$

which in log space, with $\mu = 0$, and assuming $2\sigma^2 = 1$, corresponds to

$$\hat{w} = \underset{w}{\operatorname{argmax}} \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) - \alpha \sum_{j=1}^{n} w_j^2 \tag{5.31}$$

which is in the same form as Eq. 5.26.

5.6 Multinomial logistic regression

Sometimes we need more than two classes. Perhaps we might want to do 3-way sentiment classification (positive, negative, or neutral). Or we could be classifying the part of speech of a word (choosing from 10, 30, or even 50 different parts of speech), or assigning semantic labels like the named entities or semantic relations we will introduce in Chapter 17.

multinominal logistic regression

In such cases we use **multinominal logistic regression**, also called **softmax regression** (or, historically, the **maxent classifier**). In multinominal logistic regression the target y is a variable that ranges over more than two classes; we want to know the probability of y being in each potential class $c \in C$, $p(y = c|x)$.

softmax

The multinominal logistic classifier uses a generalization of the sigmoid, called the **softmax** function, to compute the probability $p(y = c|x)$. The softmax function takes a vector $z = [z_1, z_2, ..., z_k]$ of k arbitrary values and maps them to a probability distribution, with each value in the range (0,1], and all the values summing to 1. Like the sigmoid, it is an exponential function;

For a vector z of dimensionality k, the softmax is defined as:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad 1 \le i \le k \tag{5.32}$$

The softmax of an input vector $z = [z_1, z_2, ..., z_k]$ is thus a vector itself:

$$\text{softmax}(z) = \left[\frac{e^{z_1}}{\sum_{i=1}^{k} e^{z_i}}, \frac{e^{z_2}}{\sum_{i=1}^{k} e^{z_i}}, ..., \frac{e^{z_k}}{\sum_{i=1}^{k} e^{z_i}} \right] \tag{5.33}$$

The denominator $\sum_{i=1}^{k} e^{z_i}$ is used to normalize all the values into probabilities. Thus for example given a vector:

$$z = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

the result softmax(z) is

$$[0.055, 0.090, 0.0067, 0.10, 0.74, 0.010]$$

Again like the sigmoid, the input to the softmax will be the dot product between a weight vector w and an input vector x (plus a bias). But now we'll need separate weight vectors (and bias) for each of the K classes.

$$p(y = c|x) = \frac{e^{w_c \cdot x + b_c}}{\sum_{j=1}^{k} e^{w_j \cdot x + b_j}} \tag{5.34}$$

Like the sigmoid, the softmax has the property of squashing values toward 0 or 1. thus if one of the inputs is larger than the others, will tend to push its probability toward 1, and suppress the probabilities of the smaller inputs.

5.6.1 Features in Multinomial Logistic Regression

For multiclass classification the input features need to be a function of both the observation x and the candidate output class c. Thus instead of the notation x_i, f_i or $f_i(x)$, when we're discussing features we will use the notation $f_i(c,x)$, meaning feature i for a particular class c for a given observation x.

In binary classification, a positive weight on a feature pointed toward y=1 and a negative weight toward y=0... but in multiclass a feature could be evidence for or against an individual class.

Let's look at some sample features for a few NLP tasks to help understand this perhaps unintuitive use of features that are functions of both the observation x and the class c,

Suppose we are doing text classification, and instead of binary classification our task is to assign one of the 3 classes $+$, $-$, or 0 (neutral) to a document. Now a feature related to exclamation marks might have a negative weight for 0 documents, and a positive weight for $+$ or $-$ documents:

Var	Definition	Wt
$f_1(0,x)$	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	−4.5
$f_1(+,x)$	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	2.6
$f_1(0,x)$	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	1.3

5.6.2 Learning in Multinomial Logistic Regression

Multinomial logistic regression has a slightly different loss function than binary logistic regression because it uses the softmax rather than sigmoid classifier, The loss function for a single example x is the sum of the logs of the K output classes:

$$\begin{aligned} L_{CE}(\hat{y},y) &= -\sum_{k=1}^{K} 1\{y=k\}\log p(y=k|x) \\ &= -\sum_{k=1}^{K} 1\{y=k\}\log \frac{e^{w_k\cdot x+b_k}}{\sum_{j=1}^{K} e^{w_j\cdot x+b_j}} \end{aligned} \tag{5.35}$$

This makes use of the function $1\{\}$ which evaluates to 1 if the condition in the brackets is true and to 0 otherwise.

The gradient for a single example turns out to be very similar to the gradient for logistic regression, although we don't show the derivation here. It is the different between the value for the true class k (which is 1) and the probability the classifier outputs for class k, weighted by the value of the input x_k:

$$\begin{aligned} \frac{\partial L_{CE}}{\partial w_k} &= (1\{y=k\} - p(y=k|x))x_k \\ &= \left(1\{y=k\} - \frac{e^{w_k\cdot x+b_k}}{\sum_{j=1}^{K} e^{w_j\cdot x+b_j}}\right)x_k \end{aligned} \tag{5.36}$$

5.7 Interpreting models

interpretable

Often we want to know more than just the correct classification of an observation. We want to know why the classifier made the decision it did. That is, we want our decision to be **interpretable**. Interpretability can be hard to define strictly, but the core idea is that as humans we should know why our algorithms reach the conclusions they do. Because the features to logistic regression are often human-designed, one way to understand a classifier's decision is to understand the role each feature it plays in the decision. Logistic regression can be combined with statistical tests (the likelihood ratio test, or the Wald test); investigating whether a particular feature is significant by one of these tests, or inspecting its magnitude (how large is the weight w associated with the feature?) can help us interpret why the classifier made the decision it makes. This is enormously important for building transparent models.

Furthermore, in addition to its use as a classifier, logistic regression in NLP and many other fields is widely used as an analytic tool for testing hypotheses about the

effect of various explanatory variables (features). In text classification, perhaps we want to know if logically negative words (*no, not, never*) are more likely to be associated with negative sentiment, or if negative reviews of movies are more likely to discuss the cinematography. However, in doing so it's necessary to control for potential confounds: other factors that might influence sentiment (the movie genre, the year it was made, perhaps the length of the review in words). Or we might be studying the relationship between NLP-extracted linguistic features and non-linguistic outcomes (hospital readmissions, political outcomes, or product sales), but need to control for confounds (the age of the patient, the county of voting, the brand of the product). In such cases, logistic regression allows us to test whether some feature is associated with some outcome above and beyond the effect of other features.

5.8 Advanced: Deriving the Gradient Equation

In this section we give the derivation of the gradient of the cross-entropy loss function L_{CE} for logistic regression. Let's start with some quick calculus refreshers. First, the derivative of $\ln(x)$:

$$\frac{d}{dx}\ln(x) = \frac{1}{x} \tag{5.37}$$

Second, the (very elegant) derivative of the sigmoid:

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-z) \tag{5.38}$$

chain rule

Finally, the **chain rule** of derivatives. Suppose we are computing the derivative of a composite function $f(x) = u(v(x))$. The derivative of $f(x)$ is the derivative of $u(x)$ with respect to $v(x)$ times the derivative of $v(x)$ with respect to x:

$$\frac{df}{dx} = \frac{du}{dv}\cdot\frac{dv}{dx} \tag{5.39}$$

First, we want to know the derivative of the loss function with respect to a single weight w_j (we'll need to compute it for each weight, and for the bias):

$$\begin{aligned}\frac{\partial LL(w,b)}{\partial w_j} &= \frac{\partial}{\partial w_j} - [y\log\sigma(w\cdot x+b)+(1-y)\log(1-\sigma(w\cdot x+b))] \\ &= -\left[\frac{\partial}{\partial w_j}y\log\sigma(w\cdot x+b)+\frac{\partial}{\partial w_j}(1-y)\log[1-\sigma(w\cdot x+b)]\right]\end{aligned} \tag{5.40}$$

Next, using the chain rule, and relying on the derivative of log:

$$\frac{\partial LL(w,b)}{\partial w_j} = -\frac{y}{\sigma(w\cdot x+b)}\frac{\partial}{\partial w_j}\sigma(w\cdot x+b)+\frac{1-y}{1-\sigma(w\cdot x+b)}\frac{\partial}{\partial w_j}1-\sigma(w\cdot x+b) \tag{5.41}$$

Rearranging terms:

$$\frac{\partial LL(w,b)}{\partial w_j} = -\left[\frac{y}{\sigma(w\cdot x+b)}-\frac{1-y}{1-\sigma(w\cdot x+b)}\right]\frac{\partial}{\partial w_j}\sigma(w\cdot x+b) \tag{5.42}$$

And now plugging in the derivative of the sigmoid, and using the chain rule one more time, we end up with Eq. 5.43:

$$\begin{aligned}
\frac{\partial LL(w,b)}{\partial w_j} &= -\left[\frac{y-\sigma(w\cdot x+b)}{\sigma(w\cdot x+b)[1-\sigma(w\cdot x+b)]}\right]\sigma(w\cdot x+b)[1-\sigma(w\cdot x+b)]\frac{\partial(w\cdot x+b)}{\partial w_j}\\
&= -\left[\frac{y-\sigma(w\cdot x+b)}{\sigma(w\cdot x+b)[1-\sigma(w\cdot x+b)]}\right]\sigma(w\cdot x+b)[1-\sigma(w\cdot x+b)]x_j\\
&= -[y-\sigma(w\cdot x+b)]x_j\\
&= [\sigma(w\cdot x+b)-y]x_j
\end{aligned} \tag{5.43}$$

5.9 Summary

This chapter introduced the **logistic regression** model of **classification**.

- Logistic regression is a supervised machine learning classifier that extracts real-valued features from the input, multiplies each by a weight, sums them, and passes the sum through a **sigmoid** function to generate a probability. A threshold is used to make a decision.
- Logistic regression can be used with two classes (e.g., positive and negative sentiment) or with multiple classes (**multinomial logistic regression**, for example for n-ary text classification, part-of-speech labeling, etc.).
- Multinomial logistic regression uses the **softmax** function to compute probabilities.
- The weights (vector w and bias b) are learned from a labeled training set via a loss function, such as the **cross-entropy loss**, that must be minimized.
- Minimizing this loss function is a **convex optimization** problem, and iterative algorithms like **gradient descent** are used to find the optimal weights.
- **Regularization** is used to avoid overfitting.
- Logistic regression is also one of the most useful analytic tools, because of its ability to transparently study the importance of individual features.

Bibliographical and Historical Notes

Logistic regression was developed in the field of statistics, where it was used for the analysis of binary data by the 1960s, and was particularly common in medicine (Cox, 1969). Starting in the late 1970s it became widely used in linguistics as one of the formal foundations of the study of linguistic variation (Sankoff and Labov, 1979).

Nonetheless, logistic regression didn't become common in natural language processing until the 1990s, when it seems to have appeared simultaneously from two directions. The first source was the neighboring fields of information retrieval and speech processing, both of which had made use of regression, and both of which lent many other statistical techniques to NLP. Indeed a very early use of logistic regression for document routing was one of the first NLP applications to use (LSI) embeddings as word representations (Schütze et al., 1995).

maximum entropy

At the same time in the early 1990s logistic regression was developed and applied to NLP at IBM Research under the name **maximum entropy** modeling or

maxent (Berger et al., 1996), seemingly independent of the statistical literature. Under that name it was applied to language modeling (Rosenfeld, 1996), part-of-speech tagging ((Ratnaparkhi, 1996)), parsing (Ratnaparkhi, 1997), and text classification (Nigam et al., 1999).

More on classification can be found in machine learning textbooks (Hastie et al. 2001, Witten and Frank 2005, Bishop 2006, Murphy 2012).

Exercises

CHAPTER

6 Vector Semantics

The asphalt that Los Angeles is famous for occurs mainly on its freeways. But in the middle of the city is another patch of asphalt, the La Brea tar pits, and this asphalt preserves millions of fossil bones from the last of the Ice Ages of the Pleistocene Epoch. One of these fossils is the *Smilodon*, or sabre-toothed tiger, instantly recognizable by its long canines. Five million years ago or so, a completely different sabre-tooth tiger called *Thylacosmilus* lived in Argentina and other parts of South America. Thylacosmilus was a marsupial whereas Smilodon was a placental mammal, but Thylacosmilus had the same long upper canines and, like Smilodon, had a protective bone flange on the lower jaw. The similarity of these two mammals is one of many examples of parallel or convergent evolution, in which particular contexts or environments lead to the evolution of very similar structures in different species (Gould, 1980).

The role of context is also important in the similarity of a less biological kind of organism: the word. Words that occur in *similar contexts* tend to have *similar meanings*. This link between similarity in how words are distributed and similarity in what they mean is called the **distributional hypothesis**. The hypothesis was first formulated in the 1950s by linguists like Joos (1950), Harris (1954), and Firth (1957), who noticed that words which are synonyms (like *oculist* and *eye-doctor*) tended to occur in the same environment (e.g., near words like *eye* or *examined*) with the amount of meaning difference between two words "corresponding roughly to the amount of difference in their environments" (Harris, 1954, 157).

distributional hypothesis

In this chapter we introduce a model known as **vector semantics**, which instantiates this linguistic hypothesis by learning representations of the meaning of words directly from their distributions in texts. These representations are used in every natural language processing application that makes use of meaning. These word representations are also the first example we will see in the book of **representation learning**, automatically learning useful representations of the input text. Finding such **unsupervised** ways to learn representations of the input, instead of creating representations by hand via **feature engineering**, is an important focus of recent NLP research (Bengio et al., 2013).

vector semantics

representation learning

We'll begin, however, by introducing some basic principles of word meaning, which will motivate the vector semantic models of this chapter as well as extensions that we'll return to in Chapter 19, Chapter 20, and Chapter 18.

6.1 Lexical Semantics

How should we represent the meaning of a word? In the N-gram models we saw in Chapter 3, and in many traditional NLP applications, our only representation of a word is as a string of letters, or perhaps as an index in a vocabulary list. This representation is not that different from a tradition in philosophy, perhaps you've seen it in introductory logic classes, in which the meaning of words is often represented by just spelling the word with small capital letters; representing the meaning of "dog" as DOG, and "cat" as CAT).

Representing the meaning of a word by capitalizing it is a pretty unsatisfactory model. You might have seen the old philosophy joke:

Q: What's the meaning of life?
A: LIFE

Surely we can do better than this! After all, we'll want a model of word meaning to do all sorts of things for us. It should tell us that some words have similar meanings (*cat* is similar to *dog*), other words are antonyms (*cold* is the opposite of *hot*). It should know that some words have positive connotations (*happy*) while others have negative connotations (*sad*). It should represent the fact that the meanings of *buy*, *sell*, and *pay* offer differing perspectives on the same underlying purchasing event (If I buy something from you, you've probably sold it to me, and I likely paid you).

More generally, a model of word meaning should allow us to draw useful inferences that will help us solve meaning-related tasks like question-answering, summarization, paraphrase or plagiarism detection, and dialogue.

In this section we summarize some of these desiderata, drawing on results in the linguistic study of word meaning, which is called **lexical semantics**.

lexical semantics

Lemmas and Senses Let's start by looking at how one word (we'll choose *mouse*) might be defined in a dictionary: [1]

```
mouse (N)
1.  any of numerous small rodents...
2.  a hand-operated device that controls a cursor...
```

lemma

citation form

Here the form *mouse* is the **lemma**, also called the **citation form**. The form *mouse* would also be the lemma for the word *mice*; dictionaries don't have separate definitions for inflected forms like *mice*. Similarly *sing* is the lemma for *sing*, *sang*, *sung*. In many languages the infinitive form is used as the lemma for the verb, so Spanish *dormir* "to sleep" is the lemma for *duermes* "you sleep". The specific forms *sung* or *carpets* or *sing* or *duermes* are called **wordforms**.

wordform

As the example above shows, each lemma can have multiple meanings; the lemma *mouse* can refer to the rodent or the cursor control device. We call each of these aspects of the meaning of *mouse* a **word sense**. The fact that lemmas can be **homonymous** (have multiple senses) can make interpretation difficult (is someone who types "mouse info" to a search engine looking for a pet or a tool?). Chapter 19 will discuss the problem of homonymy, and introduce *word sense disambiguation*, the task of determining which sense of a word is being used in a particular context.

Relationships between words or senses One important component of word meaning is the relationship between word senses. For example when one word has a sense

[1] This example shortened from the online dictionary WordNet, discussed in Chapter 19.

synonym

whose meaning is identical to a sense of another word, or nearly identical, we say the two senses of those two words are **synonyms**. Synonyms include such pairs as

couch/sofa vomit/throw up filbert/hazelnut car/automobile

propositional meaning

A more formal definition of synonymy (between words rather than senses) is that two words are synonymous if they are substitutable one for the other in any sentence without changing the *truth conditions* of the sentence, the situations in which the sentence would be true. We often say in this case that the two words have the same **propositional meaning**.

principle of contrast

While substitutions between some pairs of words like *car* / *automobile* or *water* / H_2O are truth preserving, the words are still not identical in meaning. Indeed, probably no two words are absolutely identical in meaning. One of the fundamental tenets of semantics, called the **principle of contrast** (Bréal 1897, ?, Clark 1987), is the assumption that a difference in linguistic form is always associated with at least some difference in meaning. For example, the word H_2O is used in scientific contexts and would be inappropriate in a hiking guide—*water* would be more appropriate— and this difference in genre is part of the meaning of the word. In practice, the word *synonym* is therefore commonly used to describe a relationship of approximate or rough synonymy.

antonym

Where synonyms are words with identical or similar meanings, **Antonyms** are words with an opposite meaning, like:

long/short big/little fast/slow cold/hot dark/light
rise/fall up/down in/out

reversives

Two senses can be antonyms if they define a binary opposition or are at opposite ends of some scale. This is the case for *long/short*, *fast/slow*, or *big/little*, which are at opposite ends of the *length* or *size* scale. Another group of antonyms, **reversives**, describe change or movement in opposite directions, such as *rise/fall* or *up/down*.

Antonyms thus differ completely with respect to one aspect of their meaning—their position on a scale or their direction—but are otherwise very similar, sharing almost all other aspects of meaning. Thus, automatically distinguishing synonyms from antonyms can be difficult.

Word Similarity: While words don't have many synonyms, most words do have lots of *similar* words. *Cat* is not a synonym of *dog*, but *cats* and *dogs* are certainly similar words. In moving from synonymy to similarity, it will be useful to shift from talking about relations between word senses (like synonymy) to relations between words (like similarity). Dealing with words avoids having to commit to a particular representation of word senses, which will turn out to simplify our task.

similarity

The notion of word **similarity** is very useful in larger semantic tasks. For example knowing how similar two words are is helpful if we are trying to decide if two phrases or sentences mean similar things. Phrase or sentence similarity is useful in such natural language understanding tasks as question answering, paraphrasing, and summarization.

One way of getting values for word similarity is to ask humans to judge how similar one word is to another. A number of datasets have resulted from such experiments. For example the SimLex-999 dataset (Hill et al., 2015) gives values on a scale from 0 to 10, like the examples below, which range from near-synonyms (*vanish*, *disappear*) to pairs that scarcely seem to have anything in common (*hole*, *agreement*):

vanish	disappear	9.8
behave	obey	7.3
belief	impression	5.95
muscle	bone	3.65
modest	flexible	0.98
hole	agreement	0.3

relatedness
association

Word Relatedness: The meaning of two words can be related in ways others than similarity. One such class of connections is called word **relatedness** (Budanitsky and Hirst, 2006), also traditionally called word **association** in psychology.

Consider the meanings of the words *coffee* and *cup*; Coffee is not similar to cup; they share practically no features (coffee is a plant or a beverage, while a cup is an manufactured object with a particular shape).

But coffee and cup are clearly related; they are associated in the world by commonly co-participating in a shared event (the event of drinking coffee out of a cup). Similarly the nouns *scalpel* and *surgeon* are not similar but are related eventively (a surgeon tends to make use of a scalpel).

semantic field

One common kind of relatedness between words is if they belong to the same **semantic field**. A semantic field is a set of words which cover a particular semantic domain and bear structured relations with each other.

For example, words might be related by being in the semantic field of hospitals (*surgeon*, *scalpel*, *nurse*, *anaesthetic*, *hospital*), restaurants (*waiter*, *menu*, *plate*, *food*, *menu*) *chef*), or houses (*door*, *roof*, *kitchen*, *family*, *bed*).

topic models

Semantic fields are also related to **topic models**, like **Latent Dirichlet Allocation**, **LDA**, which apply unsupervised learning on large sets of texts to induce sets of associated words from text. Semantic fields and topic models are a very useful tool for discovering topical structure in documents.

semantic frame

Semantic Frames and Roles: Closely related to semantic fields is the idea of a **semantic frame**. A semantic frame is a set of words that denote perspectives or participants in a particular type of event. A commercial transaction, for example, is a kind of event in which one entity trades money to another entity in return for some good or service, after which the good changes hands or perhaps the service is performed. This event can be encoded lexically by using verbs like *buy* (the event from the perspective of the buyer) *sell* (from the perspective of the seller), *pay* (focusing on the monetary aspect), or nouns like *buyer*. Frames have semantic roles (like *buyer*, *seller*, *goods*, *money*), and words in a sentence can take on these roles.S

Knowing that *buy* and *sell* have this relation makes it possible for a system to know that a sentence like *Sam bought the book from Ling* could be paraphrased as *Ling sold the book to Sam*, and that Sam has the role of the *buyer* in the frame and Ling the *seller*. Being able to recognize such paraphrases is important for question answering, and can help in shifting perspective for machine translation.

hyponym
hypernym
superordinate

Taxonomic Relations: Another way word senses can be related is taxonomically. A word (or sense) is a **hyponym** of another word or sense if the first is more specific, denoting a subclass of the other. For example, *car* is a hyponym of *vehicle*; *dog* is a hyponym of *animal*, and *mango* is a hyponym of *fruit*. Conversely, we say that *vehicle* is a **hypernym** of *car*, and *animal* is a hypernym of *dog*. It is unfortunate that the two words (hypernym and hyponym) are very similar and hence easily confused; for this reason, the word **superordinate** is often used instead of **hypernym**.

Superordinate	vehicle	fruit	furniture	mammal
Subordinate	car	mango	chair	dog

We can define hypernymy more formally by saying that the class denoted by the superordinate extensionally includes the class denoted by the hyponym. Thus, the class of animals includes as members all dogs, and the class of moving actions includes all walking actions. Hypernymy can also be defined in terms of **entailment**. Under this definition, a sense A is a hyponym of a sense B if everything that is A is also B, and hence being an A entails being a B, or $\forall x \; A(x) \Rightarrow B(x)$. Hyponymy/hypernymy is usually a transitive relation; if A is a hyponym of B and B is a hyponym of C, then A is a hyponym of C. Another name for the hypernym/hyponym structure

IS-A

is the **IS-A** hierarchy, in which we say A IS-A B, or B **subsumes A**.

Hypernymy is useful for tasks like textual entailment or question answering; knowing that *leukemia* is a type of *cancer*, for example, would certainly be useful in answering questions about leukemia.

connotations

Connotation: Finally, words have *affective meanings* or **connotations**. The word *connotation* has different meanings in different fields, but here we use it to mean the aspects of a word's meaning that are related to a writer or reader's emotions, sentiment, opinions, or evaluations. For example some words have positive connotations (*happy*) while others have negative connotations (*sad*). Some words describe positive evaluation (*great*, *love*) and others negative evaluation (*terrible*, *hate*). Pos-

sentiment

itive or negative evaluation expressed through language is called **sentiment**, as we saw in Chapter 4, and word sentiment plays a role in important tasks like sentiment analysis, stance detection, and many aspects of natural language processing to the language of politics and consumer reviews.

Early work on affective meaning (Osgood et al., 1957) found that words varied along three important dimensions of affective meaning. These are now generally called *valence*, *arousal*, and *dominance*, defined as follows:

valence: the pleasantness of the stimulus

arousal: the intensity of emotion provoked by the stimulus

dominance: the degree of control exerted by the stimulus

Thus words like *happy* or *satisfied* are high on valence, while *unhappy* or *annoyed* are low on valence. *Excited* or *frenzied* are high on arousal, while *relaxed* or *calm are low on arousal.* Important *or* controlling *are high on dominance, while* awed *or* influenced *are low on dominance*. Each word is thus represented by three numbers, corresponding to its value on each of the three dimensions, like the examples below:

	Valence	Arousal	Dominance
courageous	8.05	5.5	7.38
music	7.67	5.57	6.5
heartbreak	2.45	5.65	3.58
cub	6.71	3.95	4.24
life	6.68	5.59	5.89

Osgood et al. (1957) noticed that in using these 3 numbers to represent the meaning of a word, the model was representing each word as a point in a three-dimensional space, a vector whose three dimensions corresponded to the word's rating on the three scales. This revolutionary idea that word meaning word could be represented as a point in space (e.g., that part of the meaning of *heartbreak* can be represented as the point $[2.45, 5.65, 3.58]$) was the first expression of the vector semantics models that we introduce next.

6.2 Vector Semantics

How can we build a computational model that successfully deals with the different aspects of word meaning we saw in the previous section (word senses, word similarity and relatedness, lexical fields and frames, connotation)?

vector semantics

A perfect model that completely deals with each of these aspects of word meaning turns out to be elusive. But the current best model, called **vector semantics**, draws its inspiration from linguistic and philosophical work of the 1950's.

During that period, the philosopher Ludwig Wittgenstein, skeptical of the possibility of building a completely formal theory of meaning definitions for each word, suggested instead that "the meaning of a word is its use in the language" (Wittgenstein, 1953, PI 43). That is, instead of using some logical language to define each word, we should define words by some representation of how the word was used by actual people in speaking and understanding.

Linguists of the period like Joos (1950), Harris (1954), and Firth (1957) (the linguistic distributionalists), came up with a specific idea for realizing Wittgenstein's intuition: define a word by the environment or distribution it occurs in in language use. A word's distribution is the set of contexts in which it occurs, the neighboring words or grammatical environments. The idea is that two words that occur in very similar distributions (that occur together with very similar words) are likely to have the same meaning.

Let's see an example illustrating this distributionalist approach. Suppose you didn't know what the Cantonese word *ongchoi* meant, but you do see it in the following sentences or contexts:

(6.1) Ongchoi is delicious sauteed with garlic.

(6.2) Ongchoi is superb over rice.

(6.3) ...ongchoi leaves with salty sauces...

And furthermore let's suppose that you had seen many of these context words occurring in contexts like:

(6.4) ...spinach sauteed with garlic over rice...

(6.5) ...chard stems and leaves are delicious...

(6.6) ...collard greens and other salty leafy greens

The fact that *ongchoi* occurs with words like *rice* and *garlic* and *delicious* and *salty*, as do words like *spinach*, *chard*, and *collard greens* might suggest to the reader that ongchoi is a leafy green similar to these other leafy greens.[2]

We can do the same thing computationally by just counting words in the context of *ongchoi*; we'll tend to see words like *sauteed* and *eaten* and *garlic*. The fact that these words and other similar context words also occur around the word *spinach* or *collard greens* can help us discover the similarity between these words and *ongchoi*.

Vector semantics thus combines two intuitions: the distributionalist intuition (defining a word by counting what other words occur in its environment), and the vector intuition of of Osgood et al. (1957) we saw in the last section on connotation: defining the meaning of a word w as a vector, a list of numbers, a point in N-dimensional space. There are various versions of vector semantics, each defining the numbers in the vector somewhat differently, but in each case the numbers are based in some way on counts of neighboring words.

[2] It's in fact *Ipomoea aquatica*, a relative of morning glory sometimes called *water spinach* in English.

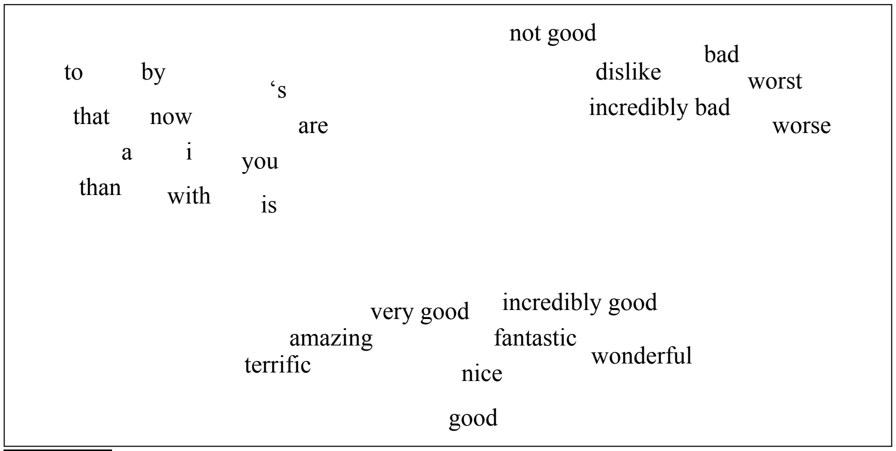


Figure 6.1 A two-dimensional (t-SNE) projection of embeddings for some words and phrases, showing that words with similar meanings are nearby in space. The original 60-dimensional embeddings were trained for a sentiment analysis task. Simplified from Li et al. (2015).

The idea of vector semantics is thus to represent a word as a point in some multi-dimensional semantic space. Vectors for representing words are generally called **embeddings**, because the word is embedded in a particular vector space. Fig. 6.1 displays a visualization of embeddings that were learned for a sentiment analysis task, showing the location of some selected words projected down from the original 60-dimensional space into a two dimensional space.

embeddings

Notice that positive and negative words seem to be located in distinct portions of the space (and different also from the neutral function words). This suggests one of the great advantages of vector semantics: it offers a fine-grained model of meaning that lets us also implement word similarity (and phrase similarity). For example, the sentiment analysis classifier we saw in Chapter 4 only works if enough of the important sentimental words that appear in the test set also appeared in the training set. But if words were represented as embeddings, we could assign sentiment as long as words with *similar meanings* as the test set words occurred in the training set. Vector semantic models are also extremely practical because they can be learned automatically from text without any complex labeling or supervision.

As a result of these advantages, vector models of meaning are now the standard way to represent the meaning of words in NLP. In this chapter we'll introduce the two most commonly used models. . First is the **tf-idf** model, often used a a baseline, in which the meaning of a word is defined by a simple function of the counts of nearby words. We will see that this method results in very long vectors that are **sparse**, i.e. contain mostly zeros (since most words simply never occur in the context of others).

Then we'll introduce the **word2vec** model, one of a family of models that are ways of constructing short, **dense** vectors that have useful semantic properties.

We'll also introduce the **cosine**, the standard way to use embeddings (vectors) to compute functions like *semantic similarity*, the similarity between two words, two sentences, or two documents, an important tool in practical applications like question answering, summarization, or automatic essay grading.

6.3 Words and Vectors

Vector or distributional models of meaning are generally based on a **co-occurrence matrix**, a way of representing how often words co-occur. This matrix can be constructed in various ways; let's s begin by looking at one such co-occurrence matrix, a term-document matrix.

6.3.1 Vectors and documents

term-document matrix

In a **term-document matrix**, each row represents a word in the vocabulary and each column represents a document from some collection of documents. Fig. 6.2 shows a small selection from a term-document matrix showing the occurrence of four words in four plays by Shakespeare. Each cell in this matrix represents the number of times a particular word (defined by the row) occurs in a particular document (defined by the column). Thus *clown* appeared 117 times in *Twelfth Night*.

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	1	8	15
soldier	2	2	12	36
fool	37	58	1	5
clown	5	117	0	0

Figure 6.2 The term-document matrix for four words in four Shakespeare plays. Each cell contains the number of times the (row) word occurs in the (column) document.

vector space model

The term-document matrix of Fig. 6.2 was first defined as part of the **vector space model** of information retrieval (Salton, 1971). In this model, a document is represented as a count vector, a column in Fig. 6.3.

vector

To review some basic linear algebra, a **vector** is, at heart, just a list or array of numbers. So *As You Like It* is represented as the list [1,2,37,5] and *Julius Caesar* is represented as the list [8,12,1,0]. A **vector space** is a collection of vectors, characterized by their **dimension**. In the example in Fig. 6.3, the vectors are of dimension 4, just so they fit on the page; in real term-document matrices, the vectors representing each document would have dimensionality $|V|$, the vocabulary size.

vector space

dimension

The ordering of the numbers in a vector space is not arbitrary; each position indicates a meaningful dimension on which the documents can vary. Thus the first dimension for both these vectors corresponds to the number of times the word *battle* occurs, and we can compare each dimension, noting for example that the vectors for *As You Like It* and *Twelfth Night* have the same value 1 for the first dimension.

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	1	8	15
soldier	2	2	12	36
fool	37	58	1	5
clown	5	117	0	0

Figure 6.3 The term-document matrix for four words in four Shakespeare plays. The red boxes show that each document is represented as a column vector of length four.

We can think of the vector for a document as identifying a point in $|V|$-dimensional space; thus the documents in Fig. 6.3 are points in 4-dimensional space. Since 4-dimensional spaces are hard to draw in textbooks, Fig. 6.4 shows a visualization in two dimensions; we've arbitrarily chosen the dimensions corresponding to the words *battle* and *fool*.

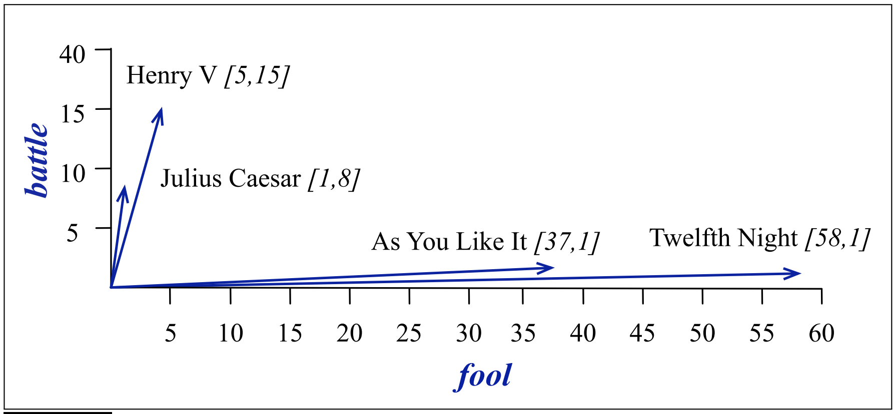


Figure 6.4 A spatial visualization of the document vectors for the four Shakespeare play documents, showing just two of the dimensions, corresponding to the words *battle* and *fool*. The comedies have high values for the *fool* dimension and low values for the *battle* dimension.

Term-document matrices were originally defined as a means of finding similar documents for the task of document **information retrieval**. Two documents that are similar will tend to have similar words, and if two documents have similar words their column vectors will tend to be similar. The vectors for the comedies *As You like It* [1,2,37,5] and *Twelfth Night* [1,2,58,117] look a lot more like each other (more fools and clowns than soldiers and battles) than they do like *Julius Caesar* [8,12,1,0] or *Henry V* [15,36,5,0]. We can see the intuition with the raw numbers; in the first dimension (battle) the comedies have low numbers and the others have high numbers, and we can see it visually in Fig. 6.4; we'll see very shortly how to quantify this intuition more formally.

A real term-document matrix, of course, wouldn't just have 4 rows and columns, let alone 2. More generally, the term-document matrix X has $|V|$ rows (one for each word type in the vocabulary) and D columns (one for each document in the collection); as we'll see, vocabulary sizes are generally at least in the tens of thousands, and the number of documents can be enormous (think about all the pages on the web).

information retrieval

Information retrieval (IR) is the task of finding the document d from the D documents in some collection that best matches a query q. For IR we'll therefore also represent a query by a vector, also of length $|V|$, and we'll need a way to compare two vectors to find how similar they are. (Doing IR will also require efficient ways to store and manipulate these vectors, which is accomplished by making use of the convenient fact that these vectors are sparse, i.e., mostly zeros).

Later in the chapter we'll introduce some of the components of this vector comparison process: the tf-idf term weighting, and the cosine similarity metric.

6.3.2 Words as vectors

We've seen that documents can be represented as vectors in a vector space. But vector semantics can also be used to represent the meaning of *words*, by associating each word with a vector.

row vector

The word vector is now a **row vector** rather than a column vector, and hence the dimensions of the vector are different. The four dimensions of the vector for *fool*, [37,58,1,5], correspond to the four Shakespeare plays. The same four dimensions are used to form the vectors for the other 3 words: *clown*, [5, 117, 0, 0]; *bat-*

tle, [1,1,8,15]; and *soldier* [2,2,12,36]. Each entry in the vector thus represents the counts of the word's occurrence in the document corresponding to that dimension.

For documents, we saw that similar documents had similar vectors, because similar documents tend to have similar words. This same principle applies to words: similar words have similar vectors because they tend to occur in similar documents. The term-document matrix thus lets us represent the meaning of a word by the documents it tends to occur in.

term-term matrix
word-word matrix

However, it is most common to use a different kind of context for the dimensions of a word's vector representation. Rather than the term-document matrix we use the **term-term matrix**, more commonly called the **word-word matrix** or the **term-context matrix**, in which the columns are labeled by words rather than documents. This matrix is thus of dimensionality $|V| \times |V|$ and each cell records the number of times the row (target) word and the column (context) word co-occur in some context in some training corpus. The context could be the document, in which case the cell represents the number of times the two words appear in the same document. It is most common, however, to use smaller contexts, generally a window around the word, for example of 4 words to the left and 4 words to the right, in which case the cell represents the number of times (in some training corpus) the column word occurs in such a ± 4 word window around the row word.

For example here are 7-word windows surrounding four sample words from the Brown corpus (just one example of each word):

sugar, a sliced lemon, a tablespoonful of	**apricot**	jam, a pinch each of,
their enjoyment. Cautiously she sampled her first	**pineapple**	and another fruit whose taste she likened
well suited to programming on the digital	**computer**.	In finding the optimal R-stage policy from
for the purpose of gathering data and	**information**	necessary for the study authorized in the

For each word we collect the counts (from the windows around each occurrence) of the occurrences of context words. Fig. 6.5 shows a selection from the word-word co-occurrence matrix computed from the Brown corpus for these four words.

	aardvark	...	**computer**	**data**	**pinch**	**result**	**sugar**	...
apricot	0	...	0	0	1	0	1	
pineapple	0	...	0	0	1	0	1	
digital	0	...	2	1	0	1	0	
information	0	...	1	6	0	4	0	

Figure 6.5 Co-occurrence vectors for four words, computed from the Brown corpus, showing only six of the dimensions (hand-picked for pedagogical purposes). The vector for the word *digital* is outlined in red. Note that a real vector would have vastly more dimensions and thus be much sparser.

Note in Fig. 6.5 that the two words *apricot* and *pineapple* are more similar to each other (both *pinch* and *sugar* tend to occur in their window) than they are to other words like *digital*; conversely, *digital* and *information* are more similar to each other than, say, to *apricot*. Fig. 6.6 shows a spatial visualization.

Note that $|V|$, the length of the vector, is generally the size of the vocabulary, usually between 10,000 and 50,000 words (using the most frequent words in the training corpus; keeping words after about the most frequent 50,000 or so is generally not helpful). But of course since most of these numbers are zero these are **sparse** vector representations, and there are efficient algorithms for storing and computing with sparse matrices.

Now that we have some intuitions, let's move on to examine the details of computing word similarity. Afterwards we'll discuss the tf-idf method of weighting cells.

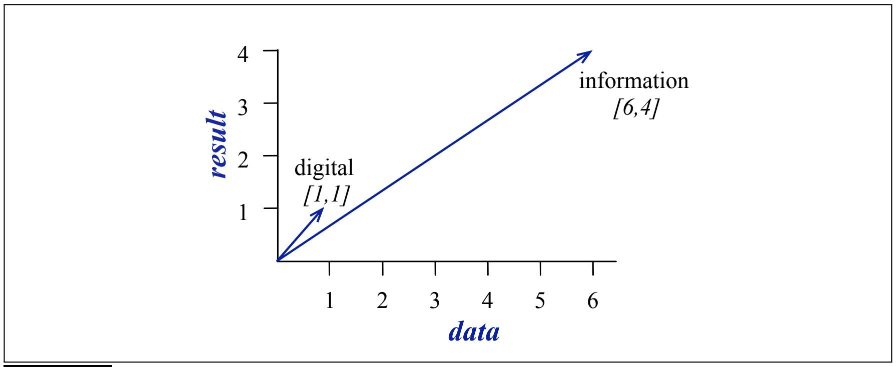


Figure 6.6 A spatial visualization of word vectors for *digital* and *information*, showing just two of the dimensions, corresponding to the words *data* and *result*.

6.4 Cosine for measuring similarity

To define similarity between two target words v and w, we need a measure for taking two such vectors and giving a measure of vector similarity. By far the most common similarity metric is the **cosine** of the angle between the vectors.

dot product
inner product

The cosine—like most measures for vector similarity used in NLP—is based on the **dot product** operator from linear algebra, also called the **inner product**:

$$\text{dot-product}(\vec{v},\vec{w}) = \vec{v}\cdot\vec{w} = \sum_{i=1}^{N} v_i w_i = v_1 w_1 + v_2 w_2 + ... + v_N w_N \tag{6.7}$$

As we will see, most metrics for similarity between vectors are based on the dot product. The dot product acts as a similarity metric because it will tend to be high just when the two vectors have large values in the same dimensions. Alternatively, vectors that have zeros in different dimensions—orthogonal vectors—will have a dot product of 0, representing their strong dissimilarity.

vector length

This raw dot-product, however, has a problem as a similarity metric: it favors **long** vectors. The **vector length** is defined as

$$|\vec{v}| = \sqrt{\sum_{i=1}^{N} v_i^2} \tag{6.8}$$

The dot product is higher if a vector is longer, with higher values in each dimension. More frequent words have longer vectors, since they tend to co-occur with more words and have higher co-occurrence values with each of them. The raw dot product thus will be higher for frequent words. But this is a problem; we'd like a similarity metric that tells us how similar two words are regardless of their frequency.

The simplest way to modify the dot product to normalize for the vector length is to divide the dot product by the lengths of each of the two vectors. This **normalized dot product** turns out to be the same as the cosine of the angle between the two vectors, following from the definition of the dot product between two vectors $\vec{a}$ and $\vec{b}$:

$$\begin{aligned} \vec{a} \cdot \vec{b} &= |\vec{a}||\vec{b}| \cos\theta \\ \frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|} &= \cos\theta \end{aligned} \tag{6.9}$$

cosine

The **cosine** similarity metric between two vectors $\vec{v}$ and $\vec{w}$ thus can be computed as:

$$\text{cosine}(\vec{v}, \vec{w}) = \frac{\vec{v} \cdot \vec{w}}{|\vec{v}||\vec{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2} \sqrt{\sum_{i=1}^{N} w_i^2}} \tag{6.10}$$

unit vector

For some applications we pre-normalize each vector, by dividing it by its length, creating a **unit vector** of length 1. Thus we could compute a unit vector from $\vec{a}$ by dividing it by $|\vec{a}|$. For unit vectors, the dot product is the same as the cosine.

The cosine value ranges from 1 for vectors pointing in the same direction, through 0 for vectors that are orthogonal, to -1 for vectors pointing in opposite directions. But raw frequency values are non-negative, so the cosine for these vectors ranges from 0–1.

Let's see how the cosine computes which of the words *apricot* or *digital* is closer in meaning to *information*, just using raw counts from the following simplified table:

	large	**data**	**computer**
apricot	2	0	0
digital	0	1	2
information	1	6	1

$$\begin{aligned} \cos(\text{apricot}, \text{information}) &= \frac{2+0+0}{\sqrt{4+0+0}\sqrt{1+36+1}} = \frac{2}{2\sqrt{38}} = .16 \\ \cos(\text{digital}, \text{information}) &= \frac{0+6+2}{\sqrt{0+1+4}\sqrt{1+36+1}} = \frac{8}{\sqrt{38}\sqrt{5}} = .58 \end{aligned} \tag{6.11}$$

The model decides that *information* is closer to *digital* than it is to *apricot*, a result that seems sensible. Fig. 6.7 shows a visualization.

6.5 TF-IDF: Weighing terms in the vector

The co-occurrence matrix in Fig. 6.5 represented each cell by the raw frequency of the co-occurrence of two words.

It turns out, however, that simple frequency isn't the best measure of association between words. One problem is that raw frequency is very skewed and not very discriminative. If we want to know what kinds of contexts are shared by *apricot* and *pineapple* but not by *digital* and *information*, we're not going to get good discrimination from words like *the*, *it*, or *they*, which occur frequently with all sorts of words and aren't informative about any particular word.

It's a bit of a paradox. Word that occur nearby frequently (maybe *sugar* appears often in our corpus near *apricot*) are more important than words that only appear

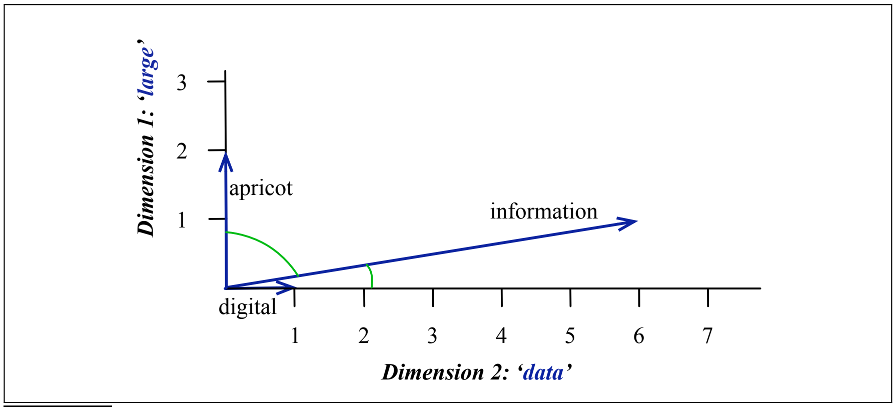


Figure 6.7 A graphical demonstration of the cosine measure of similarity, showing vectors for three words (*apricot*, *digital*, and *information*) in the two dimensional space defined by counts of the words *data* and *large* in the neighborhood. Note that the angle between *digital* and *information* is smaller than the angle between *apricot* and *information*. When two vectors are more similar, the cosine is larger but the angle is smaller; the cosine has its maximum (1) when the angle between two vectors is smallest (0°); the cosine of all other angles is less than 1.

once or twice. Yet words that are too frequent—ubiquitous, like *the*— are unimportant. How can we balance these two conflicting constraints?

The **tf-idf algorithm** (the '-' here is a hyphen, not a minus sign) algorithm is the product of two terms, each term capturing one of these two intuitions:

term frequency

1. The first is the **term frequency** (Luhn, 1957): simply the frequency of the word in the document, although we may also use functions of this frequency like the log frequency.

inverse document frequency IDF

2. The second factor is used to give a higher weight to words that occur only in a few documents. Terms that are limited to a few documents are useful for discriminating those documents from the rest of the collection; terms that occur frequently across the entire collection aren't as helpful. The **inverse document frequency** or **IDF** term weight (Sparck Jones, 1972) is one way of assigning higher weights to these more discriminative words. IDF is defined using the fraction N/df_i, where N is the total number of documents in the collection, and df_i is the number of documents in which term i occurs. The fewer documents in which a term occurs, the higher this weight. The lowest weight of 1 is assigned to terms that occur in all the documents. Because of the large number of documents in many collections, this measure is usually squashed with a log function.

   It's usually clear what counts as a document: when processing a collection of encyclopedia articles like Wikipedia, the document is a Wikipedia page; in processing newspaper articles, the document is a single article. Occasionally your corpus might not have appropriate document divisions and you might need to break up the corpus into documents yourself.

   The resulting definition for inverse document frequency (IDF) is thus

$$\text{idf}_i = \log\left(\frac{N}{\text{df}_i}\right) \tag{6.12}$$

tf-idf

The **tf-idf** weighting of the value for word i in document j, w_{ij} thus combines

term frequency with IDF:

$$w_{ij} = \text{tf}_{ij}\text{idf}_i \tag{6.13}$$

The tf-idf weighting is by far the dominant way of weighting co-occurrence matrices in information retrieval, but also plays a role in many other aspects of natural language processing. It's also a great baseline, the simple thing to try first.

6.6 Applications of the tf-idf vector model

In summary, the vector semantics model we've described so far represents a target word as a vector with dimensions corresponding to all the words in the vocabulary (length $|V|$, with vocabularies of 20,000 to 50,000), which is also sparse (most values are zero). The values in each dimension are the frequency with which the target word co-occurs with each neighboring context word, weighted by tf-idf. The model computes the similarity between two words x and y by taking the cosine of their tf-idf vectors; high cosine, high similarity. This entire model is sometimes referred to for short as the **tf-idf** model, after the weighting function.

One common use for a tf-idf model is to compute word similarity, a useful tool for tasks like finding word paraphrases, tracking changes in word meaning, or automatically discovering meanings of words in different corpora. For example, we can find the 10 most similar words to any target word w by computing the cosines between w and each of the $V-1$ other words, sorting, and looking at the top 10.

The tf-idf vector model can also be used to decide if two documents are similar. We represent a document by taking the vectors of all the words in the document, and computing the **centroid** of all those vectors. The centroid is the multidimensional version of the mean; the centroid of a set of vectors is a single vector that has the minimum sum of squared distances to each of the vectors in the set. Given k word vectors $w_1, w_2, ..., w_k$, the centroid **document vector** d is:

centroid

document vector

$$d = \frac{w_1 + w_2 + ... + w_k}{k} \tag{6.14}$$

Given two documents, we can then compute their document vectors d_1 and d_2, and estimate the similarity between the two documents by $\cos(d_1, d_2)$.

Document similarity is also useful for all sorts of applications; information retrieval, plagiarism detection, news recommender systems, and even for digital humanities tasks like comparing different versions of a text to see which are similar to each other.

6.7 Word2vec

In the previous sections we saw how to represent a word as a sparse, long vector with dimensions corresponding to the words in the vocabulary, and whose values were tf-idf or other functions of the count of the word co-occurring with each neighboring word. In this section we turn to an alternative method for representing a word: the use of vectors that are **short** (of length perhaps 50-500) and **dense** (most values are non-zero).

It turns out that dense vectors work better in every NLP task than sparse vectors. While we don't complete understand all the reasons for this, we have some intuitions. First, dense vectors may be more successfully included as features in machine learning systems; for example if we use 100-dimensional word embeddings as features, a classifier can just learn 100 weights to represent a function of word meaning; if we instead put in a 50,000 dimensional vector, a classifier would have to learn tens of thousands of weights for each of the sparse dimensions. Second, because they contain fewer parameters than sparse vectors of explicit counts, dense vectors may generalize better and help avoid overfitting. Finally, dense vectors may do a better job of capturing synonymy than sparse vectors. For example, *car* and *automobile* are synonyms; but in a typical sparse vector representation, the *car* dimension and the *automobile* dimension are distinct dimensions. Because the relationship between these two dimensions is not modeled, sparse vectors may fail to capture the similarity between a word with *car* as a neighbor and a word with *automobile* as a neighbor.

skip-gram
SGNS
word2vec

In this section we introduce one method for very dense, short vectors, **skip-gram with negative sampling**, sometimes called **SGNS**. The skip-gram algorithm is one of two algorithms in a software package called **word2vec**, and so sometimes the algorithm is loosely referred to as word2vec (Mikolov et al. 2013, Mikolov et al. 2013a). The word2vec methods are fast, efficient to train, and easily available online with code and pretrained embeddings. We point to other embedding methods, like the equally popular *GloVe* (Pennington et al., 2014), at the end of the chapter.

The intuition of word2vec is that instead of counting how often each word w occurs near, say, *apricot*, we'll instead train a classifier on a binary prediction task: "Is word *w* likely to show up near *apricot*?" We don't actually care about this prediction task; instead we'll take the learned classifier *weights* as the word embeddings.

The revolutionary intuition here is that we can just use running text as implicitly supervised training data for such a classifier; a word *s* that occurs near the target word *apricot* acts as gold 'correct answer' to the question "Is word *w* likely to show up near *apricot*?" This avoids the need for any sort of hand-labeled supervision signal. This idea was first proposed in the task of neural language modeling, when Bengio et al. (2003) and Collobert et al. (2011) showed that a neural language model (a neural network that learned to predict the next word from prior words) could just use the next word in running text as its supervision signal, and could be used to learn an embedding representation for each word as part of doing this prediction task.

We'll see how to do neural networks in the next chapter, but word2vec is a much simpler model than the neural network language model, in two ways. First, word2vec simplifies the task (making it binary classification instead of word prediction). Second, word2vec simplifies the architecture (training a logistic regression classifier instead of a multi-layer neural network with hidden layers that demand more sophisticated training algorithms). The intuition of skip-gram is:

1. Treat the target word and a neighboring context word as positive examples.
2. Randomly sample other words in the lexicon to get negative samples
3. Use logistic regression to train a classifier to distinguish those two cases
4. Use the regression weights as the embeddings

6.7.1 The classifier

Let's start by thinking about the classification task, and then turn to how to train. Imagine a sentence like the following, with a target word *apricot* and assume we're using a window of ±2 context words:

```
... lemon,  a [tablespoon of apricot jam,     a] pinch ...
               c1         c2    t     c3      c4
```

Our goal is to train a classifier such that, given a tuple (t,c) of a target word t paired with a candidate context word c (for example (*apricot*, *jam*), or perhaps (*apricot*, *aardvark*) it will return the probability that c is a real context word (true for *jam*, false for *aardvark*):

$$P(+|t,c) \tag{6.15}$$

The probability that word c is not a real context word for t is just 1 minus Eq. 6.15:

$$P(-|t,c) = 1 - P(+|t,c) \tag{6.16}$$

How does the classifier compute the probability P? The intuition of the skip-gram model is to base this probability on similarity: a word is likely to occur near the target if its embedding is similar to the target embedding. How can we compute similarity between embeddings? Recall that two vectors are similar if they have a high dot product (cosine, the most popular similarity metric, is just a normalized dot product). In other words:

$$Similarity(t,c) \approx t \cdot c \tag{6.17}$$

Of course, the dot product $t \cdot c$ is not a probability, it's just a number ranging from 0 to ∞. (Recall, for that matter, that cosine isn't a probability either). To turn the dot product into a probability, we'll use the **logistic** or **sigmoid** function $\sigma(x)$, the fundamental core of logistic regression:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{6.18}$$

The probability that word c is a real context word for target word t is thus computed as:

$$P(+|t,c) = \frac{1}{1+e^{-t \cdot c}} \tag{6.19}$$

The sigmoid function just returns a number between 0 and 1, so to make it a probability we'll need to make sure that the total probability of the two possible events (c being a context word, and c not being a context word) sum to 1.

The probability that word c is not a real context word for t is thus:

$$\begin{aligned} P(-|t,c) &= 1 - P(+|t,c) \\ &= \frac{e^{-t \cdot c}}{1+e^{-t \cdot c}} \end{aligned} \tag{6.20}$$

Equation 6.19 give us the probability for one word, but we need to take account of the multiple context words in the window. Skip-gram makes the strong but very useful simplifying assumption that all context words are independent, allowing us to just multiply their probabilities:

$$P(+|t,c_{1:k}) = \prod_{i=1}^{k} \frac{1}{1+e^{-t\cdot c_i}} \tag{6.21}$$

$$\log P(+|t,c_{1:k}) = \sum_{i=1}^{k} \log \frac{1}{1+e^{-t\cdot c_i}} \tag{6.22}$$

In summary, skip-gram trains a probabilistic classifier that, given a test target word t and its context window of k words $c_{1:k}$, assigns a probability based on how similar this context window is to the target word. The probability is based on applying the logistic (sigmoid) function to the dot product of the embeddings of the target word with each context word. We could thus compute this probability if only we had embeddings for each word target and context word in the vocabulary. Let's now turn to learning these embeddings (which is the real goal of training this classifier in the first place).

6.7.2 Learning skip-gram embeddings

Word2vec learns embeddings by starting with an initial set of embedding vectors and then iteratively shifting the embedding of each word w to be more like the embeddings of words that occur nearby in texts, and less like the embeddings of words that don't occur nearby.

Let's start by considering a single piece of the training data, from the sentence above:

```
... lemon,  a [tablespoon of apricot jam,      a] pinch ...
              c1          c2     t      c3        c4
```

This example has a target word t (apricot), and 4 context words in the $L = \pm 2$ window, resulting in 4 positive training instances (on the left below):

positive examples +

t	c
apricot	tablespoon
apricot	of
apricot	preserves
apricot	or

negative examples -

t	c	t	c
apricot	aardvark	apricot	twelve
apricot	puddle	apricot	hello
apricot	where	apricot	dear
apricot	coaxial	apricot	forever

For training a binary classifier we also need negative examples, and in fact skip-gram uses more negative examples than positive examples, the ratio set by a parameter k. So for each of these (t,c) training instances we'll create k negative samples, each consisting of the target t plus a 'noise word'. A noise word is a random word from the lexicon, constrained not to be the target word t. The right above shows the setting where $k = 2$, so we'll have 2 negative examples in the negative training set $-$ for each positive example t,c.

The noise words are chosen according to their weighted unigram frequency $p_\alpha(w)$, where α is a weight. If we were sampling according to unweighted frequency $p(w)$, it would mean that with unigram probability $p(\text{“}the\text{”})$ we would choose

the word *the* as a noise word, with unigram probability p("*aardvark*") we would choose *aardvark*, and so on. But in practice it is common to set $\alpha = .75$, i.e. use the weighting $p^{\frac{3}{4}}(w)$:

$$P_\alpha(w) = \frac{count(w)^\alpha}{\sum_w count(w)^\alpha} \quad (6.23)$$

Setting $\alpha = .75$ gives better performance because it gives rare noise words slightly higher probability: for rare words, $P_\alpha(w) > P(w)$. To visualize this intuition, it might help to work out the probabilities for an example with two events, $P(a) = .99$ and $P(b) = .01$:

$$\begin{aligned} P_\alpha(a) &= \frac{.99^{.75}}{.99^{.75} + .01^{.75}} = .97 \\ P_\alpha(b) &= \frac{.01^{.75}}{.99^{.75} + .01^{.75}} = .03 \end{aligned} \quad (6.24)$$

Given the set of positive and negative training instances, and an initial set of embeddings, the goal of the learning algorithm is to adjust those embeddings such that we

- Maximize the similarity of the target word, context word pairs (t,c) drawn from the positive examples
- Minimize the similarity of the (t,c) pairs drawn from the negative examples.

We can express this formally over the whole training set as:

$$L(\theta) = \sum_{(t,c)\in +} \log P(+|t,c) + \sum_{(t,c)\in -} \log P(-|t,c) \quad (6.25)$$

Or, focusing in on one word/context pair (t,c) with its k noise words $n_1 ... n_k$, the learning objective L is:

$$\begin{aligned} L(\theta) &= \log P(+|t,c) + \sum_{i=1}^{k} \log P(-|t,n_i) \\ &= \log \sigma(c \cdot t) + \sum_{i=1}^{k} \log \sigma(-n_i \cdot t) \\ &= \log \frac{1}{1+e^{-c \cdot t}} + \sum_{i=1}^{k} \log \frac{1}{1+e^{n_i \cdot t}} \end{aligned} \quad (6.26)$$

That is, we want to maximize the dot product of the word with the actual context words, and minimize the dot products of the word with the k negative sampled non-neighbor words.

We can then use stochastic gradient descent to train to this objective, iteratively modifying the parameters (the embeddings for each target word t and each context word or noise word c in the vocabulary) to maximize the objective.

target embedding **context embedding**

Note that the skip-gram model thus actually learns **two** separate embeddings for each word w: the **target embedding** t and the **context embedding** c. These embeddings are stored in two matrices, the **target matrix** T and the **context matrix**

C. So each row i of the target matrix T is the $1 \times d$ vector embedding t_i for word i in the vocabulary V, and each column i of the context matrix C is a $d \times 1$ vector embedding c_i for word i in V. Fig. 6.8 shows an intuition of the learning task for the embeddings encoded in these two matrices.

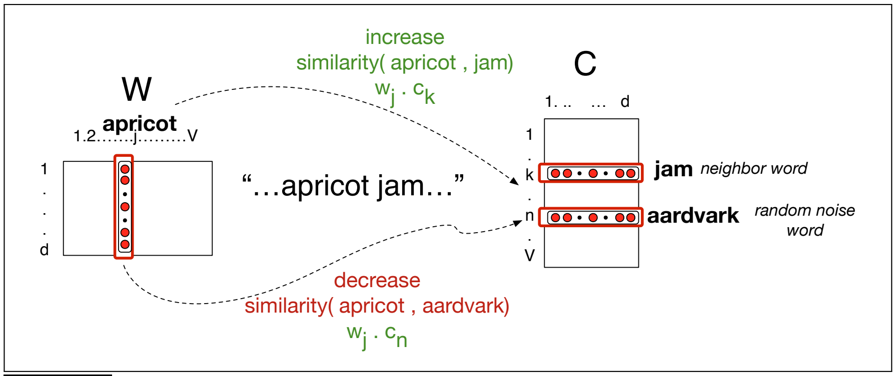


Figure 6.8 The skip-gram model tries to shift embeddings so the target embedding (here for *apricot*) are closer to (have a higher dot product with) context embeddings for nearby words (here *jam*) and further from (have a lower dot product with) context embeddings for words that don't occur nearby (here *aardvark*).

Just as in logistic regression, then, the learning algorithm starts with randomly initialized W and C matrices, and then walks through the training corpus using gradient descent to move W and C so as to maximize the objective in Eq. 6.26. Thus the matrices W and C function as the parameters θ that logistic regression is tuning.

Once the embeddings are learned, we'll have two embeddings for each word w_i: t_i and c_i. We can choose to throw away the C matrix and just keep W, in which case each word i will be represented by the vector t_i.

Alternatively we can add the two embeddings together, using the summed embedding $t_i + c_i$ as the new d-dimensional embedding, or we can concatenate them into an embedding of dimensionality $2d$.

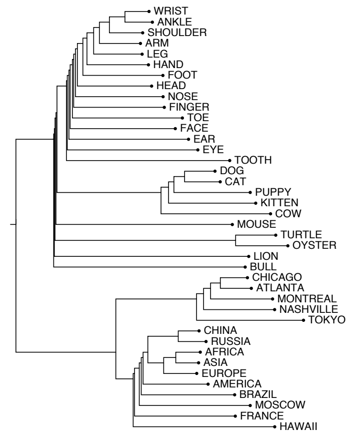
As with the simple count-based methods like tf-idf, the context window size L effects the performance of skip-gram embeddings, and experiments often tune the parameter L on a dev set. One difference from the count-based methods is that for skip-grams, the larger the window size the more computation the algorithm requires for training (more neighboring words must be predicted).

6.8 Visualizing Embeddings

Visualizing embeddings is an important goal in helping understands, apply, and improve these models of word meaning. But how can we visualize a (for example) 100-dimensional vector?

The simplest way to visualize the meaning of a word w embedded in a space is to list the most similar words to w sorting all words in the vocabulary by their cosines. For example the 7 closest words to *frog* using the GloVe embeddings are: *frogs*, *toad*, *litoria*, *leptodactylidae*, *rana*, *lizard*, and *eleutherodactylus* (Pennington et al., 2014)

Yet another visualization method is to use a clustering algorithm to show a hierarchical representation of which words are similar to others in the embedding space. The example on the right uses hierarchical clustering of some embedding vectors for nouns as a visualization method (Rohde et al., 2006).



Probably the most common visualization method, however, is to project the 100 dimensions of a word down into 2 dimensions. Fig. 6.1 showed one such visualization, using a projection method called t-SNE (van der Maaten and Hinton, 2008).

6.9 Semantic properties of embeddings

Vector semantic models have a number of parameters. One parameter that is relevant to both sparse tf-idf vectors and dense word2vec vectors is the size of the context window used to collect counts. This is generally between 1 and 10 words on each side of the target word (for a total context of 3-20 words).

The choice depends on on the goals of the representation. Shorter context windows tend to lead to representations that are a bit more syntactic, since the information is coming from immediately nearby words. When the vectors are computed from short context windows, the most similar words to a target word *w* tend to be semantically similar words with the same parts of speech. When vectors are computed from long context windows, the highest cosine words to a target word *w* tend to be words that are topically related but not similar.

For example Levy and Goldberg (2014a) showed that using skip-gram with a window of ±2, the most similar words to the word *Hogwarts* (from the Harry Potter series) were names of other fictional schools: *Sunnydale (from Buffy the Vampire Slayer)* or *Evernight* (from a vampire series). With a window of ±5, the most similar words to *Hogwarts* were other words topically related to the Harry Potter series: *Dumbledore*, *Malfoy*, and *half-blood*.

first-order co-occurrence

second-order co-occurrence

It's also often useful to distinguish two kinds of similarity or association between words (Schütze and Pedersen, 1993). Two words have **first-order co-occurrence** (sometimes called **syntagmatic association**) if they are typically nearby each other. Thus *wrote* is a first-order associate of *book* or *poem*. Two words have **second-order co-occurrence** (sometimes called **paradigmatic association**) if they have similar neighbors. Thus *wrote* is a second-order associate of words like *said* or *remarked*.

Analogy Another semantic property of embeddings is their ability to capture relational meanings. Mikolov et al. (2013b) and Levy and Goldberg (2014b) show that the *offsets* between vector embeddings can capture some analogical relations between words. For example, the result of the expression vector(*'king'*) - vector(*'man'*) + vector(*'woman'*) is a vector close to vector(*'queen'*); the left panel in Fig. 6.9 visualizes this, again projected down into 2 dimensions. Similarly, they found that the expression vector(*'Paris'*) - vector(*'France'*) + vector(*'Italy'*) results in a vector that is very close to vector(*'Rome'*).

Embeddings and Historical Semantics: Embeddings can also be a useful tool for studying how meaning changes over time, by computing multiple embedding spaces, each from texts written in a particular time period. For example Fig. 6.10

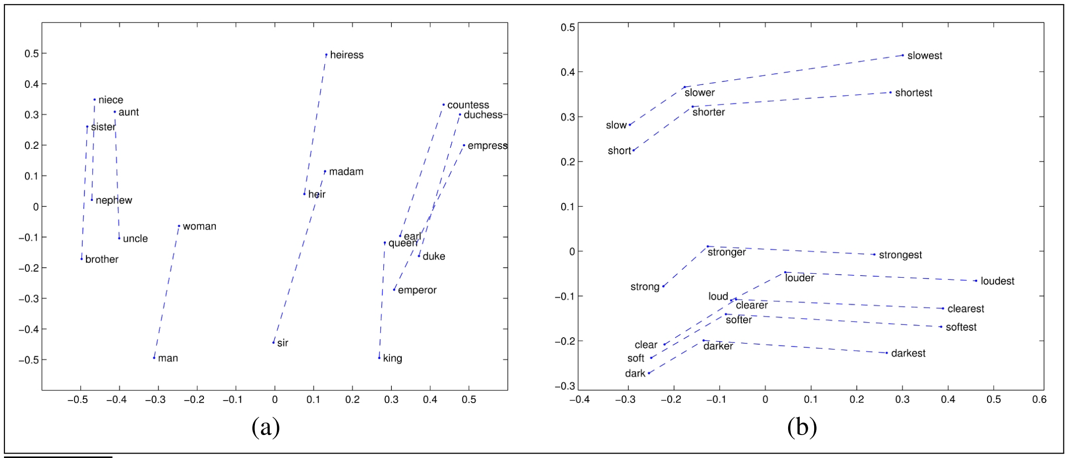


Figure 6.9 Relational properties of the vector space, shown by projecting vectors onto two dimensions. (a) 'king' - 'man' + 'woman' is close to 'queen' (b) offsets seem to capture comparative and superlative morphology (Pennington et al., 2014).

shows a visualization of changes in meaning in English words over the last two centuries, computed by building separate embedding spaces for each decade from historical corpora like Google N-grams (Lin et al., 2012) and the Corpus of Historical American English (Davies, 2012).

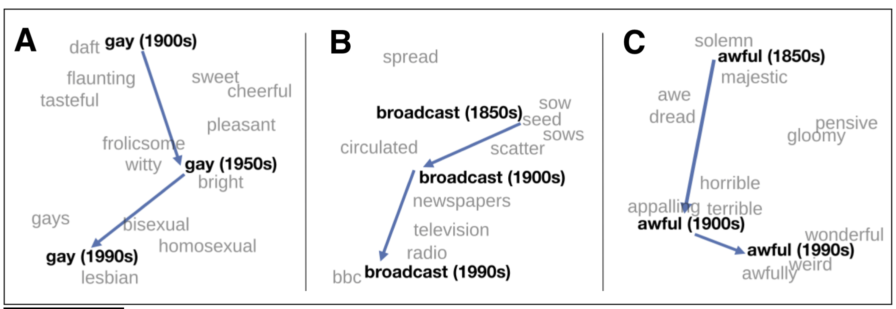


Figure 6.10 A t-SNE visualization of the semantic change of 3 words in English using word2vec vectors. The modern sense of each word, and the grey context words, are computed from the most recent (modern) time-point embedding space. Earlier points are computed from earlier historical embedding spaces. The visualizations show the changes in the word *gay* from meanings related to "cheerful" or "frolicsome" to referring to homosexuality, the development of the modern "transmission" sense of *broadcast* from its original sense of sowing seeds, and the pejoration of the word *awful* as it shifted from meaning "full of awe" to meaning "terrible or appalling" (Hamilton et al., 2016).

6.10 Bias and Embeddings

In addition to their ability to learn word meaning from text, embeddings, alas, also reproduce implicit biases and stereotypes. Recall that embeddings model analogical relations; 'queen' as the closest word to 'king' - 'man' + 'woman' implies the analogy *man:woman::king:queen*. But embeddings also exhibit gender stereotypes. (Bolukbasi et al., 2016) find that the closest occupation to 'man' - 'computer

programmer' + 'woman' in word2vec embeddings trained on news text is 'homemaker', and that the embeddings similarly suggest the analogy 'father' is to 'doctor' as 'mother' is to 'nurse'. Algorithms that used embeddings as part of an algorithm to search for potential programmers or doctors might thus incorrectly downweight documents with women's names.

Embeddings also encode the implicit associations that are a property of human reasoning. The Implicit Association Test (Greenwald et al., 1998) measures people's associations between concepts (like 'flowers' or 'insects') and attributes (like 'pleasantness' and 'unpleasantness') by measuring differences in the latency with which they label words in the various categories.[3] Using such methods, people in the United States have been shown to associate African-American names with unpleasant words (more than European-American names), male names more with mathematics and female names with the arts, and old people's names with unpleasant words (Greenwald et al. 1998, Nosek et al. 2002a, Nosek et al. 2002b). Caliskan et al. (2017) replicated all these findings of implicit associations using GloVe vectors and cosine similarity instead of human latencies. For example Afrian American names like 'Leroy' and 'Shaniqua' had a higher GloVe cosine with unpleasant words while European American names ('Brad', 'Greg', 'Courtney') had a higher cosine with pleasant words. Any embedding-aware algorithm that made use of word sentiment could thus lead to bias against African Americans.

Recent research focuses on ways to try to remove the kinds of biases, for example by developing a transformation of the embedding space that removes gender stereotypes but preserves definitional gender (Bolukbasi et al. 2016, Zhao et al. 2017).

Historical embeddings are also being used to measure biases in the past. Garg et al. (2018) used embeddings from historical texts to measure the association between embeddings for occupations and embeddings for names of various ethnicities or genders (for example the relative cosine similarity of women's names versus men's to occupation words like 'librarian' or 'carpenter') across the 20th century. They found that the cosines correlate with the empirical historical percentages of women or ethnic groups in those occupation. Historical embeddings also replicated old surveys of ethnic stereotypes; the tendency of experimental participants in 1933 to associate adjectives like 'industrious' or 'superstitious' with, e.g., Chinese ethnicity, correlates with the cosine between Chinese last names and those adjectives using embeddings trained on 1930s text. They also were able to document historical gender biases, such as the fact that embeddings for adjectives related to competence ('smart', 'wise', 'thoughtful', 'resourceful') had a higher cosine with male than female words, and showed that this bias has been slowly decreasing since 1960.

We will return in later chapters to this question about the role of bias in natural language processing and machine learning in general.

6.11 Evaluating Vector Models

The most important evaluation metric for vector models is extrinsic evaluation on tasks; adding them as features into any NLP task and seeing whether this improves

[3] Roughly speaking, if humans associate 'flowers' with 'pleasantness' and 'insects' with 'unpleasantness', when they are instructed to push a red button for 'flowers' (daisy, iris, lilac) and 'pleasant words' (love, laughter, pleasure) and a green button for 'insects' (flea, spider, mosquito) and 'unpleasant words' (abuse, hatred, ugly) they are faster than in an incongruous condition where they push a red button for 'flowers' and 'unpleasant words' and a green button for 'insects' and 'pleasant words'.

performance over some other model.

Nonetheless it is useful to have intrinsic evaluations. The most common metric is to test their performance on **similarity**, computing the correlation between an algorithm's word similarity scores and word similarity ratings assigned by humans. **WordSim-353** (Finkelstein et al., 2002) is a commonly used set of of ratings from 0 to 10 for 353 noun pairs; for example (*plane*, *car*) had an average score of 5.77. **SimLex-999** (Hill et al., 2015) is a more difficult dataset that quantifies similarity (*cup, mug*) rather than relatedness (*cup, coffee*), and including both concrete and abstract adjective, noun and verb pairs. The **TOEFL dataset** is a set of 80 questions, each consisting of a target word with 4 additional word choices; the task is to choose which is the correct synonym, as in the example: *Levied is closest in meaning to: imposed, believed, requested, correlated* (Landauer and Dumais, 1997). All of these datasets present words without context.

Slightly more realistic are intrinsic similarity tasks that include context. The Stanford Contextual Word Similarity (SCWS) dataset (Huang et al., 2012) offers a richer evaluation scenario, giving human judgments on 2,003 pairs of words in their sentential context, including nouns, verbs, and adjectives. This dataset enables the evaluation of word similarity algorithms that can make use of context words. The *semantic textual similarity* task (Agirre et al. 2012, Agirre et al. 2015) evaluates the performance of sentence-level similarity algorithms, consisting of a set of pairs of sentences, each pair with human-labeled similarity scores.

Another task used for evaluate is an analogy task, where the system has to solve problems of the form *a is to b as c is to d*, given *a, b,* and *c* and having to find *d*. Thus given *Athens is to Greece as Oslo is to* ______, the system must fill in the word *Norway*. Or more syntactically-oriented examples: given *mouse*, *mice*, and *dollar* the system must return *dollars*. Large sets of such tuples have been created (Mikolov et al. 2013, Mikolov et al. 2013b).

6.12 Summary

- In vector semantics, a word is modeled as a vector—a point in high-dimensional space, also called an **embedding**.
- Vector semantic models fall into two classes: **sparse** and **dense**. In sparse models like **tf-idf** each dimension corresponds to a word in the vocabulary V;
- Cell in sparse models are functions of **co-occurrence counts**. The **term-document** matrix has rows for each word (**term**) in the vocabulary and a column for each document.
- The **word-context** matrix has a row for each (target) word in the vocabulary and a column for each context term in the vocabulary.
- A common sparse weighting is **tf-idf**, which weights each cell by its **term frequency** and **inverse document frequency**.
- Word and document similarity is computed by computing the **dot product** between vectors. The cosine of two vectors—a normalized dot product—is the most popular such metric.
- Dense vector models have dimensionality 50-300 and the dimensions are harder to interpret.
- The **word2vec** family of models, including **skip-gram** and **CBOW**, is a popular efficient way to compute dense embeddings.

- Skip-gram trains a logistic regression classifier to compute the probability that two words are 'likely to occur nearby in text'. This probability is computed from the dot product between the embeddings for the two words,
- Skip-gram use stochastic gradient descent to train the classifier, by learning embeddings that have a high dot-product with embeddings of words that occur nearby and a low dot-product with noise words.

Bibliographical and Historical Notes

The idea of vector semantics arose out of research in the 1950s in three distinct fields: linguistics, psychology, and computer science, each of which contributed a fundamental aspect of the model.

The idea that meaning was related to distribution of words in context was widespread in linguistic theory of the 1950s, among distributionalists like Zellig Harris, Martin Joos, and J. R. Firth, and semioticians like Thomas Sebeok. As Joos (1950) put it,

> the linguist's "meaning" of a morpheme... is by definition the set of conditional probabilities of its occurrence in context with all other morphemes.

The idea that the meaning of a word might be modeled as a point in a multi-dimensional semantic space came from psychologists like Charles E. Osgood, who had been studying how people responded to the meaning of words by assigning values along scales like *happy/sad*, or *hard/soft*. Osgood et al. (1957) proposed that the meaning of a word in general could be modeled as a point in a multidimensional Euclidean space, and that the similarity of meaning between two words could be modeled as the distance between these points in the space.

mechanical indexing

A final intellectual source in the 1950s and early 1960s was the field then called **mechanical indexing**, now known as **information retrieval**. In what became known as the **vector space model** for information retrieval (Salton 1971,Sparck Jones 1986), researchers demonstrated new ways to define the meaning of words in terms of vectors (Switzer, 1965), and refined methods for word similarity based on measures of statistical association between words like mutual information (Giuliano, 1965) and IDF (Sparck Jones, 1972), and showed that the meaning of documents could be represented in the same vector spaces used for words.

semantic feature

More distantly related is the idea of defining words by a vector of discrete features, which has a venerable history in our field, with roots at least as far back as Descartes and Leibniz (Wierzbicka 1992, Wierzbicka 1996). By the middle of the 20th century, beginning with the work of Hjelmslev (Hjelmslev, 1969) and fleshed out in early models of generative grammar (Katz and Fodor, 1963), the idea arose of representing meaning with **semantic features**, symbols that represent some sort of primitive meaning. For example words like *hen*, *rooster*, or *chick*, have something in common (they all describe chickens) and something different (their age and sex), representable as:

hen	`+female, +chicken, +adult`
rooster	`-female, +chicken, +adult`
chick	`+chicken, -adult`

The dimensions used by vector models of meaning to define words, however, are only abstractly related to this idea of a small fixed number of hand-built dimensions.

Nonetheless, there has been some attempt to show that certain dimensions of embedding models do contribute some specific compositional aspect of meaning like these early semantic features.

The first use of dense vectors to model word meaning was the **latent semantic indexing** (LSI) model (Deerwester et al., 1988) recast as **LSA** (**latent semantic analysis**) (Deerwester et al., 1990). In LSA SVD is applied to a term-document matrix (each cell weighted by log frequency and normalized by entropy), and then using the first 300 dimensions as the embedding. LSA was then quickly widely applied: as a cognitive model Landauer and Dumais (1997), and tasks like spell checking (Jones and Martin, 1997), language modeling (Bellegarda 1997, Coccaro and Jurafsky 1998, Bellegarda 2000) morphology induction (Schone and Jurafsky 2000, Schone and Jurafsky 2001), and essay grading (Rehder et al., 1998). Related models were simultaneously developed and applied to word sense disambiguation by Schütze (1992b). LSA also led to the earliest use of embeddings to represent words in a probabilistic classifier, in the logistic regression document router of Schütze et al. (1995). The idea of SVD on the term-term matrix (rather than the term-document matrix) as a model of meaning for NLP was proposed soon after LSA by Schütze (1992b). Schütze applied the low-rank (97-dimensional) embeddings produced by SVD to the task of word sense disambiguation, analyzed the resulting semantic space, and also suggested possible techniques like dropping high-order dimensions. See Schütze (1997a).

A number of alternative matrix models followed on from the early SVD work, including Probabilistic Latent Semantic Indexing (PLSI) (Hofmann, 1999) Latent Dirichlet Allocation (LDA) (Blei et al., 2003). Nonnegative Matrix Factorization (NMF) (Lee and Seung, 1999).

By the next decade, Bengio et al. (2003) and Bengio et al. (2006) showed that neural language models could also be used to develop embeddings as part of the task of word prediction. Collobert and Weston (2007), Collobert and Weston (2008), and Collobert et al. (2011) then demonstrated that embeddings could play a role for representing word meanings for a number of NLP tasks. (Turian et al., 2010) compared the value of different kinds of embeddings for different NLP tasks. Mikolov et al. (2011) showed that recurrent neural nets could be used as language models. The idea of simplifying the hidden layer of these neural net language models to create the skip-gram and CBOW algorithms was proposed by Mikolov et al. (2013). The negative sampling training algorithm was proposed in Mikolov et al. (2013a).

The other most widely-used embedding model is GloVe (Pennington et al., 2014), which is based on ratios of probabilities from the word-word co-occurrence matrix. A popular sparse embedding algorithm weighting that is an alternative to tf-idf is **PPMI** (Positive Pointwise Mutual Information) (Church and Hanks 1989, Dagan et al. 1993, Niwa and Nitta 1994). Emebeddings have been build using non-negative matrix factorization (Fyshe et al., 2015).

PPMI

Studies of embeddings include results showing an elegant mathematical relationship between sparse and dense embeddings (Levy and Goldberg, 2014c), as well as numerous surveys of embeddings and their parameterizations. (Bullinaria and Levy 2007, Bullinaria and Levy 2012, Lapesa and Evert 2014, Kiela and Clark 2014, Levy et al. 2015).

Exercises

CHAPTER

7 Neural Networks and Neural Language Models

"[M]achines of this character can behave in a very complicated manner when the number of units is large."

Alan Turing (1948) "Intelligent Machines", page 6

Neural networks are an essential computational tool for language processing, and a very old one. They are called neural because their origins lie in the **McCulloch-Pitts neuron** (McCulloch and Pitts, 1943), a simplified model of the human neuron as a kind of computing element that could be described in terms of propositional logic. But the modern use in language processing no longer draws on these early biological inspirations.

Instead, a modern neural network is a network of small computing units, each of which takes a vector of input values and produces a single output value. In this chapter we introduce the neural net applied to classification. The architecture we introduce is called a **feed-forward network** because the computation proceeds iteratively from one layer of units to the next. The use of modern neural nets is often called **deep learning**, because modern networks are often **deep** (have many layers).

deep learning

deep

Neural networks share much of the same mathematics as logistic regression. But neural networks are a more powerful classifier than logistic regression, and indeed a minimal neural network (technically one with a single 'hidden layer') can be shown to learn any function.

Neural net classifiers are different from logistic regression in another way. With logistic regression, we applied the regression classifier to many different tasks by developing many rich kinds of feature templates based on domain knowledge. When working with neural networks, it is more common to avoid the use of rich hand-derived features, instead building neural networks that take raw words as inputs and learn to induce features as part of the process of learning to classify. We saw examples of this kind of representation learning for embeddings in Chapter 6. Nets that are very deep are particularly good at representation learning for that reason deep neural nets are the right tool for large scale problems that offer sufficient data to learn features automatically.

In this chapter we'll see feedforward networks as classifiers, and apply them to the simple task of language modeling: assigning probabilities to word sequences and predicting upcoming words. In later chapters we'll introduce many other aspects of neural models, such as the **recurrent neural network** and the **encoder-decoder** model.

7.1 Units

The building block of a neural network is a single computational unit. A unit takes a set of real valued numbers as input, performs some computation on them, and produces an output.

bias term

At its heart, a neural unit is taking a weighted sum of its inputs, with one additional term in the sum called a **bias term**. Thus given a set of inputs $x_1...x_n$, a unit has a set of corresponding weights $w_1...w_n$ and a bias b, so the weighted sum z can be represented as:

$$z = b + \sum_i w_i x_i \tag{7.1}$$

vector

Often it's more convenient to express this weighted sum using vector notation; recall from linear algebra that a **vector** is, at heart, just a list or array of numbers. Thus we'll talk about z in terms of a weight vector w, a scalar bias b, and an input vector x, and we'll replace the sum with the convenient **dot product**:

$$z = w \cdot x + b \tag{7.2}$$

As defined in Eq. 7.2, z is just a real valued number.

activation

Finally, instead of using z, a linear function of x, as the output, neural units apply a non-linear function f to z. We will refer to the output of this function as the **activation** value for the unit, a. Since we are just modeling a single unit, the activation for the node is in fact the final output of the network, which we'll generally call y. So the value y is defined as:

$$y = a = f(z) \tag{7.3}$$

sigmoid

We'll discuss three popular non-linear functions $f()$ below (the sigmoid, the tanh, and the rectified linear ReLU) but it's pedagogically convenient to start with the **sigmoid** function since we saw it in Chapter 5:

$$y = \sigma(z) = \frac{1}{1+e^{-z}} \tag{7.4}$$

The sigmoid (shown in Fig. 7.1) has a number of advantages; it maps the output into the range $[0,1]$, which is useful in squashing outliers toward 0 or 1. And it's differentiable, which as we saw in Section 5.8 will be handy for learning.

Substituting the sigmoid equation into Eq. 7.2 gives us the final value for the output of a neural unit:

$$y = \sigma(w \cdot x + b) = \frac{1}{1+exp(-(w \cdot x + b))} \tag{7.5}$$

Fig. 7.2 shows a final schematic of a basic neural unit. In this example the unit takes 3 input values x_1, x_2, and x_3, and computes a weighted sum, multiplying each value by a weight (w_1, w_2, and w_3, respectively), adds them to a bias term b, and then passes the resulting sum through a sigmoid function to result in a number between 0 and 1.

Let's walk through an example just to get an intuition. Let's suppose we have a unit with the following weight vector and bias:

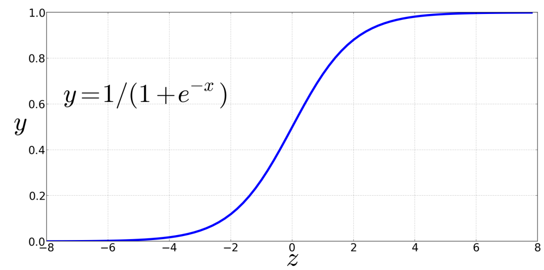


Figure 7.1 The sigmoid function takes a real value and maps it to the range $[0,1]$. Because it is nearly linear around 0 but has a sharp slope toward the ends, it tends to squash outlier values toward 0 or 1.

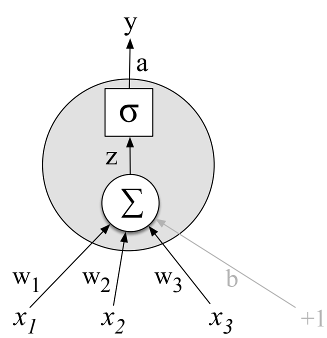


Figure 7.2 A neural unit, taking 3 inputs x_1, x_2, and x_3 (and a bias b that we represent as a weight for an input clamped at +1) and producing an output y. We include some convenient intermediate variables: the output of the summation, z, and the output of the sigmoid, a. In this case the output of the unit y is the same as a, but in deeper networks we'll reserve y to mean the final output of the entire network, leaving a as the activation of an individual node.

$$
\begin{aligned}
w &= [0.2, 0.3, 0.9] \\
b &= 0.5
\end{aligned}
$$

What would this unit do with the following input vector:

$$x = [0.5, 0.6, 0.1]$$

The resulting output y would be:

$$y = \sigma(w \cdot x + b) = \frac{1}{1+e^{-(w \cdot x+b)}} = \frac{1}{1+e^{-(.5*.2+.6*.3+.1*.9+.5)}} = e^{-0.87} = .70$$

In practice, the sigmoid is not commonly used as an activation function. A function that is very similar but almost always better is the **tanh** function shown in Fig. 7.3a; tanh is a variant of the sigmoid that ranges from -1 to +1:

tanh

$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{7.6}$$

The simplest activation function, and perhaps the most commonly used, is the rectified linear unit, also called the **ReLU**, shown in Fig. 7.3b. It's just the same as x

ReLU

when x is positive, and 0 otherwise:

$$y = max(x, 0) \tag{7.7}$$

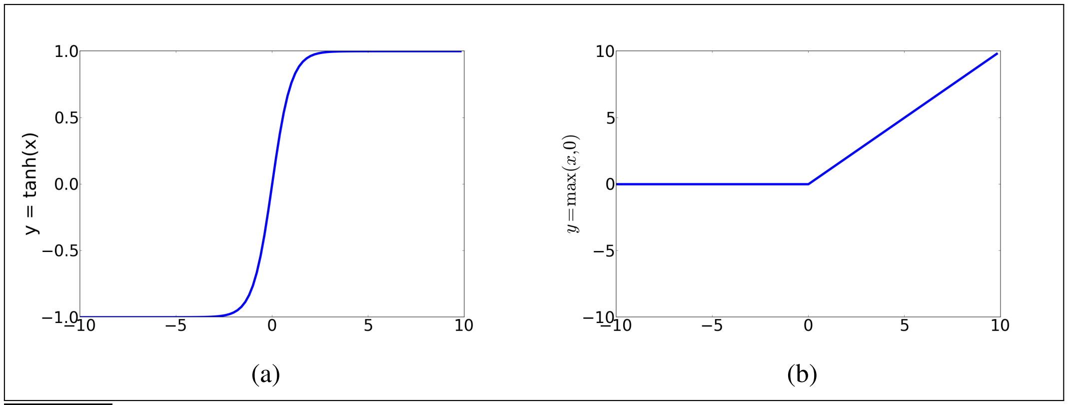


Figure 7.3 The tanh and ReLU activation functions.

These activation functions have different properties that make them useful for different language applications or network architectures. For example the rectifier function has nice properties that result from it being very close to linear. In the sigmoid or tanh functions, very high values of z result in values of y that are **saturated**, i.e., extremely close to 1, which causes problems for learning. Rectifiers don't have this problem, since the output of values close to 1 also approaches 1 in a nice gentle linear way. By contrast, the tanh function has the nice properties of being smoothly differentiable and mapping outlier values toward the mean.

saturated

7.2 The XOR problem

Early in the history of neural networks it was realized that the power of neural networks, as with the real neurons that inspired them, comes from combining these units into larger networks.

One of the most clever demonstrations of the need for multi-layer networks was the proof by Minsky and Papert (1969) that a single neural unit cannot compute some very simple functions of its input. Consider the very simple task of computing simple logical functions of two inputs, like AND, OR, and XOR. As a reminder, here are the truth tables for those functions:

AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

perceptron

This example was first shown for the **perceptron**, which is a very simple neural unit that has a binary output and no non-linear activation function. The output y of

a perceptron is 0 or 1, and just computed as follows (using the same weight w, input x, and bias b as in Eq. 7.2):

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases} \tag{7.8}$$

It's very easy to build a perceptron that can compute the logical AND and OR functions of its binary inputs; Fig. 7.4 shows the necessary weights.

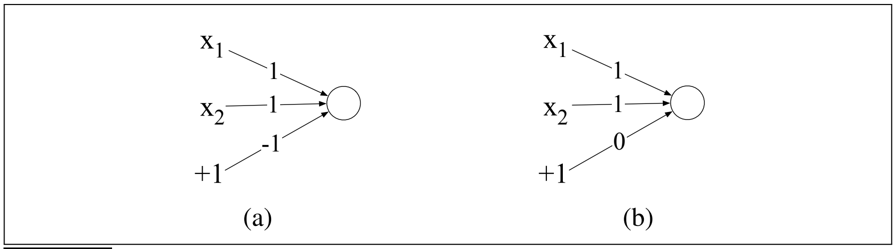


Figure 7.4 The weights w and bias b for perceptrons for computing logical functions. The inputs are shown as x_1 and x_2 and the bias as a special node with value $+1$ which is multiplied with the bias weight b. (a) logical AND, showing weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = -1$. (b) logical OR, showing weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = 0$. These weights/biases are just one from an infinite number of possible sets of weights and biases that would implement the functions.

It turns out, however, that it's not possible to build a perceptron to compute logical XOR! (It's worth spending a moment to give it a try!)

The intuition behind this important result relies on understanding that a perceptron is a linear classifier. For a two-dimensional input x_0 and x_1, the perception equation, $w_1x_1 + w_2x_2 + b = 0$ is the equation of a line (we can see this by putting it in the standard linear format: $x_2 = -(w_1/w_2)x_1 - b$.) This line acts as a **decision boundary** in two-dimensional space in which the output 0 is assigned to all inputs lying on one side of the line, and the output 1 to all input points lying on the other side of the line. If we had more than 2 inputs, the decision boundary becomes a hyperplane instead of a line, but the idea is the same, separating the space into two categories.

decision boundary

Fig. 7.5 shows the possible logical inputs (`00`, `01`, `10`, and `11`) and the line drawn by one possible set of parameters for an AND and an OR classifier. Notice that there is simply no way to draw a line that separates the positive cases of XOR (01 and 10) from the negative cases (00 and 11). We say that XOR is not a **linearly separable** function. Of course we could draw a boundary with a curve, or some other function, but not a single line.

linearly separable

7.2.1 The solution: neural networks

While the XOR function cannot be calculated by a single perceptron, it can be calculated by a layered network of units. Let's see an example of how to do this from Goodfellow et al. (2016) that computes XOR using two layers of ReLU-based units. Fig. 7.6 shows a figure with the input being processed by two layers of neural units. The middle layer (called h) has two units, and the output layer (called y) has one unit. A set of weights and biases are shown for each ReLU that correctly computes the XOR function

Let's walk through what happens with the input x = [0 0]. If we multiply each input value by the appropriate weight, sum, and then add the bias b, we get the vector

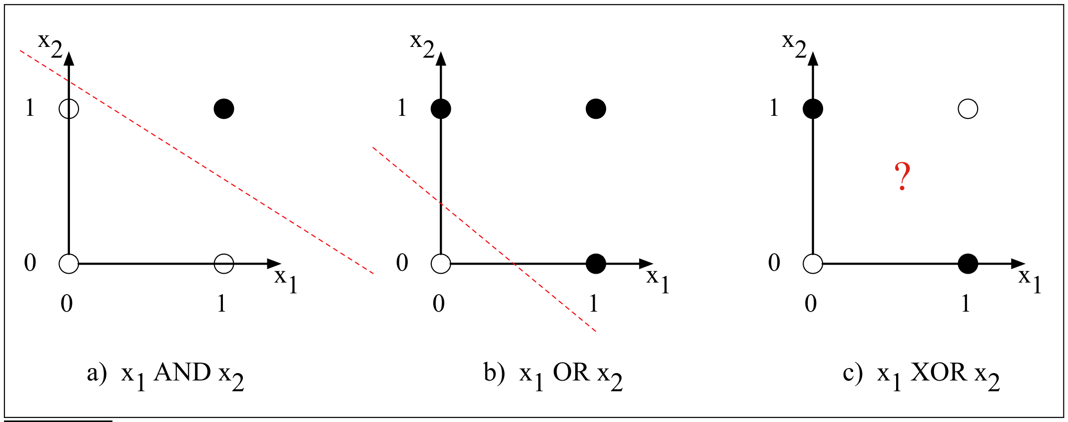


Figure 7.5 The functions AND, OR, and XOR, represented with input x_0 on the x-axis and input x_1 on the y axis, Filled circles represent perceptron outputs of 1, and white circles perceptron outputs of 0. There is no way to draw a line that correctly separates the two categories for XOR. Figure styled after Russell and Norvig (2002).

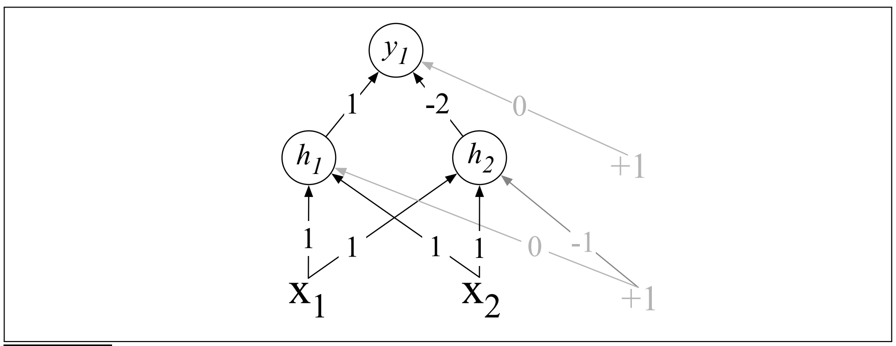


Figure 7.6 XOR solution after Goodfellow et al. (2016). There are three ReLU units, in two layers; we've called them h_1, h_2 (h for "hidden layer") and y_1. As before, the numbers on the arrows represent the weights w for each unit, and we represent the bias b as a weight on a unit clamped to +1, with the bias weights/units in gray.

[0 -1], and we then we apply the rectified linear transformation to give the output of the h layer as [0 0]. Now we once again multiply by the weights, sum, and add the bias (0 in this case) resulting in the value 0. The reader should work through the computation of the remaining 3 possible input pairs to see that the resulting y values correctly are 1 for the inputs [0 1] and [1 0] and 0 for [0 0] and [1 1].

It's also instructive to look at the intermediate results, the outputs of the two hidden nodes h_0 and h_1. We showed in the previous paragraph that the h vector for the inputs x = [0 0] was [0 0]. Fig. 7.7b shows the values of the h layer for all 4 inputs. Notice that hidden representations of the two input points x = [0 1] and x = [1 0] (the two cases with XOR output = 1) are merged to the single point h = [1 0]. The merger makes it easy to linearly separate the positive and negative cases of XOR. In other words, we can view the hidden layer of the network is forming a representation for the input.

In this example we just stipulated the weights in Fig. 7.6. But for real examples the weights for neural networks are learned automatically using the error backpropagation algorithm to be introduced in Section 7.4. That means the hidden layers will learn to form useful representations. This intuition, that neural networks can automatically learn useful representations of the input, is one of their key advantages,

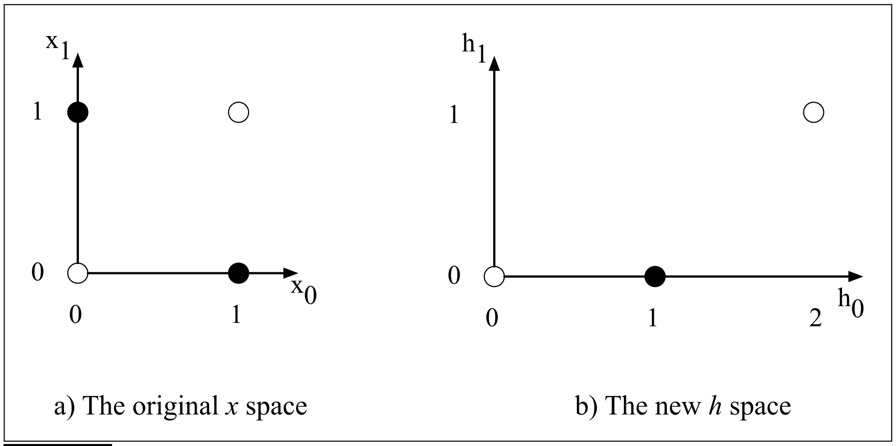


Figure 7.7 The hidden layer forming a new representation of the input. Here is the representation of the hidden layer, h, compared to the original input representation x. Notice that the input point [0 1] has been collapsed with the input point [1 0], making it possible to linearly separate the positive and negative cases of XOR. After Goodfellow et al. (2016).

and one that we will return to again and again in later chapters.

Note that the solution to the XOR problem requires a network of units with non-linear activation functions. A network made up of simple linear (perceptron) units cannot solve the XOR problem. This is because a network formed by many layers of purely linear units can always be reduced (shown to be computationally identical to) a single layer of linear units with appropriate weights, and we've already shown (visually, in Fig. 7.5) that a single unit cannot solve the XOR problem.

7.3 Feed-Forward Neural Networks

feed-forward network

Let's now walk through a slightly more formal presentation of the simplest kind of neural network, the **feed-forward network**. A feed-forward network is a multilayer network in which the units are connected with no cycles; the outputs from units in each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers. (In Chapter 9 we'll introduce networks with cycles, called **recurrent neural networks**.)

multi-layer perceptrons
MLP

For historical reasons multilayer networks, especially feedforward networks, are sometimes called **multi-layer perceptrons** (or **MLP**s); this is a technical misnomer, since the units in modern multilayer networks aren't perceptrons (perceptrons are purely linear, but modern networks are made up of units with non-linearities like sigmoids), but at some point the name stuck.

Simple feed-forward networks have three kinds of nodes: input units, hidden units, and output units. Fig. 7.8 shows a picture.

The input units are simply scalar values just as we saw in Fig. 7.2.

hidden layer

fully-connected

The core of the neural network is the **hidden layer** formed of **hidden units**, each of which is a neural unit as described in Section 7.1, taking a weighted sum of its inputs and then applying a non-linearity. In the standard architecture, each layer is **fully-connected**, meaning that each unit in each layer takes as input the outputs from all the units in the previous layer, and there is a link between every pair of units from two adjacent layers. Thus each hidden unit sums over all the input units.

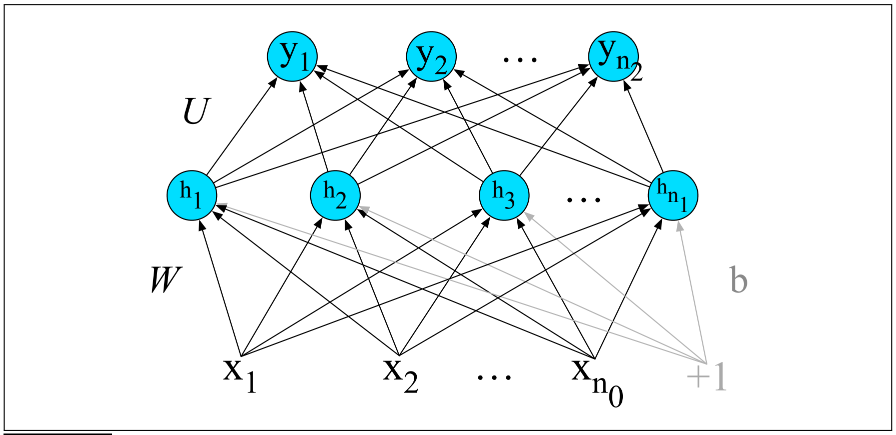


Figure 7.8 A simple 2-layer feed-forward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

Recall that a single hidden unit has parameters w (the weight vector) and b (the bias scalar). We represent the parameters for the entire hidden layer by combining the weight vector $\mathbf{w}_i$ and bias b_i for each unit i into a single weight matrix W and a single bias vector $\mathbf{b}$ for the whole layer (see Fig. 7.8). Each element W_{ij} of the weight matrix W represents the weight of the connection from the ith input unit x_i to the the jth hidden unit h_j.

The advantage of using a single matrix W for the weights of the entire layer is that now that hidden layer computation for a feedforward network can be done very efficiently with simple matrix operations. In fact, the computation only has three steps: multiplying the weight matrix by the input vector x, adding the bias vector b, and applying the activation function g (such as the sigmoid, tanh, or relu activation function defined above).

The output of the hidden layer, the vector h, is thus the following, using the sigmoid function σ:

$$h = \sigma(Wx+b) \tag{7.9}$$

Notice that we're applying the σ function here to a vector, while in Eq. 7.4 it was applied to a scalar. We're thus allowing $\sigma(\cdot)$, and indeed any activation function $g(\cdot)$, to apply to a vector element-wise, so $g[z_1,z_2,z_3] = [g(z_1),g(z_2),g(z_3)]$.

Let's introduce some constants to represent the dimensionalities of these vectors and matrices. We'll refer to the input layer as layer 0 of the network, and use have n_0 represent the number of inputs, so x is a vector of real numbers of dimension n_0, or more formally $x \in \mathbb{R}^{n_0}$. Let's call the hidden layer layer 1 and the output layer layer 2. The hidden layer has dimensionality n_1, so $h \in \mathbb{R}^{n_1}$ and also $b \in \mathbb{R}^{n_1}$ (since each hidden unit can take a different bias value). And the weight matrix W has dimensionality $W \in \mathbb{R}^{n_1 \times n_0}$.

Take a moment to convince yourself that the matrix multiplication in Eq. 7.9 will compute the value of each h_{ij} as $\sum_{i=1}^{n_x} w_{ij}x_i + b_j$.

As we saw in Section 7.2, the resulting value h (for *hidden* but also for *hypothesis*) forms a *representation* of the input. The role of the output layer is to take this new representation h and compute a final output. This output could be a real-valued number, but in many cases the goal of the network is to make some sort of classification decision, and so we will focus on the case of classification.

If we are doing a binary task like sentiment classification, we might have a single

output node, and its value y is the probability of positive versus negative sentiment. If we are doing multinomial classification, such as assigning a part-of-speech tag, we might have one output node for each potential part-of-speech, whose output value is the probability of that part-of-speech, and the values of all the output nodes must sum to one. The output layer thus gives a probability distribution across the output nodes.

Let's see how this happens. Like the hidden layer, the output layer has a weight matrix (let's call it U), but output layers may not t have a bias vector b, so we'll simplify by eliminating the bias vector in this example. The weight matrix is multiplied by its input vector (h) to produce the intermediate output z.

$$z = Uh$$

There are n_2 output nodes, so $z \in \mathbb{R}^{n_2}$, weight matrix U has dimensionality $U \in \mathbb{R}^{n_2 \times n_1}$, and element U_{ij} is the weight from unit j in the hidden layer to unit i in the output layer.

However, z can't be the output of the classifier, since it's a vector of real-valued numbers, while what we need for classification is a vector of probabilities. There is a convenient function for **normalizing** a vector of real values, by which we mean converting it to a vector that encodes a probability distribution (all the numbers lie between 0 and 1 and sum to 1): the **softmax** function that we saw on page 94 of Chapter 5. For a vector z of dimensionality d, the softmax is defined as:

normalizing

softmax

$$\text{softmax}(z_i) \;=\; \frac{e^{z_i}}{\sum_{j=1}^{d} e^{z_j}} \;\; 1 \leq i \leq d \tag{7.10}$$

Thus for example given a vector z=[0.6 1.1 -1.5 1.2 3.2 -1.1], softmax(z) is [0.055 0.090 0.0067 0.10 0.74 0.010].

You may recall that softmax was exactly what is used to create a probability distribution from a vector of real-valued numbers (computed from summing weights times features) in logistic regression in Chapter 5.

That means we can think of a neural network classifier with one hidden layer as building a vector h which is a hidden layer representation of the input, and then running standard logistic regression on the features that the network develops in h. By contrast, in Chapter 5 the features were mainly designed by hand via feature templates. So a neural network is like logistic regression, but (a) with many layers, since a deep neural network is like layer after layer of logistic regression classifiers, and (b) rather than forming the features by feature templates, the prior layers of the network induce the feature representations themselves.

Here are the final equations for a feed-forward network with a single hidden layer, which takes an input vector x, outputs a probability distribution y, and is parameterized by weight matrices W and U and a bias vector b:

$$\begin{aligned} h &= \sigma(Wx+b) \\ z &= Uh \\ y &= \text{softmax}(z) \end{aligned} \tag{7.11}$$

We'll call this network a 2-layer network (we traditionally don't count the input layer when numbering layers, but do count the output layer). So by this terminology logistic regression is a 1-layer network.

Let's now set up some notation to make it easier to talk about deeper networks of depth more than 2. We'll use superscripts in square brackets to mean layer numbers, starting at 0 for the input layer. So $W^{[1]}$ will mean the weight matrix for the

(first) hidden layer, and $b^{[1]}$ will mean the bias vector for the (first) hidden layer. n_j will mean the number of units at layer j. We'll use $g(\cdot)$ to stand for the activation function, which will tend to be ReLU or tanh for intermediate layers and softmax for output layers. We'll use $a^{[i]}$ to mean the output from layer i, and $z^{[i]}$ to mean the combination of weights and biases $W^{[i]}a^{[i-1]} + b^{[i]}$. The 0th layer is for inputs, so the inputs x we'll refer to more generally as $a^{[0]}$.

Thus we'll represent a 3-layer net as follows:

$$\begin{aligned}
z^{[1]} &= W^{[1]}a^{[0]} + b^{[1]} \\
a^{[1]} &= g^{[1]}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= g^{[2]}(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned} \tag{7.12}$$

Note that with this notation, the equations for the computation done at each layer are the same. The algorithm for computing the forward step in an n-layer feed-forward network, given the input vector $a^{[0]}$ is thus simply:

for i **in** 1..n
  $z^{[i]} = W^{[i]}\ a^{[i-1]} + b^{[i]}$
  $a^{[i]} = g^{[i]}(z^{[i]})$
$\hat{y} = a^{[n]}$

The activation functions $g(\cdot)$ are generally different at the final layer. Thus $g^{[2]}$ might be softmax for multinomial classification or sigmoid for binary classification, while ReLU or tanh might be the activation function $g()$ at the internal layers.

7.4 Training Neural Nets

A feedforward neural net is an instance of supervised machine learning in which we know the correct output y for each observation x. What the system produces, via Eq. 7.12, is $\hat{y}$, the system's estimate of the true y. The goal of the training procedure is to learn parameters $W^{[i]}$ and $b[i]$ for each layer i that make $\hat{y}$ for each training observation as close as possible to the true y .

In general, we do all this by drawing on the methods we introduced in Chapter 5 for logistic regression, so the reader should be comfortable with that chapter before proceeding.

First, we'll need a **loss function** that models the distance between the system output and the gold output, and it's common to use the loss used for logistic regression, the **cross-entropy loss**.

Second, to find the parameters that minimize this loss function, we'll use the **gradient descent** optimization algorithm introduced in Chapter 5. There are some differences

Third, gradient descent requires knowing the **gradient** of the loss function, the vector that contains the partial derivative of the loss function with respect to each of the parameters. Here is one part where learning for neural networks is more complex than for logistic logistic regression. In logistic regression, for each observation we could directly compute the derivative of the loss function with respect to an individual w or b. But for neural networks, with millions of parameters in many layers, it's

much harder to see how to compute the partial derivative of some weight in layer 1 when the loss is attached to some much later layer. How do we partial out the loss over all those intermediate layers?

The answer is the algorithm called **error back-propagation** or **reverse differentiation**.

7.4.1 Loss function

cross entropy loss

The **cross entropy loss**, that is used in neural networks is the same one we saw for logistic regression.

In fact, if the neural network is being used as a binary classifier, with the sigmoid at the final layer, the loss function is exactly the same as we saw with logistic regression in Eq. 5.10:

$$L_{CE}(\hat{y},y) = -\log p(y|x) \;=\; -[y\log\hat{y}+(1-y)\log(1-\hat{y})] \tag{7.13}$$

What about if the neural network is being used as a multinomial classifier? Let y be a vector over the C classes representing the true output probability distribution. The cross entropy loss here is

$$L_{CE}(\hat{y},y) = -\sum_{i=1}^{C} y_i \log\hat{y}_i \tag{7.14}$$

We can simplify this equation further. Assume this is a **hard classification** task, meaning that only one class is the correct one, and that there is one output unit in y for each class. If the true class is i, then y is a vector where $y_i = 1$ and $y_j = 0 \;\; \forall j \neq i$. A vector like this, with one value=1 and the rest 0, is called a **one-hot vector**. Now let $\hat{y}$ be the vector output from the network. The sum in Eq. 7.14 will be 0 except for the true class. Hence the cross-entropy loss is simply the log probability of the correct class, and we therefore also call this the **negative log likelihood loss**:

negative log likelihood loss

$$L_{CE}(\hat{y},y) \;=\; -\log\hat{y}_i \tag{7.15}$$

Plugging in the softmax formula from Eq. 7.10, and with K the number of classes:

$$L_{CE}(\hat{y},y) \;=\; -\log\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{7.16}$$

7.4.2 Computing the Gradient

How do we compute the gradient of this loss function? Computing the gradient requires the partial derivative of the loss function with respect to each parameter. For a network with one weight layer and sigmoid output (which is what logistic regression is), we could simply use the derivative of the loss that we used for logistic regression in: Eq. 7.17 (and derived in Section 5.8):

$$\begin{aligned}
\frac{\partial L_{CE}(w,b)}{\partial w_j} &= (\hat{y}-y)\,x_j \\
&= (\sigma(w\cdot x+b)-y)\,x_j
\end{aligned} \tag{7.17}$$

Or for a network with one hidden layer and softmax output, we could use the derivative of the softmax loss from Eq. 5.36:

$$\begin{aligned}\frac{\partial L_{CE}}{\partial w_k} &= (1\{y=k\}-p(y=k|x))x_k \\ &= \left(1\{y=k\}-\frac{e^{w_k\cdot x+b_k}}{\sum_{j=1}^{K}e^{w_j\cdot x+b_j}}\right)x_k \end{aligned} \tag{7.18}$$

But these derivatives only give correct updates for one weight layer: the last one! For deep networks, computing the gradients for each weight is much more complex, since we are computing the derivative with respect to weight parameters that appear all the way back in the very early layers of the network, even though the loss is computed only at the very end of the network.

error backpropagation

The solution to computing this gradient is an algorithm called **error backpropagation** or **backprop** (Rumelhart et al., 1986). While backprop was invented specially for neural networks, it turns out to be the same as a more general procedure called **backward differentiation**, which depends on the notion of **computation graphs**. Let's see how that works in the next subsection.

7.4.3 Computation Graphs

A computation graph is a representation of the process of computing a mathematical expression, in which the computation is broken down into separate operations, each of which is modeled as a node in a graph.

Consider computing the function $L(a,b,c) = c(a+2b)$. If we make each of the component addition and multiplication operations explicit, and add names (d and e) for the intermediate outputs, the resulting series of computations is:

$$\begin{aligned} d &= 2*b \\ e &= a+d \\ L &= c*e \end{aligned}$$

We can now represent this as a graph, with nodes for each operation, and directed edges showing the outputs from each operation as the inputs to the next, as in Fig. 7.9. The simplest use of computation graphs is to compute the value of the function with some given inputs. In the figure, we've assumed the inputs $a=3$, $b=1$, $c=-1$, and we've shown the result of the **forward pass** to compute the result $L(3,1,-1)=10$. In the forward pass of a computation graph, we apply each operation left to right, passing the outputs of each computation as the input to the next node.

7.4.4 Backward differentiation on computation graphs

The importance of the computation graph comes from the **backward pass**, which is used to compute the derivatives that we'll need for the weight update. In this example our goal is to compute the derivative of the output function L with respect to each of the input variables, i.e., $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$. The derivative $\frac{\partial L}{\partial a}$, tells us how much a small change in a affects L.

chain rule

Backwards differentiation makes use of the **chain rule** in calculus. Suppose we are computing the derivative of a composite function $f(x)=u(v(x))$. The derivative

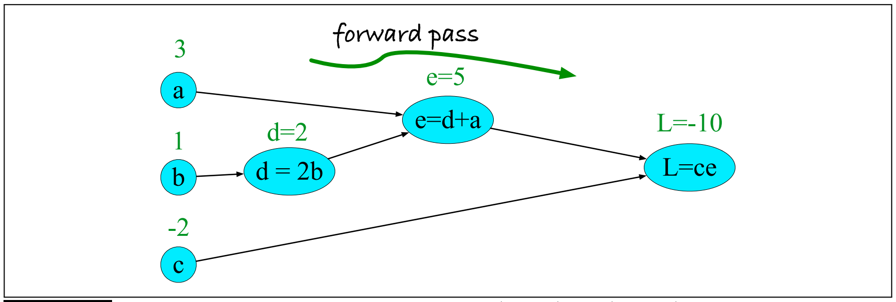


Figure 7.9 Computation graph for the function $L(a,b,c) = c(a+2b)$, with values for input nodes $a = 3$, $b = 1$, $c = -1$, showing the forward pass computation of L.

of $f(x)$ is the derivative of $u(x)$ with respect to $v(x)$ times the derivative of $v(x)$ with respect to x:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx} \tag{7.19}$$

The chain rule extends to more than two functions. If computing the derivative of a composite function $f(x) = u(v(w(x)))$, the derivative of $f(x)$ is:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx} \tag{7.20}$$

Let's now compute the 3 derivatives we need. Since in the computation graph $L = ce$, we can directly compute the derivative $\frac{\partial L}{\partial c}$:

$$\frac{\partial L}{\partial c} = e \tag{7.21}$$

For the other two, we'll need to use the chain rule:

$$\begin{aligned} \frac{\partial L}{\partial a} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} \\ \frac{\partial L}{\partial b} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b} \end{aligned} \tag{7.22}$$

Eq. 7.22 thus requires four intermediate derivatives: $\frac{\partial L}{\partial e}$, $\frac{\partial e}{\partial a}$, $\frac{\partial e}{\partial d}$, and $\frac{\partial d}{\partial b}$, which are as follows (making use of the fact that the derivative of a sum is the sum of the derivatives):

$$\begin{aligned} L = ce &: \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e \\ e = a+d &: \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1 \\ d = 2b &: \quad \frac{\partial d}{\partial b} = 2 \end{aligned} \tag{7.23}$$

In the backward pass, we compute each of these partials along each edge of the graph from right to left, multiplying the necessary partials to result in the final derivative we need. Thus we begin by annotating the final node with $\frac{\partial L}{\partial L} = 1$. Moving to the left, we then compute $\frac{\partial L}{\partial c}$ and $\frac{\partial L}{\partial e}$, and so on, until we have annotated the graph

all the way to the input variables. The forward pass conveniently already will have computed the values of the forward intermediate variables we need (like d and e) to compute these derivatives. Fig. 7.10 shows the backward pass. At each node we need to compute the local partial derivative with respect to the parent, multiply it by the partial derivative that is being passed down from the parent, and then pass it to the child.

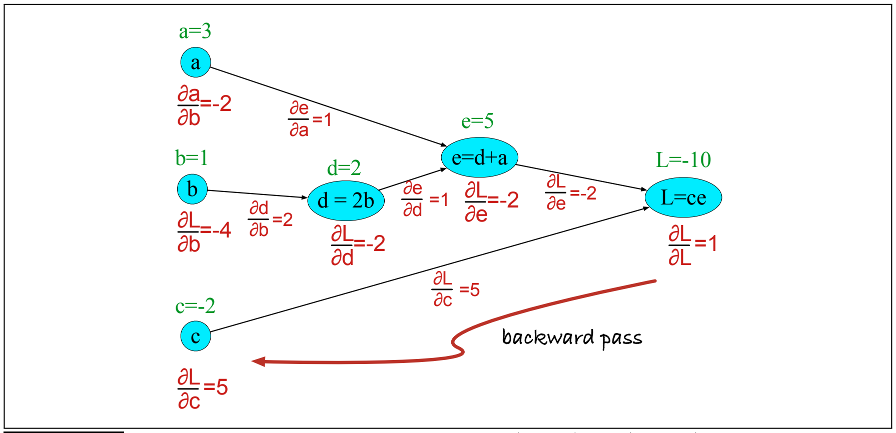


Figure 7.10 Computation graph for the function $L(a,b,c) = c(a+2b)$, showing the backward pass computation of $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$.

Of course computation graphs for real neural networks are much more complex. Fig. 7.11 shows a sample computation graph for a 2-layer neural network with $n_0 = 2$, $n_1 = 2$, and $n_2 = 1$, assuming binary classification and hence using a sigmoid output unit for simplicity. The weights that need updating (those for which we need to know the partial derivative of the loss function) are shown in orange.

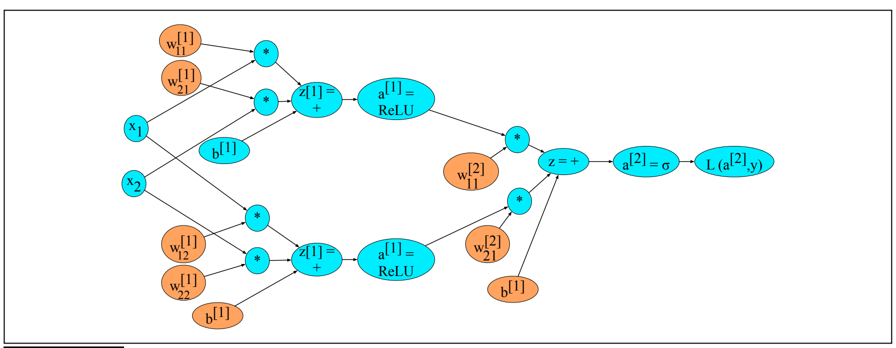


Figure 7.11 Sample computation graph for a simple 2-layer neural net (= 1 hidden layer) with two input dimensions and 2 hidden dimensions.

In order to do the backward pass, we'll need to know the derivatives of all the functions in the graph. We already saw in Section 5.8 the derivative of the sigmoid σ:

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-z) \tag{7.24}$$

We'll also need the derivatives of each of the other activation functions. The derivative of tanh is:

$$\frac{d\tanh(z)}{dz} = 1-\tanh^2(z) \tag{7.25}$$

The derivative of the ReLU is

$$\frac{d\,\text{ReLU}(z)}{dz} = \begin{cases} 0 & for \;\; x < 0 \\ 1 & for \;\; x \geq 0 \end{cases} \tag{7.26}$$

7.4.5 More details on learning

Optimization in neural networks is a non-convex optimization problem, more complex than for logistic regression, and for that and other reasons there are many best practices for successful learning.

For logistic regression we can initialize gradient descent with all the weights and biases having the value 0. In neural networks, by contrast, we need to initialize the weights with small random numbers. It's also helpful to normalize the input values to have 0 mean and unit variance.

Various forms of regularization are used to prevent overfitting. One of the most important is **dropout**: randomly dropping some units and their connections from the network during training (Hinton et al. 2012, Srivastava et al. 2014).

dropout

hyperparameter

Hyperparameter tuning is also important. The parameters of a neural network are the weights W and biases b; those are learned by gradient descent. The hyperparameters are things that are set by the algorithm designer and not learned in the same way, although they must be tuned. Hyperparameters include the learning rate η, the minibatch size, the model architecture (the number of layers, the number of hidden nodes per layer, the choice of activation functions), how to regularize, and so on. Gradient descent itself also has many architectural variants such as Adam (Kingma and Ba, 2015).

Finally, most modern neural networks are built using computation graph formalisms that make all the work of gradient computation and parallelization onto vector-based GPUs (Graphic Processing Units) very easy and natural. Pytorch (Paszke et al., 2017) and TensorFlow (Abadi et al., 2015) are two of the most popular. The interested reader should consult a neural network textbook for further details; some suggestions are at the end of the chapter.

7.5 Neural Language Models

As our first application of neural networks, let's consider **language modeling**: predicting upcoming words from prior word context.

Neural net-based language models turn out to have many advantages over the n-gram language models of Chapter 3. Among these are that neural language models don't need smoothing, they can handle much longer histories, and they can generalize over contexts of similar words. For a training set of a given size, a neural language model has much higher predictive accuracy than an n-gram language model Furthermore, neural language models underlie many of the models we'll introduce for tasks like machine translation, dialog, and language generation.

On the other hand, there is a cost for this improved performance: neural net language models are strikingly slower to train than traditional language models, and so for many tasks an n-gram language model is still the right tool.

In this chapter we'll describe simple feedforward neural language models, first introduced by Bengio et al. (2003). Modern neural language models are generally not feedforward but recurrent, using the technology that we will introduce in Chapter 9.

A feedforward neural LM is a standard feedforward network that takes as input at time t a representation of some number of previous words (w_{t-1}, w_{t-2}, etc) and outputs a probability distribution over possible next words. Thus—like the n-gram LM—the feedforward neural LM approximates the probability of a word given the entire prior context $P(w_t|w_1^{t-1})$ by approximating based on the N previous words:

$$P(w_t|w_1^{t-1}) \approx P(w_t|w_{t-N+1}^{t-1}) \tag{7.27}$$

In the following examples we'll use a 4-gram example, so we'll show a net to estimate the probability $P(w_t = i|w_{t-1}, w_{t-2}, w_{t-3})$.

7.5.1 Embeddings

In neural language models, the prior context is represented by embeddings of the previous words. Representing the prior context as embeddings, rather than by exact words as used in n-gram language models, allows neural language models to generalize to unseen data much better than n-gram language models. For example, suppose we've seen this sentence in training:

I have to make sure when I get home to feed the cat.

but we've never seen the word "dog" after the words "feed the". In our test set we are trying to predict what comes after the prefix "I forgot when I got home to feed the".

An n-gram language model will predict "cat", but not "dog". But a neural LM, which can make use of the fact that "cat" and "dog" have similar embeddings, will be able to assign a reasonably high probability to "dog" as well as "cat", merely because they have similar vectors.

Let's see how this works in practice. Let's assume we have an embedding dictionary E that gives us, for each word in our vocabulary V, the embedding for that word, perhaps precomputed by an algorithm like word2vec from Chapter 6.

Fig. 7.12 shows a sketch of this simplified FFNNLM with N=3; we have a moving window at time t with an embedding vector representing each of the 3 previous words (words w_{t-1}, w_{t-2}, and w_{t-3}). These 3 vectors are concatenated together to produce x, the input layer of a neural network whose output is a softmax with a probability distribution over words. Thus y_{42}, the value of output node 42 is the probability of the next word w_t being V_{42}, the vocabulary word with index 42.

The model shown in Fig. 7.12 is quite sufficient, assuming we learn the embeddings separately by a method like the **word2vec** methods of Chapter 6. The method of using another algorithm to learn the embedding representations we use for input words is called **pretraining**. If those pretrained embeddings are sufficient for your purposes, then this is all you need.

pretraining

However, often we'd like to learn the embeddings simultaneously with training the network. This is true when whatever task the network is designed for (sentiment classification, or translation, or parsing) places strong constraints on what makes a good representation.

Let's therefore show an architecture that allows the embeddings to be learned. To do this, we'll add an extra layer to the network, and propagate the error all the way back to the embedding vectors, starting with embeddings with random values and slowly moving toward sensible representations.

For this to work at the input layer, instead of pre-trained embeddings, we're going to represent each of the N previous words as a one-hot vector of length $|V|$, i.e., with one dimension for each word in the vocabulary. A **one-hot vector** is a vector

one-hot vector

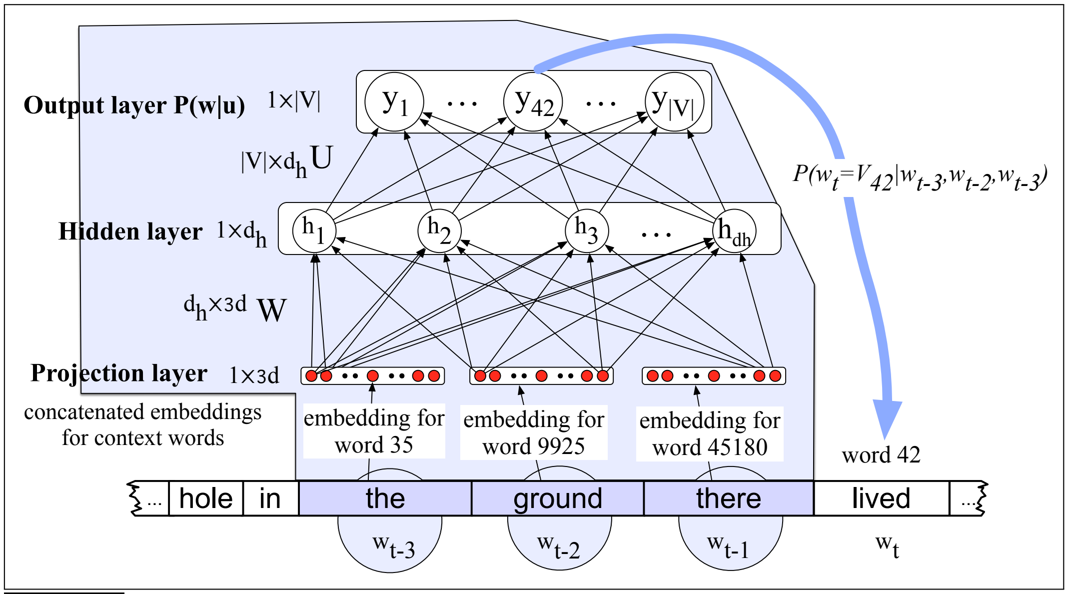


Figure 7.12 A simplified view of a feedforward neural language model moving through a text. At each timestep t the network takes the 3 context words, converts each to a d-dimensional embeddings, and concatenates the 3 embeddings together to get the $1 \times Nd$ unit input layer x for the network. These units are multiplied by a weight matrix W and bias vector b and then an activation function to produce a hidden layer h, which is then multiplied by another weight matrix U. (For graphic simplicity we don't show b in this and future pictures). Finally, a softmax output layer predicts at each node i the probability that the next word w_t will be vocabulary word V_i. (This picture is simplified because it assumes we just look up in an embedding dictionary E the d-dimensional embedding vector for each word, precomputed by an algorithm like word2vec.)

that has one element equal to 1—in the dimension corresponding to that word's index in the vocabulary— while all the other elements are set to zero.

Thus in a one-hot representation for the word "toothpaste", supposing it happens to have index 5 in the vocabulary, x_5 is one and and $x_i = 0\ \forall i \neq 5$, as shown here:

```
[0 0 0 0 1 0 0 ... 0 0 0 0]
 1 2 3 4 5 6 7 ...   ... |V|
```

Fig. 7.13 shows the additional layers needed to learn the embeddings during LM training. Here the N=3 context words are represented as 3 one-hot vectors, fully connected to the embedding layer via 3 instantiations of the E embedding matrix. Note that we don't want to learn separate weight matrices for mapping each of the 3 previous words to the projection layer, we want one single embedding dictionary E that's shared among these three. That's because over time, many different words will appear as w_{t-2} or w_{t-1}, and we'd like to just represent each word with one vector, whichever context position it appears in. The embedding weight matrix E thus has a row for each word, each a vector of d dimensions, and hence has dimensionality $V \times d$.

Let's walk through the forward pass of Fig. 7.13.

1. **Select three embeddings from E**: Given the three previous words, we look up their indices, create 3 one-hot vectors, and then multiply each by the embedding matrix E. Consider w_{t-3}. The one-hot vector for 'the' is (index 35) is multiplied by the embedding matrix E, to give the first part of the first hidden layer, called the **projection layer**. Since each row of the input matrix E is just an embedding for a word, and the input is a one-hot columnvector x_i for word

projection layer

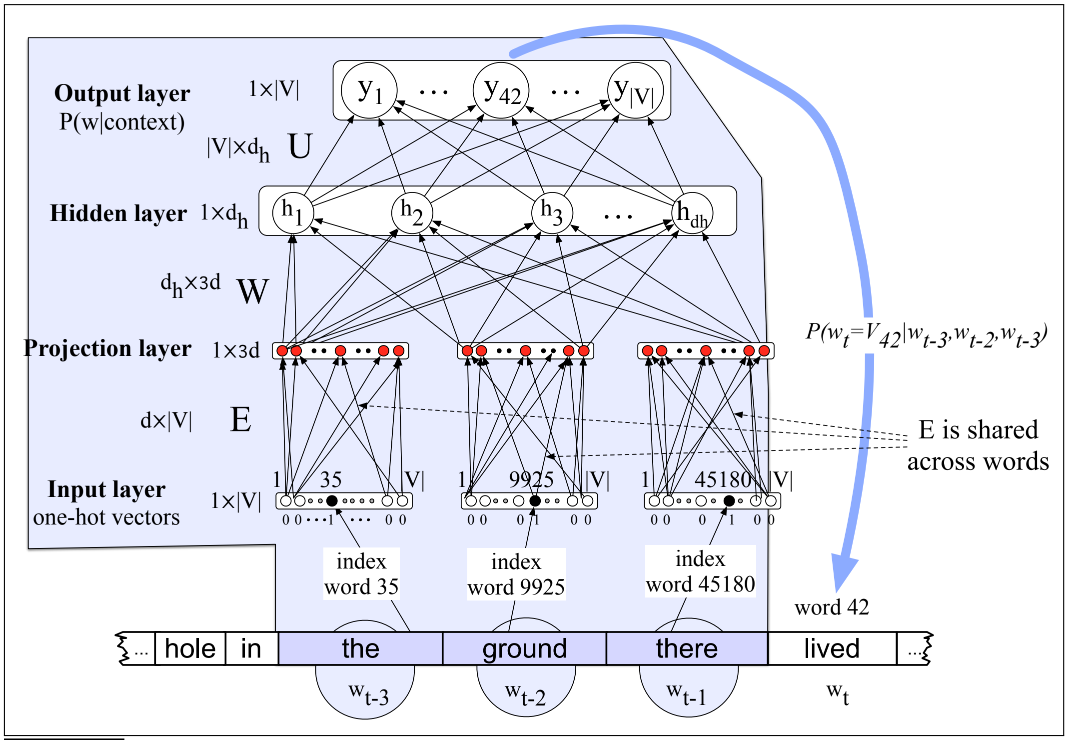


Figure 7.13 learning all the way back to embeddings. notice that the embedding matrix E is shared among the 3 context words.

V_i, the projection layer for input w will be $Ex_i = e_i$, the embedding for word i. We now concatenate the three embeddings for the context words.

2. **Multiply by W**: We now multiply by W (and add b) and pass through the rectified linear (or other) activation function to get the hidden layer h.
3. **Multiply by U**: h is now multiplied by U
4. **Apply softmax**: After the softmax, each node i in the output layer estimates the probability $P(w_t = i|w_{t-1}, w_{t-2}, w_{t-3})$

In summary, if we use e to represent the projection layer, formed by concatenating the 3 embedding for the three context vectors, the equations for a neural language model become:

$$e = (Ex_1, Ex_2, ..., Ex) \tag{7.28}$$
$$h = \sigma(We + b) \tag{7.29}$$
$$z = Uh \tag{7.30}$$
$$y = \text{softmax}(z) \tag{7.31}$$

7.5.2 Training the neural language model

To train the model, i.e. to set all the parameters $\theta = E, W, U, b$, we do gradient descent (Fig. 5.5), using error back propagation on the computation graph to compute the gradient. Training thus not only sets the weights W and U of the network, but also as we're predicting upcoming words, we're learning the embeddings E for each words that best predict upcoming words.

Generally training proceedings by taking as input a very long text, concatenating all the sentences, start with random weights, and then iteratively moving through the text predicting each word w_t. At each word w_t, the cross-entropy (negative log likelihood) loss is:

$$L = -\log p(w_t|w_{t-1},...,w_{t-n+1}) \tag{7.32}$$

The gradient is for this loss is then:

$$\theta_{t+1} = \theta_t - \eta \frac{\partial -\log p(w_t|w_{t-1},...,w_{t-n+1})}{\partial \theta} \tag{7.33}$$

This gradient can be computed in any standard neural network framework which will then backpropagate through U, W, b, E.

Training the parameters to minimize loss will result both in an algorithm for language modeling (a word predictor) but also a new set of embeddings E that can be used as word representations for other tasks.

7.6 Summary

- Neural networks are built out of **neural units**, originally inspired by human neurons but now simple an abstract computational device.
- Each neural unit multiplies input values by a weight vector, adds a bias, and then applies a non-linear activation function like sigmoid, tanh, or rectified linear.
- In a **fully-connected**, **feedforward** network, each unit in layer i is connected to each unit in layer $i+1$, and there are no cycles.
- The power of neural networks comes from the ability of early layers to learn representations that can be utilized by later layers in the network.
- Neural networks are trained by optimization algorithms like **gradient descent**.
- **Error back propagation**, backward differentiation on a **computation graph**, is used to compute the gradients of the loss function for a network.
- **Neural language models** use a neural network as a probabilistic classifier, to compute the probability of the next word given the previous n words.
- Neural language models can use pretrained **embeddings**, or can learn embeddings from scratch in the process of language modeling.

Bibliographical and Historical Notes

The origins of neural networks lie in the 1940s **McCulloch-Pitts neuron** (McCulloch and Pitts, 1943), a simplified model of the human neuron as a kind of computing element that could be described in terms of propositional logic. By the late 1950s and early 1960s, a number of labs (including Frank Rosenblatt at Cornell and Bernard Widrow at Stanford) developed research into neural networks; this phase saw the development of the perceptron (Rosenblatt, 1958), and the transformation of the threshold into a bias, a notation we still use (Widrow and Hoff, 1960).

The field of neural networks declined after it was shown that a single perceptron unit was unable to model functions as simple as XOR (Minsky and Papert, 1969). While some small amount of work continued during the next two decades, a major revival for the field didn't come until the 1980s, when practical tools for building deeper networks like error back propagation became widespread (Rumelhart et al., 1986). During the 1980s a wide variety of neural network and related architectures were developed, particularly for applications in psychology and cognitive science (Rumelhart and McClelland 1986b, McClelland and Elman 1986, Rumelhart and McClelland 1986a,Elman 1990), for which the term **connectionist** or **parallel distributed processing** was often used (Feldman and Ballard 1982, Smolensky 1988). Many of the principles and techniques developed in this period are foundational to modern work, including the ideas of distributed representations (Hinton, 1986), recurrent networks (Elman, 1990), and the use of tensors for compositionality (Smolensky, 1990).

connectionist

By the 1990s larger neural networks began to be applied to many practical language processing tasks as well, like handwriting recognition (LeCun et al. 1989, LeCun et al. 1990) and speech recognition (Morgan and Bourlard 1989, Morgan and Bourlard 1990). By the early 2000s, improvements in computer hardware and advances in optimization and training techniques made it possible to train even larger and deeper networks, leading to the modern term **deep learning** (Hinton et al. 2006, Bengio et al. 2007). We cover more related history in Chapter 9.

There are a number of excellent books on the subject. Goldberg (2017) has a superb and comprehensive coverage of neural networks for natural language processing. For neural networks in general see Goodfellow et al. (2016) and Nielsen (2015).

CHAPTER

8 Part-of-Speech Tagging

Dionysius Thrax of Alexandria (*c.* 100 B.C.), or perhaps someone else (it was a long time ago), wrote a grammatical sketch of Greek (a "*technē*") that summarized the linguistic knowledge of his day. This work is the source of an astonishing proportion of modern linguistic vocabulary, including words like *syntax*, *diphthong*, *clitic*, and *analogy*. Also included are a description of eight **parts-of-speech**: noun, verb, pronoun, preposition, adverb, conjunction, participle, and article. Although earlier scholars (including Aristotle as well as the Stoics) had their own lists of parts-of-speech, it was Thrax's set of eight that became the basis for practically all subsequent part-of-speech descriptions of most European languages for the next 2000 years.

parts-of-speech

Schoolhouse Rock was a series of popular animated educational television clips from the 1970s. Its Grammar Rock sequence included songs about exactly 8 parts-of-speech, including the late great Bob Dorough's *Conjunction Junction*:

> *Conjunction Junction, what's your function?*
> *Hooking up words and phrases and clauses...*

Although the list of 8 was slightly modified from Thrax's original, the astonishing durability of the parts-of-speech through two millenia is an indicator of both the importance and the transparency of their role in human language.[1]

POS

Parts-of-speech (also known as **POS**, **word classes**, or **syntactic categories**) are useful because they reveal a lot about a word and its neighbors. Knowing whether a word is a **noun** or a **verb** tells us about likely neighboring words (nouns are preceded by determiners and adjectives, verbs by nouns) and syntactic structure word (nouns are generally part of noun phrases), making part-of-speech tagging a key aspect of parsing (Chapter 11). Parts of speech are useful features for labeling **named entities** like people or organizations in **information extraction** (Chapter 17), or for coreference resolution (Chapter 21). A word's part-of-speech can even play a role in speech recognition or synthesis, e.g., the word *content* is pronounced *CONtent* when it is a noun and *conTENT* when it is an adjective.

This chapter introduces parts-of-speech, and then introduces two algorithms for **part-of-speech tagging**, the task of assigning parts-of-speech to words. One is generative— Hidden Markov Model (HMM)—and one is discriminative—the Maximum Entropy Markov Model (MEMM). Chapter 9 then introduces a third algorithm based on the recurrent neural network (RNN). All three have roughly equal performance but, as we'll see, have different tradeoffs.

8.1 (Mostly) English Word Classes

Until now we have been using part-of-speech terms like **noun** and **verb** rather freely. In this section we give a more complete definition of these and other classes. While word classes do have semantic tendencies—adjectives, for example, often

[1] Nonetheless, eight isn't very many and, as we'll see, recent tagsets have more.

describe *properties* and nouns *people*— parts-of-speech are traditionally defined instead based on syntactic and morphological function, grouping words that have similar neighboring words (their **distributional** properties) or take similar affixes (their morphological properties).

closed class
open class

Parts-of-speech can be divided into two broad supercategories: **closed class** types and **open class** types. Closed classes are those with relatively fixed membership, such as prepositions—new prepositions are rarely coined. By contrast, nouns and verbs are open classes—new nouns and verbs like *iPhone* or *to fax* are continually being created or borrowed. Any given speaker or corpus may have different open class words, but all speakers of a language, and sufficiently large corpora, likely share the set of closed class words. Closed class words are generally **function words** like *of*, *it*, *and*, or *you*, which tend to be very short, occur frequently, and often have structuring uses in grammar.

function word

Four major open classes occur in the languages of the world: **nouns**, **verbs**, **adjectives**, and **adverbs**. English has all four, although not every language does. The syntactic class **noun** includes the words for most people, places, or things, but others as well. Nouns include concrete terms like *ship* and *chair*, abstractions like *bandwidth* and *relationship*, and verb-like terms like *pacing* as in *His pacing to and fro became quite annoying*. What defines a noun in English, then, are things like its ability to occur with determiners (*a goat, its bandwidth, Plato's Republic*), to take possessives (*IBM's annual revenue*), and for most but not all nouns to occur in the plural form (*goats, abaci*).

noun

proper noun

Open class nouns fall into two classes. **Proper nouns**, like *Regina*, *Colorado*, and *IBM*, are names of specific persons or entities. In English, they generally aren't preceded by articles (e.g., *the book is upstairs*, but *Regina is upstairs*). In written English, proper nouns are usually capitalized. The other class, **common nouns**, are divided in many languages, including English, into **count nouns** and **mass nouns**. Count nouns allow grammatical enumeration, occurring in both the singular and plural (*goat/goats, relationship/relationships*) and they can be counted (*one goat, two goats*). Mass nouns are used when something is conceptualized as a homogeneous group. So words like *snow, salt*, and *communism* are not counted (i.e., **two snows* or **two communisms*). Mass nouns can also appear without articles where singular count nouns cannot (*Snow is white* but not **Goat is white*).

common noun
count noun
mass noun

verb

Verbs refer to actions and processes, including main verbs like *draw*, *provide*, and *go*. English verbs have inflections (non-third-person-sg (*eat*), third-person-sg (*eats*), progressive (*eating*), past participle (*eaten*)). While many researchers believe that all human languages have the categories of noun and verb, others have argued that some languages, such as Riau Indonesian and Tongan, don't even make this distinction (Broschart 1997; Evans 2000; Gil 2000) .

adjective

The third open class English form is **adjectives**, a class that includes many terms for properties or qualities. Most languages have adjectives for the concepts of color (*white*, *black*), age (*old*, *young*), and value (*good*, *bad*), but there are languages without adjectives. In Korean, for example, the words corresponding to English adjectives act as a subclass of verbs, so what is in English an adjective "beautiful" acts in Korean like a verb meaning "to be beautiful".

adverb

The final open class form, **adverbs**, is rather a hodge-podge in both form and meaning. In the following all the italicized words are adverbs:

Actually, I ran *home extremely quickly yesterday*

What coherence the class has semantically may be solely that each of these words can be viewed as modifying something (often verbs, hence the name "ad-

locative degree manner temporal

verb”, but also other adverbs and entire verb phrases). **Directional adverbs** or **locative adverbs** (*home*, *here*, *downhill*) specify the direction or location of some action; **degree adverbs** (*extremely*, *very*, *somewhat*) specify the extent of some action, process, or property; **manner adverbs** (*slowly*, *slinkily*, *delicately*) describe the manner of some action or process; and **temporal adverbs** describe the time that some action or event took place (*yesterday*, *Monday*). Because of the heterogeneous nature of this class, some adverbs (e.g., temporal adverbs like *Monday*) are tagged in some tagging schemes as nouns.

The closed classes differ more from language to language than do the open classes. Some of the important closed classes in English include:

> **prepositions:** on, under, over, near, by, at, from, to, with
> **particles:** up, down, on, off, in, out, at, by
> **determiners:** a, an, the
> **conjunctions:** and, but, or, as, if, when
> **pronouns:** she, who, I, others
> **auxiliary verbs:** can, may, should, are
> **numerals:** one, two, three, first, second, third

preposition particle

Prepositions occur before noun phrases. Semantically they often indicate spatial or temporal relations, whether literal (*on it*, *before then*, *by the house*) or metaphorical (*on time*, *with gusto*, *beside herself*), but often indicate other relations as well, like marking the agent in (*Hamlet was written by Shakespeare*, A **particle** resembles a preposition or an adverb and is used in combination with a verb. Particles often have extended meanings that aren’t quite the same as the prepositions they resemble, as in the particle *over* in *she turned the paper over*.

phrasal verb

A verb and a particle that act as a single syntactic and/or semantic unit are called a **phrasal verb**. The meaning of phrasal verbs is often problematically **non-compositional**—not predictable from the distinct meanings of the verb and the particle. Thus, *turn down* means something like ‘reject’, *rule out* ‘eliminate’, *find out* ‘discover’, and *go on* ‘continue’.

determiner article

A closed class that occurs with nouns, often marking the beginning of a noun phrase, is the **determiner**. One small subtype of determiners is the **article**: English has three articles: *a*, *an*, and *the*. Other determiners include *this* and *that* (*this chapter*, *that page*). *A* and *an* mark a noun phrase as indefinite, while *the* can mark it as definite; definiteness is a discourse property (Chapter 22). Articles are quite frequent in English; indeed, *the* is the most frequently occurring word in most corpora of written English, and *a* and *an* are generally right behind.

conjunctions complementizer

Conjunctions join two phrases, clauses, or sentences. Coordinating conjunctions like *and*, *or*, and *but* join two elements of equal status. Subordinating conjunctions are used when one of the elements has some embedded status. For example, *that* in *“I thought that you might like some milk”* is a subordinating conjunction that links the main clause *I thought* with the subordinate clause *you might like some milk*. This clause is called subordinate because this entire clause is the “content” of the main verb *thought*. Subordinating conjunctions like *that* which link a verb to its argument in this way are also called **complementizers**.

pronoun personal possessive wh

Pronouns are forms that often act as a kind of shorthand for referring to some noun phrase or entity or event. **Personal pronouns** refer to persons or entities (*you*, *she*, *I*, *it*, *me*, etc.). **Possessive pronouns** are forms of personal pronouns that indicate either actual possession or more often just an abstract relation between the person and some object (*my, your, his, her, its, one’s, our, their*). **Wh-pronouns** (*what, who, whom, whoever*) are used in certain question forms, or may also act as

complementizers (*Frida, who married Diego...*).

auxiliary A closed class subtype of English verbs are the **auxiliary** verbs. Cross-linguistically, auxiliaries mark semantic features of a main verb: whether an action takes place in the present, past, or future (tense), whether it is completed (aspect), whether it is negated (polarity), and whether an action is necessary, possible, suggested, or desired (mood). English auxiliaries include the **copula** verb *be*, the two verbs *do* and *have*, along with their inflected forms, as well as a class of **modal verbs**. *Be* is called a copula because it connects subjects with certain kinds of predicate nominals and adjectives (*He is a duck*). The verb *have* can mark the perfect tenses (*I have gone*, *I had gone*), and *be* is used as part of the passive (*We were robbed*) or progressive (*We are leaving*) constructions. Modals are used to mark the mood associated with the event depicted by the main verb: *can* indicates ability or possibility, *may* permission or possibility, *must* necessity. There is also a modal use of *have* (e.g., *I have to go*).

copula

modal

English also has many words of more or less unique function, including **interjections** (*oh, hey, alas, uh, um*), **negatives** (*no, not*), **politeness markers** (*please, thank you*), **greetings** (*hello, goodbye*), and the existential **there** (*there are two on the table*) among others. These classes may be distinguished or lumped together as interjections or adverbs depending on the purpose of the labeling.

interjection

negative

8.2 The Penn Treebank Part-of-Speech Tagset

An important tagset for English is the 45-tag Penn Treebank tagset (Marcus et al., 1993), shown in Fig. 8.1, which has been used to label many corpora. In such labelings, parts-of-speech are generally represented by placing the tag after each word, delimited by a slash:

Tag	Description	Example	Tag	Description	Example	Tag	Description	Example
CC	coordinating conjunction	*and, but, or*	PDT	predeterminer	*all, both*	VBP	verb non-3sg present	*eat*
CD	cardinal number	*one, two*	POS	possessive ending	*'s*	VBZ	verb 3sg pres	*eats*
DT	determiner	*a, the*	PRP	personal pronoun	*I, you, he*	WDT	wh-determ.	*which, that*
EX	existential 'there'	*there*	PRP$	possess. pronoun	*your, one's*	WP	wh-pronoun	*what, who*
FW	foreign word	*mea culpa*	RB	adverb	*quickly*	WP$	wh-possess.	*whose*
IN	preposition/ subordin-conj	*of, in, by*	RBR	comparative adverb	*faster*	WRB	wh-adverb	*how, where*
JJ	adjective	*yellow*	RBS	superlatv. adverb	*fastest*	$	dollar sign	*$*
JJR	comparative adj	*bigger*	RP	particle	*up, off*	#	pound sign	*#*
JJS	superlative adj	*wildest*	SYM	symbol	*+,%, &*	“	left quote	‘ or “
LS	list item marker	*1, 2, One*	TO	“to”	*to*	”	right quote	’ or ”
MD	modal	*can, should*	UH	interjection	*ah, oops*	(	left paren	[, (, {, <
NN	sing or mass noun	*llama*	VB	verb base form	*eat*	)	right paren	],), }, >
NNS	noun, plural	*llamas*	VBD	verb past tense	*ate*	,	comma	,
NNP	proper noun, sing.	*IBM*	VBG	verb gerund	*eating*	.	sent-end punc	. ! ?
NNPS	proper noun, plu.	*Carolinas*	VBN	verb past part.	*eaten*	:	sent-mid punc	: ; ... – -

Figure 8.1 Penn Treebank part-of-speech tags (including punctuation).

(8.1) The/DT grand/JJ jury/NN commented/VBD on/IN a/DT number/NN of/IN other/JJ topics/NNS ./.

(8.2) **There/EX** are/VBP 70/CD children/NNS **there/RB**

(8.3) Preliminary/JJ findings/NNS were/VBD **reported/VBN** in/IN today/NN **'s/POS** New/NNP England/NNP Journal/NNP of/IN Medicine/NNP ./.

Example (8.1) shows the determiners *the* and *a*, the adjectives *grand* and *other*, the common nouns *jury*, *number*, and *topics*, and the past tense verb *commented*. Example (8.2) shows the use of the EX tag to mark the existential *there* construction in English, and, for comparison, another use of *there* which is tagged as an adverb (RB). Example (8.3) shows the segmentation of the possessive morpheme *'s* a passive construction, 'were reported', in which *reported* is marked as a past participle (VBN). Note that since *New England Journal of Medicine* is a proper noun, the Treebank tagging chooses to mark each noun in it separately as NNP, including *journal* and *medicine*, which might otherwise be labeled as common nouns (NN).

Corpora labeled with parts-of-speech are crucial training (and testing) sets for statistical tagging algorithms. Three main tagged corpora are consistently used for training and testing part-of-speech taggers for English. The **Brown** corpus is a million words of samples from 500 written texts from different genres published in the United States in 1961. The **WSJ** corpus contains a million words published in the Wall Street Journal in 1989. The **Switchboard** corpus consists of 2 million words of telephone conversations collected in 1990-1991. The corpora were created by running an automatic part-of-speech tagger on the texts and then human annotators hand-corrected each tag.

Brown

WSJ

Switchboard

There are some minor differences in the tagsets used by the corpora. For example in the WSJ and Brown corpora, the single Penn tag TO is used for both the infinitive *to* (*I like to race*) and the preposition *to* (*go to the store*), while in Switchboard the tag TO is reserved for the infinitive use of *to* and the preposition is tagged IN:

Well/UH ,/, I/PRP ,/, I/PRP want/VBP **to/TO** go/VB **to/IN** a/DT restaurant/NN

Finally, there are some idiosyncracies inherent in any tagset. For example, because the Penn 45 tags were collapsed from a larger 87-tag tagset, the **original Brown tagset**, some potential useful distinctions were lost. The Penn tagset was designed for a treebank in which sentences were parsed, and so it leaves off syntactic information recoverable from the parse tree. Thus for example the Penn tag IN is used for both subordinating conjunctions like *if, when, unless, after*:

after/IN spending/VBG a/DT day/NN at/IN the/DT beach/NN

and prepositions like *in, on, after*:

after/IN sunrise/NN

Words are generally tokenized before tagging. The Penn Treebank and the British National Corpus split contractions and the *'s*-genitive from their stems:[2]

would/MD n't/RB
children/NNS 's/POS

The Treebank tagset assumes that tokenization of multipart words like *New York* is done at whitespace, thus tagging. *a New York City firm* as *a/DT New/NNP York/NNP City/NNP firm/NN*.

Another commonly used tagset, the Universal POS tag set of the Universal Dependencies project (Nivre et al., 2016a), is used when building systems that can tag many languages. See Section 8.7.

[2] Indeed, the Treebank tag POS is used only for *'s*, which must be segmented in tokenization.

8.3 Part-of-Speech Tagging

Part-of-speech tagging is the process of assigning a part-of-speech marker to each word in an input text.[3] The input to a tagging algorithm is a sequence of (tokenized) words and a tagset, and the output is a sequence of tags, one per token.

Tagging is a **disambiguation** task; words are **ambiguous** —have more than one possible part-of-speech—and the goal is to find the correct tag for the situation. For example, *book* can be a verb (*book that flight*) or a noun (*hand me that book*). *That* can be a determiner (*Does that flight serve dinner*) or a complementizer (*I thought that your flight was earlier*). The goal of POS-tagging is to **resolve** these ambiguities, choosing the proper tag for the context. How common is tag ambiguity? Fig. 8.2 shows that most word types (80-86%) are unambiguous (*Janet* is always NNP, *funniest* JJS, and *hesitantly* RB). But the ambiguous words, though accounting for only 14-15% of the vocabulary, are very common words, and hence 55-67% of word tokens in running text are ambiguous.[4]

		WSJ	Brown
Types:			
Unambiguous	(1 tag)	44,432 (**86%**)	45,799 (**85%**)
Ambiguous	(2+ tags)	7,025 (**14%**)	8,050 (**15%**)
Tokens:			
Unambiguous	(1 tag)	577,421 (**45%**)	384,349 (**33%**)
Ambiguous	(2+ tags)	711,780 (**55%**)	786,646 (**67%**)

Figure 8.2 Tag ambiguity for word types in Brown and WSJ, using Treebank-3 (45-tag) tagging. Punctuation were treated as words, and words were kept in their original case.

Some of the most ambiguous frequent words are *that*, *back*, *down*, *put* and *set*; here are some examples of the 6 different parts-of-speech for the word *back*:

earnings growth took a **back/JJ** seat
a small building in the **back/NN**
a clear majority of senators **back/VBP** the bill
Dave began to **back/VB** toward the door
enable the country to buy **back/RP** about debt
I was twenty-one **back/RB** then

Nonetheless, many words are easy to disambiguate, because their different tags aren't equally likely. For example, *a* can be a determiner or the letter *a*, but the determiner sense is much more likely. This idea suggests a simplistic **baseline** algorithm for part-of-speech tagging: given an ambiguous word, choose the tag which is **most frequent** in the training corpus. This is a key concept:

> **Most Frequent Class Baseline:** Always compare a classifier against a baseline at least as good as the most frequent class baseline (assigning each token to the class it occurred in most often in the training set).

How good is this baseline? A standard way to measure the performance of part-of-speech taggers is **accuracy**: the percentage of tags correctly labeled (matching

[3] Tags are also applied to punctuation, so assumes tokenzing of commas, quotation marks, etc., and disambiguating end-of-sentence periods from periods inside words (*e.g.*, *etc.*).

[4] Note the large differences across the two genres, especially in token frequency. Tags in the WSJ corpus are less ambiguous; its focus on financial news leads to a more limited distribution of word usages than the diverse genres of the Brown corpus.

human labels on a test set). If we train on the WSJ training corpus and test on sections 22-24 of the same corpus the most-frequent-tag baseline achieves an accuracy of 92.34%. By contrast, the state of the art in part-of-speech tagging on this dataset is around 97% tag accuracy, a performance that is achievable by most algorithms (HMMs, MEMMs, neural networks, rule-based algorithms). See Section 8.7 on other languages and genres.

8.4 HMM Part-of-Speech Tagging

sequence model

In this section we introduce the use of the Hidden Markov Model for part-of-speech tagging. The HMM is a **sequence model**. A sequence model or **sequence classifier** is a model whose job is to assign a label or class to each unit in a sequence, thus mapping a sequence of observations to a sequence of labels. An HMM is a probabilistic sequence model: given a sequence of units (words, letters, morphemes, sentences, whatever), it computes a probability distribution over possible sequences of labels and chooses the best label sequence.

8.4.1 Markov Chains

Markov chain

The HMM is based on augmenting the Markov chain. A **Markov chain** is a model that tells us something about the probabilities of sequences of random variables, *states*, each of which can take on values from some set. These sets can be words, or tags, or symbols representing anything, for example the weather. A Markov chain makes a very strong assumption that if we want to predict the future in the sequence, all that matters is the current state. All the states before the current state have no impact on the future except via the current state. It's as if to predict tomorrow's weather you could examine today's weather but you weren't allowed to look at yesterday's weather.

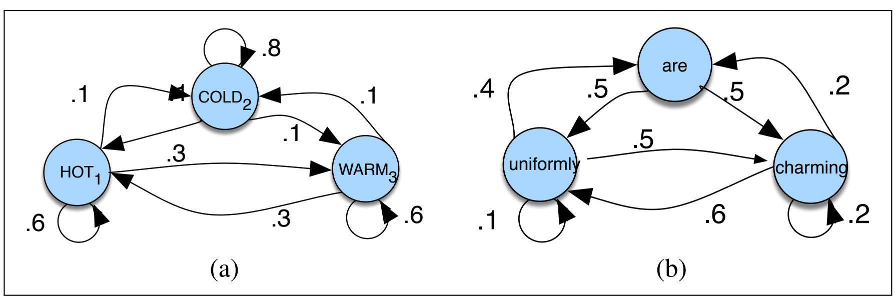


Figure 8.3 A Markov chain for weather (a) and one for words (b), showing states and transitions. A start distribution π is required; setting $\pi = [.1, .7., 2]$ for (a) would mean a probability 0.7 of starting in state 2 (cold), probability 0.1 of starting in state 1 (hot), etc.

Markov assumption

More formally, consider a sequence of state variables $q_1, q_2, ..., q_i$. A Markov model embodies the **Markov assumption** on the probabilities of this sequence: that when predicting the future, the past doesn't matter, only the present.

$$\textbf{Markov Assumption:} \quad P(q_i = a | q_1 ... q_{i-1}) = P(q_i = a | q_{i-1}) \tag{8.4}$$

Figure 8.3a shows a Markov chain for assigning a probability to a sequence of weather events, for which the vocabulary consists of HOT, COLD, and WARM. The

states are represented as nodes in the graph, and the transitions, with their probabilities, as edges. The transitions are probabilities: the values of arcs leaving a given state must sum to 1. Figure 8.3b shows a Markov chain for assigning a probability to a sequence of words $w_1...w_n$. This Markov chain should be familiar; in fact, it represents a bigram language model, with each edge expressing the probability $p(w_i|w_j)$! Given the two models in Fig. 8.3, we can assign a probability to any sequence from our vocabulary.

Formally, a Markov chain is specified by the following components:

$Q = q_1 q_2 \dots q_N$	a set of N **states**
$A = a_{01} a_{02} \dots a_{n1} \dots a_{nn}$	a **transition probability matrix** A, each a_{ij} representing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{n} a_{ij} = 1 \quad \forall i$
$\pi = \pi_1, \pi_2, ..., \pi_N$	an **initial probability distribution** over states. π_i is the probability that the Markov chain will start in state i. Some states j may have $\pi_j = 0$, meaning that they cannot be initial states. Also, $\sum_{i=1}^{n} \pi_i = 1$

Before you go on, use the sample probabilities in Fig. 8.3a (with $\pi = [.1, .7., 2]$) to compute the probability of each of the following sequences:

(8.5) hot hot hot hot
(8.6) cold hot cold hot

What does the difference in these probabilities tell you about a real-world weather fact encoded in Fig. 8.3a?

8.4.2 The Hidden Markov Model

A Markov chain is useful when we need to compute a probability for a sequence of observable events. In many cases, however, the events we are interested in are **hidden**: we don't observe them directly. For example we don't normally observe part-of-speech tags in a text. Rather, we see words, and must infer the tags from the word sequence. We call the tags **hidden** because they are not observed.

hidden

Hidden Markov model

A **hidden Markov model** (**HMM**) allows us to talk about both *observed* events (like words that we see in the input) and *hidden* events (like part-of-speech tags) that we think of as causal factors in our probabilistic model. An HMM is specified by the following components:

$Q = q_1 q_2 \dots q_N$	a set of N **states**
$A = a_{11} \dots a_{ij} \dots a_{NN}$	a **transition probability matrix** A, each a_{ij} representing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{N} a_{ij} = 1 \quad \forall i$
$O = o_1 o_2 \dots o_T$	a sequence of T **observations**, each one drawn from a vocabulary $V = v_1, v_2, ..., v_V$
$B = b_i(o_t)$	a sequence of **observation likelihoods**, also called **emission probabilities**, each expressing the probability of an observation o_t being generated from a state i
$\pi = \pi_1, \pi_2, ..., \pi_N$	an **initial probability distribution** over states. π_i is the probability that the Markov chain will start in state i. Some states j may have $\pi_j = 0$, meaning that they cannot be initial states. Also, $\sum_{i=1}^{n} \pi_i = 1$

A first-order hidden Markov model instantiates two simplifying assumptions. First, as with a first-order Markov chain, the probability of a particular state depends only on the previous state:

$$\textbf{Markov Assumption:} \quad P(q_i|q_1...q_{i-1}) = P(q_i|q_{i-1}) \tag{8.7}$$

Second, the probability of an output observation o_i depends only on the state that produced the observation q_i and not on any other states or any other observations:

$$\textbf{Output Independence:} \quad P(o_i|q_1 \ldots q_i, \ldots, q_T, o_1, \ldots, o_i, \ldots, o_T) = P(o_i|q_i) \tag{8.8}$$

8.4.3 The components of an HMM tagger

Let's start by looking at the pieces of an HMM tagger, and then we'll see how to use it to tag. An HMM has two components, the *A* and *B* probabilities.

The *A* matrix contains the tag transition probabilities $P(t_i|t_{i-1})$ which represent the probability of a tag occurring given the previous tag. For example, modal verbs like *will* are very likely to be followed by a verb in the base form, a VB, like *race*, so we expect this probability to be high. We compute the maximum likelihood estimate of this transition probability by counting, out of the times we see the first tag in a labeled corpus, how often the first tag is followed by the second:

$$P(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i)}{C(t_{i-1})} \tag{8.9}$$

In the WSJ corpus, for example, MD occurs 13124 times of which it is followed by VB 10471, for an MLE estimate of

$$P(VB|MD) = \frac{C(MD, VB)}{C(MD)} = \frac{10471}{13124} = .80 \tag{8.10}$$

Let's walk through an example, seeing how these probabilities are estimated and used in a sample tagging task, before we return to the algorithm for decoding.

In HMM tagging, the probabilities are estimated by counting on a tagged training corpus. For this example we'll use the tagged WSJ corpus.

The *B* emission probabilities, $P(w_i|t_i)$, represent the probability, given a tag (say MD), that it will be associated with a given word (say *will*). The MLE of the emission probability is

$$P(w_i|t_i) = \frac{C(t_i, w_i)}{C(t_i)} \tag{8.11}$$

Of the 13124 occurrences of MD in the WSJ corpus, it is associated with *will* 4046 times:

$$P(will|MD) = \frac{C(MD, will)}{C(MD)} = \frac{4046}{13124} = .31 \tag{8.12}$$

We saw this kind of Bayesian modeling in Chapter 4; recall that this likelihood term is not asking "which is the most likely tag for the word *will*?" That would be the posterior $P(\text{MD}|\text{will})$. Instead, $P(\text{will}|\text{MD})$ answers the slightly counterintuitive question "If we were going to generate a MD, how likely is it that this modal would be *will*?"

The *A* transition probabilities, and *B* observation likelihoods of the HMM are illustrated in Fig. 8.4 for three states in an HMM part-of-speech tagger; the full tagger would have one state for each tag.

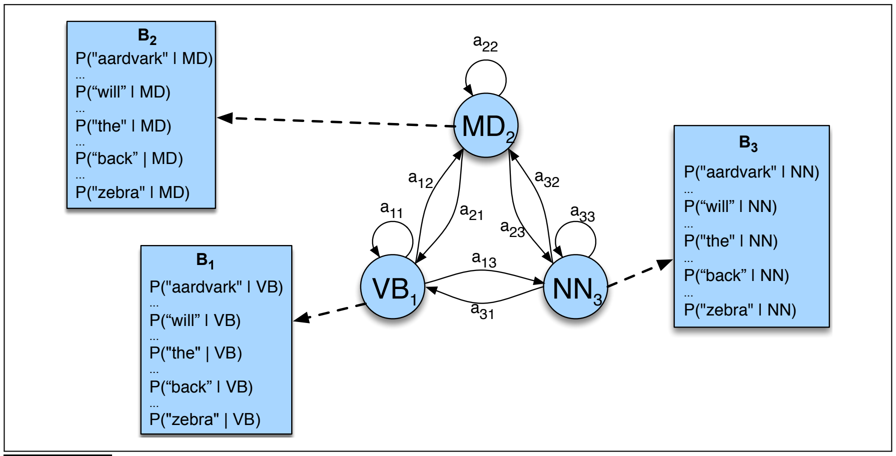


Figure 8.4 An illustration of the two parts of an HMM representation: the A transition probabilities used to compute the prior probability, and the B observation likelihoods that are associated with each state, one likelihood for each possible observation word.

8.4.4 HMM tagging as decoding

For any model, such as an HMM, that contains hidden variables, the task of determining the hidden variables sequence corresponding to the sequence of observations is called **decoding**. More formally,

decoding

> **Decoding**: Given as input an HMM $\lambda = (A, B)$ and a sequence of observations $O = o_1, o_2, ..., o_T$, find the most probable sequence of states $Q = q_1 q_2 q_3 \ldots q_T$.

For part of speech tagging, the goal of HMM decoding is to choose the tag sequence t_1^n that is most probable given the observation sequence of n words words w_1^n:

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} P(t_1^n | w_1^n) \tag{8.13}$$

The way we'll do this in the HMM is to use Bayes' rule to instead compute:

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} \frac{P(w_1^n | t_1^n) P(t_1^n)}{P(w_1^n)} \tag{8.14}$$

Furthermore, we simplify Eq. 8.14 by dropping the denominator $P(w_1^n)$:

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} P(w_1^n | t_1^n) P(t_1^n) \tag{8.15}$$

HMM taggers make two further simplifying assumptions. The first is that the probability of a word appearing depends only on its own tag and is independent of neighboring words and tags:

$$P(w_1^n | t_1^n) \approx \prod_{i=1}^{n} P(w_i | t_i) \tag{8.16}$$

The second assumption, the **bigram** assumption, is that the probability of a tag is dependent only on the previous tag, rather than the entire tag sequence;

$$P(t_1^n) \approx \prod_{i=1}^{n} P(t_i | t_{i-1}) \tag{8.17}$$

Plugging the simplifying assumptions from Eq. 8.16 and Eq. 8.17 into Eq. 8.15 results in the following equation for the most probable tag sequence from a bigram tagger:

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} P(t_1^n|w_1^n) \approx \underset{t_1^n}{\operatorname{argmax}} \prod_{i=1}^{n} \overbrace{P(w_i|t_i)}^{\text{emission}} \overbrace{P(t_i|t_{i-1})}^{\text{transition}} \tag{8.18}$$

The two parts of Eq. 8.18 correspond neatly to the B **emission probability** and A **transition probability** that we just defined above!

8.4.5 The Viterbi Algorithm

Viterbi algorithm

The decoding algorithm for HMMs is the **Viterbi algorithm** shown in Fig. 8.5. As an instance of **dynamic programming**, Viterbi resembles the dynamic programming **minimum edit distance** algorithm of Chapter 2.

function VITERBI(*observations* of len T,*state-graph* of len N) **returns** *best-path*, *path-prob*

create a path probability matrix *viterbi[N,T]*
for each state s **from** 1 **to** N **do** ; initialization step
  $viterbi[s,1] \leftarrow \pi_s * b_s(o_1)$
  $backpointer[s,1] \leftarrow 0$
for each time step t **from** 2 **to** T **do** ; recursion step
  for each state s **from** 1 **to** N **do**
    $viterbi[s,t] \leftarrow \max_{s'=1}^{N} \; viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
    $backpointer[s,t] \leftarrow \operatorname{argmax}_{s'=1}^{N} \; viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
$bestpathprob \leftarrow \max_{s=1}^{N} \; viterbi[s,T]$; termination step
$bestpathpointer \leftarrow \operatorname{argmax}_{s=1}^{N} \; viterbi[s,T]$; termination step
bestpath ← the path starting at state *bestpathpointer*, that follows backpointer[] to states back in time
return *bestpath*, *bestpathprob*

Figure 8.5 Viterbi algorithm for finding the optimal sequence of tags. Given an observation sequence and an HMM $\lambda = (A,B)$, the algorithm returns the state path through the HMM that assigns maximum likelihood to the observation sequence.

The Viterbi algorithm first sets up a probability matrix or **lattice**, with one column for each observation o_t and one row for each state in the state graph. Each column thus has a cell for each state q_i in the single combined automaton. Figure 8.6 shows an intuition of this lattice for the sentence *Janet will back the bill*.

Each cell of the trellis, $v_t(j)$, represents the probability that the HMM is in state j after seeing the first t observations and passing through the most probable state sequence $q_1,...,q_{t-1}$, given the HMM λ. The value of each cell $v_t(j)$ is computed by recursively taking the most probable path that could lead us to this cell. Formally, each cell expresses the probability

$$v_t(j) = \max_{q_1,...,q_{t-1}} P(q_0,q_1...q_{t-1},o_1,o_2\ldots o_t,q_t = j|\lambda) \tag{8.19}$$

We represent the most probable path by taking the maximum over all possible previous state sequences $\max_{q_0,q_1,...,q_{t-1}}$. Like other dynamic programming algorithms,

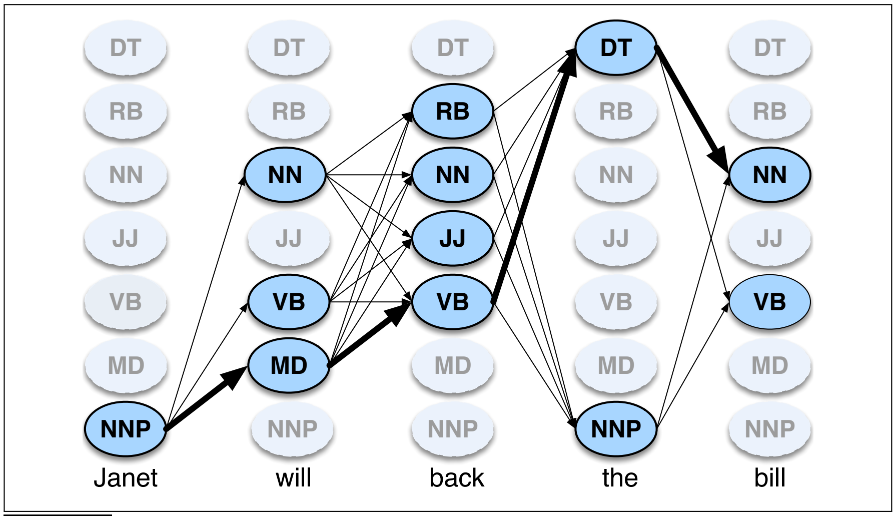


Figure 8.6 A sketch of the lattice for *Janet will back the bill*, showing the possible tags (q_i) for each word and highlighting the path corresponding to the correct tag sequence through the hidden states. States (parts-of-speech) which have a zero probability of generating a particular word according to the B matrix (such as the probability that a determiner DT will be realized as *Janet*) are greyed out.

Viterbi fills each cell recursively. Given that we had already computed the probability of being in every state at time $t-1$, we compute the Viterbi probability by taking the most probable of the extensions of the paths that lead to the current cell. For a given state q_j at time t, the value $v_t(j)$ is computed as

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i)\, a_{ij}\, b_j(o_t) \tag{8.20}$$

The three factors that are multiplied in Eq. 8.20 for extending the previous paths to compute the Viterbi probability at time t are

$v_{t-1}(i)$	the **previous Viterbi path probability** from the previous time step
a_{ij}	the **transition probability** from previous state q_i to current state q_j
$b_j(o_t)$	the **state observation likelihood** of the observation symbol o_t given the current state j

8.4.6 Working through an example

Let's tag the sentence *Janet will back the bill*; the goal is the correct series of tags (see also Fig. 8.6):

(8.21) Janet/NNP will/MD back/VB the/DT bill/NN

Let the HMM be defined by the two tables in Fig. 8.7 and Fig. 8.8. Figure 8.7 lists the a_{ij} probabilities for transitioning between the hidden states (part-of-speech tags). Figure 8.8 expresses the $b_i(o_t)$ probabilities, the *observation* likelihoods of words given tags. This table is (slightly simplified) from counts in the WSJ corpus. So the word *Janet* only appears as an NNP, *back* has 4 possible parts of speech, and the word *the* can appear as a determiner or as an NNP (in titles like "Somewhere Over the Rainbow" all words are tagged as NNP).

Figure 8.9 shows a fleshed-out version of the sketch we saw in Fig. 8.6, the Viterbi trellis for computing the best hidden state sequence for the observation sequence *Janet will back the bill*.

	NNP	**MD**	**VB**	**JJ**	**NN**	**RB**	**DT**
$<s>$	0.2767	0.0006	0.0031	0.0453	0.0449	0.0510	0.2026
NNP	0.3777	0.0110	0.0009	0.0084	0.0584	0.0090	0.0025
MD	0.0008	0.0002	0.7968	0.0005	0.0008	0.1698	0.0041
VB	0.0322	0.0005	0.0050	0.0837	0.0615	0.0514	0.2231
JJ	0.0366	0.0004	0.0001	0.0733	0.4509	0.0036	0.0036
NN	0.0096	0.0176	0.0014	0.0086	0.1216	0.0177	0.0068
RB	0.0068	0.0102	0.1011	0.1012	0.0120	0.0728	0.0479
DT	0.1147	0.0021	0.0002	0.2157	0.4744	0.0102	0.0017

Figure 8.7 The A transition probabilities $P(t_i|t_{i-1})$ computed from the WSJ corpus without smoothing. Rows are labeled with the conditioning event; thus $P(VB|MD)$ is 0.7968.

	Janet	**will**	**back**	**the**	**bill**
NNP	0.000032	0	0	0.000048	0
MD	0	0.308431	0	0	0
VB	0	0.000028	0.000672	0	0.000028
JJ	0	0	0.000340	0	0
NN	0	0.000200	0.000223	0	0.002337
RB	0	0	0.010446	0	0
DT	0	0	0	0.506099	0

Figure 8.8 Observation likelihoods B computed from the WSJ corpus without smoothing, simplified slightly.

There are $N = 5$ state columns. We begin in column 1 (for the word *Janet*) by setting the Viterbi value in each cell to the product of the π transition probability (the start probability for that state i)) and the observation likelihood of the word *Janet* given the tag for that cell. Most of the cells in the column are zero since the word *Janet* cannot be any of those tags. The reader should find this in Fig. 8.9.

Next, each cell in the *will* column gets updated. For each state, we compute the value $viterbi[s,t]$ by taking the maximum over the extensions of all the paths from the previous column that lead to the current cell according to Eq. 8.20. We have shown the values for the MD, VB, and NN cells. Each cell gets the max of the 7 values from the previous column, multiplied by the appropriate transition probability; as it happens in this case, most of them are zero from the previous column. The remaining value is multiplied by the relevant transition probability, and the (trivial) max is taken. In this case the final value, .0000002772, comes from the NNP state at the previous column. The reader should fill in the rest of the trellis in Fig. 8.9 and backtrace to reconstruct the correct state sequence NNP MD VB DT NN.

8.4.7 Extending the HMM Algorithm to Trigrams

Practical HMM taggers have a number of extensions of this simple model. One important missing feature is a wider tag context. In the tagger described above the probability of a tag depends only on the previous tag:

$$P(t_1^n) \approx \prod_{i=1}^{n} P(t_i|t_{i-1}) \tag{8.22}$$

In practice we use more of the history, letting the probability of a tag depend on

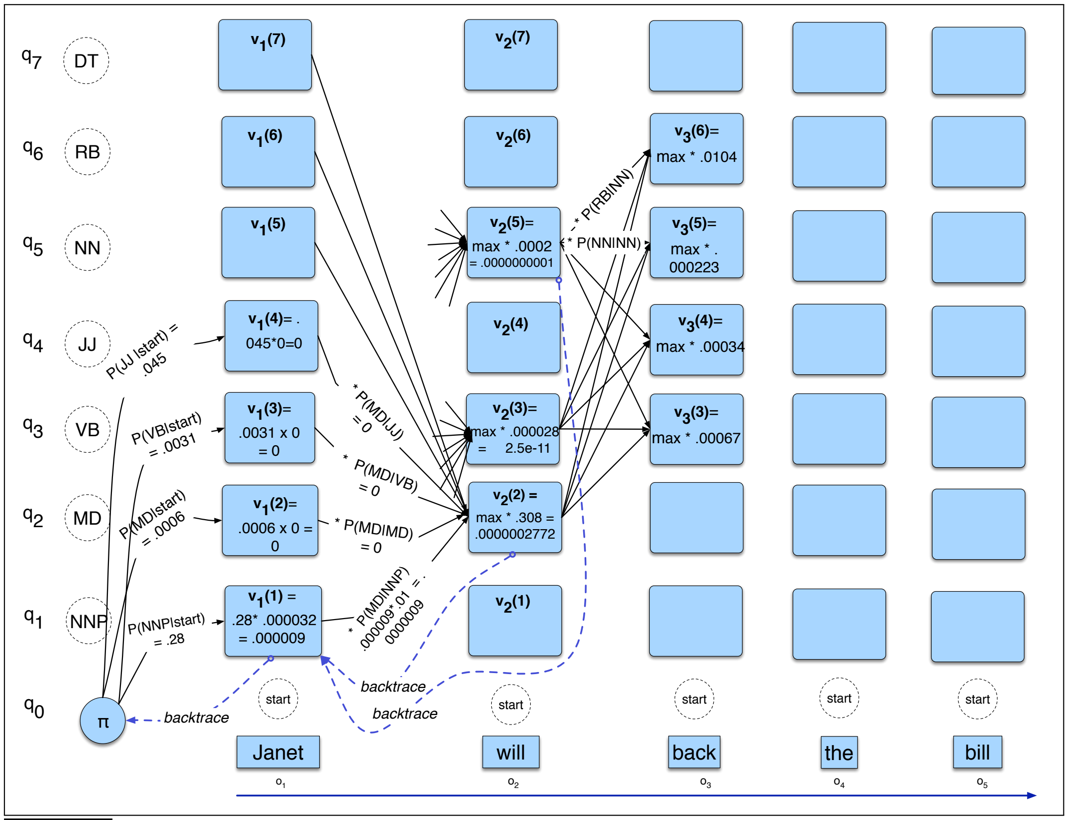


Figure 8.9 The first few entries in the individual state columns for the Viterbi algorithm. Each cell keeps the probability of the best path so far and a pointer to the previous cell along that path. We have only filled out columns 1 and 2; to avoid clutter most cells with value 0 are left empty. The rest is left as an exercise for the reader. After the cells are filled in, backtracing from the *end* state, we should be able to reconstruct the correct state sequence NNP MD VB DT NN.

the two previous tags:

$$P(t_1^n) \approx \prod_{i=1}^{n} P(t_i|t_{i-1}, t_{i-2}) \tag{8.23}$$

Extending the algorithm from bigram to trigram taggers gives a small (perhaps a half point) increase in performance, but conditioning on two previous tags instead of one requires a significant change to the Viterbi algorithm. For each cell, instead of taking a max over transitions from each cell in the previous column, we have to take a max over paths through the cells in the previous two columns, thus considering N^2 rather than N hidden states at every observation.

In addition to increasing the context window, HMM taggers have a number of other advanced features. One is to let the tagger know the location of the end of the sentence by adding dependence on an end-of-sequence marker for t_{n+1}. This gives the following equation for part-of-speech tagging:

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}}\, P(t_1^n|w_1^n) \approx \underset{t_1^n}{\operatorname{argmax}} \left[\prod_{i=1}^{n} P(w_i|t_i) P(t_i|t_{i-1}, t_{i-2}) \right] P(t_{n+1}|t_n) \tag{8.24}$$

In tagging any sentence with Eq. 8.24, three of the tags used in the context will fall off the edge of the sentence, and hence will not match regular words. These tags,

t_{-1}, t_0, and t_{n+1}, can all be set to be a single special 'sentence boundary' tag that is added to the tagset, which assumes sentences boundaries have already been marked.

One problem with trigram taggers as instantiated in Eq. 8.24 is data sparsity. Any particular sequence of tags t_{i-2}, t_{i-1}, t_i that occurs in the test set may simply never have occurred in the training set. That means we cannot compute the tag trigram probability just by the maximum likelihood estimate from counts, following Eq. 8.25:

$$P(t_i|t_{i-1}, t_{i-2}) = \frac{C(t_{i-2}, t_{i-1}, t_i)}{C(t_{i-2}, t_{i-1})} \tag{8.25}$$

Just as we saw with language modeling, many of these counts will be zero in any training set, and we will incorrectly predict that a given tag sequence will never occur! What we need is a way to estimate $P(t_i|t_{i-1}, t_{i-2})$ even if the sequence t_{i-2}, t_{i-1}, t_i never occurs in the training data.

The standard approach to solving this problem is the same interpolation idea we saw in language modeling: estimate the probability by combining more robust, but weaker estimators. For example, if we've never seen the tag sequence PRP VB TO, and so can't compute P(TO|PRP,VB) from this frequency, we still could rely on the bigram probability P(TO|VB), or even the unigram probability P(TO). The maximum likelihood estimation of each of these probabilities can be computed from a corpus with the following counts:

$$\text{Trigrams} \quad \hat{P}(t_i|t_{i-1}, t_{i-2}) = \frac{C(t_{i-2}, t_{i-1}, t_i)}{C(t_{i-2}, t_{i-1})} \tag{8.26}$$

$$\text{Bigrams} \quad \hat{P}(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i)}{C(t_{i-1})} \tag{8.27}$$

$$\text{Unigrams} \quad \hat{P}(t_i) = \frac{C(t_i)}{N} \tag{8.28}$$

The standard way to combine these three estimators to estimate the trigram probability $P(t_i|t_{i-1}, t_{i-2})$ is via linear interpolation. We estimate the probability $P(t_i|t_{i-1}t_{i-2})$ by a weighted sum of the unigram, bigram, and trigram probabilities:

$$P(t_i|t_{i-1}t_{i-2}) = \lambda_3 \hat{P}(t_i|t_{i-1}t_{i-2}) + \lambda_2 \hat{P}(t_i|t_{i-1}) + \lambda_1 \hat{P}(t_i) \tag{8.29}$$

deleted interpolation

We require $\lambda_1 + \lambda_2 + \lambda_3 = 1$, ensuring that the resulting P is a probability distribution. The λs are set by **deleted interpolation** (Jelinek and Mercer, 1980): we successively delete each trigram from the training corpus and choose the λs so as to maximize the likelihood of the rest of the corpus. The deletion helps to set the λs in such a way as to generalize to unseen data and not overfit. Figure 8.10 gives a deleted interpolation algorithm for tag trigrams.

8.4.8 Beam Search

When the number of states grows very large, the vanilla Viterbi algorithm be slow. The complexity of the algorithm is $O(N^2T)$; N (the number of states) can be large for trigram taggers, which have to consider every previous pair of the 45 tags, resulting in $45^3 = 91,125$ computations per column. N can be even larger for other applications of Viterbi, for example to decoding in neural networks, as we will see in future chapters.

beam search

One common solution to the complexity problem is the use of **beam search** decoding. In beam search, instead of keeping the entire column of states at each

function DELETED-INTERPOLATION(*corpus*) **returns** $\lambda_1, \lambda_2, \lambda_3$

$\lambda_1,\ \lambda_2,\ \lambda_3 \leftarrow 0$
foreach trigram t_1, t_2, t_3 with $C(t_1, t_2, t_3) > 0$
  depending on the maximum of the following three values
    case $\frac{C(t_1,t_2,t_3)-1}{C(t_1,t_2)-1}$: increment λ_3 by $C(t_1, t_2, t_3)$
    case $\frac{C(t_2,t_3)-1}{C(t_2)-1}$: increment λ_2 by $C(t_1, t_2, t_3)$
    case $\frac{C(t_3)-1}{N-1}$: increment λ_1 by $C(t_1, t_2, t_3)$
  end
end
normalize $\lambda_1, \lambda_2, \lambda_3$
return $\lambda_1, \lambda_2, \lambda_3$

Figure 8.10 The deleted interpolation algorithm for setting the weights for combining unigram, bigram, and trigram tag probabilities. If the denominator is 0 for any case, we define the result of that case to be 0. N is the number of tokens in the corpus. After Brants (2000).

time point t, we just keep the best few hypothesis at that point. At time t this requires computing the Viterbi score for each of the N cells, sorting the scores, and keeping only the best-scoring states. The rest are pruned out and not continued forward to time $t+1$.

beam width

One way to implement beam search is to keep a fixed number of states instead of all N current states. Here the **beam width** β is a fixed number of states. Alternatively β can be modeled as a fixed percentage of the N states, or as a probability threshold. Figure 8.11 shows the search lattice using a beam width of 2 states.

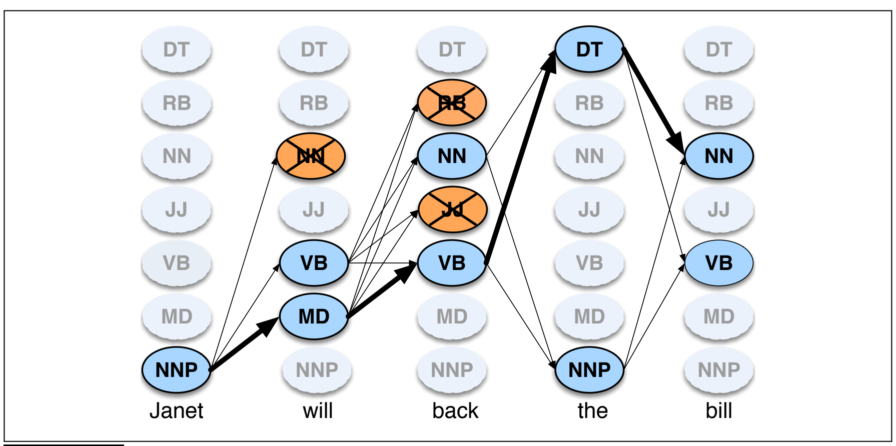


Figure 8.11 A beam search version of Fig. 8.6, showing a beam width of 2. At each time t, all (non-zero) states are computed, but then they are sorted and only the best 2 states are propagated forward and the rest are pruned, shown in orange.

8.4.9 Unknown Words

words people
never use —
could be
only I
know them Ishikawa Takuboku 1885–1912

unknown words

To achieve high accuracy with part-of-speech taggers, it is also important to have a good model for dealing with **unknown words**. Proper names and acronyms are created very often, and even new common nouns and verbs enter the language at a surprising rate. One useful feature for distinguishing parts of speech is word shape: words starting with capital letters are likely to be proper nouns (NNP).

But the strongest source of information for guessing the part-of-speech of unknown words is morphology. Words that end in *-s* are likely to be plural nouns (NNS), words ending with *-ed* tend to be past participles (VBN), words ending with *-able* adjectives (JJ), and so on. We store for each final letter sequence (for simplicity referred to as word *suffixes*) of up to 10 letters the statistics of the tag it was associated with in training. We are thus computing for each suffix of length i the probability of the tag t_i given the suffix letters (Samuelsson 1993, Brants 2000):

$$P(t_i|l_{n-i+1}\ldots l_n) \tag{8.30}$$

Back-off is used to smooth these probabilities with successively shorter suffixes. Because unknown words are unlikely to be closed-class words like prepositions, suffix probabilities can be computed only for words whose training set frequency is ≤ 10, or only for open-class words. Separate suffix tries are kept for capitalized and uncapitalized words.

Finally, because Eq. 8.30 gives a posterior estimate $p(t_i|w_i)$, we can compute the likelihood $p(w_i|t_i)$ that HMMs require by using Bayesian inversion (i.e., using Bayes rule and computation of the two priors $P(t_i)$ and $P(t_i|l_{n-i+1}\ldots l_n)$).

In addition to using capitalization information for unknown words, Brants (2000) also uses capitalization for known words by adding a capitalization feature to each tag. Thus, instead of computing $P(t_i|t_{i-1},t_{i-2})$ as in Eq. 8.26, the algorithm computes the probability $P(t_i,c_i|t_{i-1},c_{i-1},t_{i-2},c_{i-2})$. This is equivalent to having a capitalized and uncapitalized version of each tag, doubling the size of the tagset.

Combining all these features, a trigram HMM like that of Brants (2000) has a tagging accuracy of 96.7% on the Penn Treebank, perhaps just slightly below the performance of the best MEMM and neural taggers.

8.5 Maximum Entropy Markov Models

While an HMM can achieve very high accuracy, we saw that it requires a number of architectural innovations to deal with unknown words, backoff, suffixes, and so on. It would be so much easier if we could add arbitrary features directly into the model in a clean way, but that's hard for generative models like HMMs. Luckily, we've already seen a model for doing this: the logistic regression model of Chapter 5! But logistic regression isn't a sequence model; it assigns a class to a single observation. However, we could turn logistic regression into a discriminative sequence model simply by running it on successive words, using the class assigned to the prior word

as a feature in the classification of the next word. When we apply logistic regression in this way, it's called the **maximum entropy Markov model** or **MEMM**[5]

MEMM

Let the sequence of words be $W = w_1^n$ and the sequence of tags $T = t_1^n$. In an HMM to compute the best tag sequence that maximizes $P(T|W)$ we rely on Bayes' rule and the likelihood $P(W|T)$:

$$\begin{aligned}\hat{T} &= \underset{T}{\operatorname{argmax}}\, P(T|W) \\ &= \underset{T}{\operatorname{argmax}}\, P(W|T)P(T) \\ &= \underset{T}{\operatorname{argmax}} \prod_i P(word_i|tag_i) \prod_i P(tag_i|tag_{i-1}) \qquad (8.31)\end{aligned}$$

In an MEMM, by contrast, we compute the posterior $P(T|W)$ directly, training it to discriminate among the possible tag sequences:

$$\begin{aligned}\hat{T} &= \underset{T}{\operatorname{argmax}}\, P(T|W) \\ &= \underset{T}{\operatorname{argmax}} \prod_i P(t_i|w_i, t_{i-1}) \qquad (8.32)\end{aligned}$$

Consider tagging just one word. A multinomial logistic regression classifier could compute the single probability $P(t_i|w_i, t_{i-1})$ in a different way that an HMM. Fig. 8.12 shows the intuition of the difference via the direction of the arrows; HMMs compute likelihood (observation word conditioned on tags) but MEMMs compute posterior (tags conditioned on observation words).

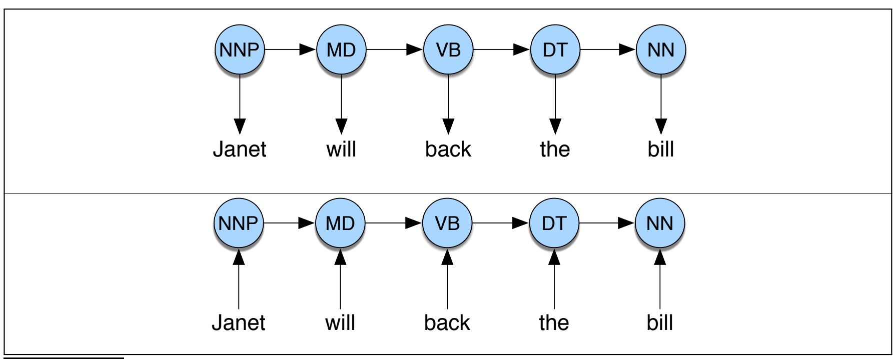


Figure 8.12 A schematic view of the HMM (top) and MEMM (bottom) representation of the probability computation for the correct sequence of tags for the *back* sentence. The HMM computes the likelihood of the observation given the hidden state, while the MEMM computes the posterior of each state, conditioned on the previous state and current observation.

8.5.1 Features in a MEMM

Of course we don't build MEMMs that condition just on w_i and t_{i-1}. The reason to use a discriminative sequence model is that it's easier to incorporate a lots of features.[6] Figure 8.13 shows a graphical intuition of some of these additional features.

5 'Maximum entropy model' is an outdated name for logistic regression; see the history section.

6 Because in HMMs all computation is based on the two probabilities $P(\text{tag}|\text{tag})$ and $P(\text{word}|\text{tag})$, if we want to include some source of knowledge into the tagging process, we must find a way to encode the knowledge into one of these two probabilities. Each time we add a feature we have to do a lot of complicated conditioning which gets harder and harder as we have more and more such features.

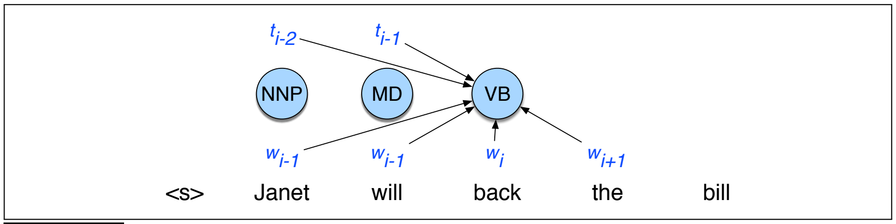


Figure 8.13 An MEMM for part-of-speech tagging showing the ability to condition on more features.

A basic MEMM part-of-speech tagger conditions on the observation word itself, neighboring words, and previous tags, and various combinations, using feature **templates** like the following:

templates

$$\begin{aligned} \langle t_i, w_{i-2}\rangle, \langle t_i, w_{i-1}\rangle, \langle t_i, w_i\rangle, \langle t_i, w_{i+1}\rangle, \langle t_i, w_{i+2}\rangle \\ \langle t_i, t_{i-1}\rangle, \langle t_i, t_{i-2}, t_{i-1}\rangle, \\ \langle t_i, t_{i-1}, w_i\rangle, \langle t_i, w_{i-1}, w_i\rangle \langle t_i, w_i, w_{i+1}\rangle, \end{aligned} \tag{8.33}$$

Recall from Chapter 5 that feature templates are used to automatically populate the set of features from every instance in the training and test set. Thus our example *Janet/NNP will/MD back/VB the/DT bill/NN*, when w_i is the word *back*, would generate the following features:

t_i = VB and w_{i-2} = Janet
t_i = VB and w_{i-1} = will
t_i = VB and w_i = back
t_i = VB and w_{i+1} = the
t_i = VB and w_{i+2} = bill
t_i = VB and t_{i-1} = MD
t_i = VB and t_{i-1} = MD and t_{i-2} = NNP
t_i = VB and w_i = back and w_{i+1} = the

Also necessary are features to deal with unknown words, expressing properties of the word's spelling or shape:

w_i contains a particular prefix (from all prefixes of length ≤ 4)
w_i contains a particular suffix (from all suffixes of length ≤ 4)
w_i contains a number
w_i contains an upper-case letter
w_i contains a hyphen
w_i is all upper case
w_i's word shape
w_i's short word shape
w_i is upper case and has a digit and a dash (like *CFC-12*)
w_i is upper case and followed within 3 words by Co., Inc., etc.

word shape

Word shape features are used to represent the abstract letter pattern of the word by mapping lower-case letters to 'x', upper-case to 'X', numbers to 'd', and retaining punctuation. Thus for example I.M.F would map to X.X.X. and DC10-30 would map to XXdd-dd. A second class of shorter word shape features is also used. In these features consecutive character types are removed, so DC10-30 would be mapped to Xd-d but I.M.F would still map to X.X.X. For example the word *well-dressed* would generate the following non-zero valued feature values:

prefix(w_i) = `w`
prefix(w_i) = `we`
prefix(w_i) = `wel`
prefix(w_i) = `well`
suffix(w_i) = `ssed`
suffix(w_i) = `sed`
suffix(w_i) = `ed`
suffix(w_i) = `d`
has-hyphen(w_i)
word-shape(w_i) = `xxxx-xxxxxxx`
short-word-shape(w_i) = `x-x`

Features for known words, like the templates in Eq. 8.33, are computed for every word seen in the training set. The unknown word features can also be computed for all words in training, or only on training words whose frequency is below some threshold. The result of the known-word templates and word-signature features is a very large set of features. Generally a feature cutoff is used in which features are thrown out if they have count < 5 in the training set.

8.5.2 Decoding and Training MEMMs

The most likely sequence of tags is then computed by combining these features of the input word w_i, its neighbors within l words w_{i-l}^{i+l}, and the previous k tags t_{i-k}^{i-1} as follows (using θ to refer to feature weights instead of w to avoid the confusion with w meaning words):

$$\begin{aligned}\hat{T} &= \underset{T}{\operatorname{argmax}}\, P(T|W) \\ &= \underset{T}{\operatorname{argmax}} \prod_i P(t_i|w_{i-l}^{i+l}, t_{i-k}^{i-1}) \\ &= \underset{T}{\operatorname{argmax}} \prod_i \frac{\exp\left(\sum_j \theta_j f_j(t_i, w_{i-l}^{i+l}, t_{i-k}^{i-1})\right)}{\sum_{t' \in \text{tagset}} \exp\left(\sum_j \theta_j f_j(t', w_{i-l}^{i+l}, t_{i-k}^{i-1})\right)}\end{aligned} \tag{8.34}$$

How should we decode to find this optimal tag sequence $\hat{T}$? The simplest way to turn logistic regression into a sequence model is to build a local classifier that classifies each word left to right, making a hard classification of the first word in the sentence, then a hard decision on the the second word, and so on. This is called a **greedy** decoding algorithm, because we greedily choose the best tag for each word, as shown in Fig. 8.14.

greedy

function GREEDY SEQUENCE DECODING(words W, model P) **returns** tag sequence T

for i = 1 **to** *length(W)*
  $\hat{t}_i \;=\; \underset{t' \in\, T}{\operatorname{argmax}}\; P(t' \mid w_{i-l}^{i+l}, t_{i-k}^{i-1})$

Figure 8.14 In greedy decoding we simply run the classifier on each token, left to right, each time making a hard decision of which is the best tag.

The problem with the greedy algorithm is that by making a hard decision on each word before moving on to the next word, the classifier can't use evidence from future decisions. Although the greedy algorithm is very fast, and occasionally has sufficient accuracy to be useful, in general the hard decision causes too much a drop in performance, and we don't use it.

Viterbi

Instead we decode an MEMM with the **Viterbi** algorithm just as with the HMM, finding the sequence of part-of-speech tags that is optimal for the whole sentence.

For example, assume that our MEMM is only conditioning on the previous tag t_{i-1} and observed word w_i. Concretely, this involves filling an $N \times T$ array with the appropriate values for $P(t_i|t_{i-1}, w_i)$, maintaining backpointers as we proceed. As with HMM Viterbi, when the table is filled, we simply follow pointers back from the maximum value in the final column to retrieve the desired set of labels. The requisite changes from the HMM-style application of Viterbi have to do only with how we fill each cell. Recall from Eq. 8.20 that the recursive step of the Viterbi equation computes the Viterbi value of time t for state j as

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i) a_{ij} b_j(o_t); \quad 1 \leq j \leq N, 1 < t \leq T \tag{8.35}$$

which is the HMM implementation of

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i) P(s_j|s_i) P(o_t|s_j) \quad 1 \leq j \leq N, 1 < t \leq T \tag{8.36}$$

The MEMM requires only a slight change to this latter formula, replacing the a and b prior and likelihood probabilities with the direct posterior:

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i) P(s_j|s_i, o_t) \quad 1 \leq j \leq N, 1 < t \leq T \tag{8.37}$$

Learning in MEMMs relies on the same supervised learning algorithms we presented for logistic regression. Given a sequence of observations, feature functions, and corresponding hidden states, we use gradient descent to train the weights to maximize the log-likelihood of the training corpus.

8.6 Bidirectionality

The one problem with the MEMM and HMM models as presented is that they are exclusively run left-to-right. While the Viterbi algorithm still allows present decisions to be influenced indirectly by future decisions, it would help even more if a decision about word w_i could directly use information about future tags t_{i+1} and t_{i+2}.

label bias
observation bias

Adding bidirectionality has another useful advantage. MEMMs have a theoretical weakness, referred to alternatively as the **label bias** or **observation bias** problem (Lafferty et al. 2001, Toutanova et al. 2003). These are names for situations when one source of information is ignored because it is **explained away** by another source. Consider an example from Toutanova et al. (2003), the sequence *will/NN to/TO fight/VB*. The tag TO is often preceded by NN but rarely by modals (MD), and so that tendency should help predict the correct NN tag for *will*. But the previous transition $P(t_{will}|\langle s \rangle)$ prefers the modal, and because $P(TO|to, t_{will})$ is so close to 1 regardless of t_{will} the model cannot make use of the transition probability and incorrectly chooses MD. The strong information that *to* must have the tag TO has **explained away** the presence of TO and so the model doesn't learn the importance of

the previous NN tag for predicting TO. Bidirectionality helps the model by making the link between TO available when tagging the NN.

CRF

One way to implement bidirectionality is to switch to a more powerful model called a **conditional random field** or **CRF**. The CRF is an undirected graphical model, which means that it's not computing a probability for each tag at each time step. Instead, at each time step the CRF computes log-linear functions over a **clique**, a set of relevant features. Unlike for an MEMM, these might include output features of words in future time steps. The probability of the best sequence is similarly computed by the Viterbi algorithm. Because a CRF normalizes probabilities over all tag sequences, rather than over all the tags at an individual time *t*, training requires computing the sum over all possible labelings, which makes CRF training quite slow.

Stanford tagger

Simpler methods can also be used; the **Stanford tagger** uses a bidirectional version of the MEMM called a cyclic dependency network (Toutanova et al., 2003).

Alternatively, any sequence model can be turned into a bidirectional model by using multiple passes. For example, the first pass would use only part-of-speech features from already-disambiguated words on the left. In the second pass, tags for all words, including those on the right, can be used. Alternately, the tagger can be run twice, once left-to-right and once right-to-left. In greedy decoding, for each word the classifier chooses the highest-scoring of the tag assigned by the left-to-right and right-to-left classifier. In Viterbi decoding, the classifier chooses the higher scoring of the two sequences (left-to-right or right-to-left). These bidirectional models lead directly into the bi-LSTM models that we will introduce in Chapter 9 as a standard neural sequence model.

8.7 Part-of-Speech Tagging for Other Languages

Augmentations to tagging algorithms become necessary when dealing with languages with rich morphology like Czech, Hungarian and Turkish.

These productive word-formation processes result in a large vocabulary for these languages: a 250,000 word token corpus of Hungarian has more than twice as many word types as a similarly sized corpus of English (Oravecz and Dienes, 2002), while a 10 million word token corpus of Turkish contains four times as many word types as a similarly sized English corpus (Hakkani-Tür et al., 2002). Large vocabularies mean many unknown words, and these unknown words cause significant performance degradations in a wide variety of languages (including Czech, Slovene, Estonian, and Romanian) (Hajič, 2000).

Highly inflectional languages also have much more information than English coded in word morphology, like **case** (nominative, accusative, genitive) or **gender** (masculine, feminine). Because this information is important for tasks like parsing and coreference resolution, part-of-speech taggers for morphologically rich languages need to label words with case and gender information. Tagsets for morphologically rich languages are therefore sequences of morphological tags rather than a single primitive tag. Here's a Turkish example, in which the word *izin* has three possible morphological/part-of-speech tags and meanings (Hakkani-Tür et al., 2002):

1. Yerdeki **izin** temizlenmesi gerek. iz + `Noun+A3sg+Pnon+Gen`
   The trace on the floor should be cleaned.
2. Üzerinde parmak **izin** kalmiş iz + `Noun+A3sg+P2sg+Nom`
   Your finger **print** is left on (it).

3. Içeri girmek için **izin** alman gerekiyor. izin + `Noun+A3sg+Pnon+Nom`
   You need a **permission** to enter.

Using a morphological parse sequence like `Noun+A3sg+Pnon+Gen` as the part-of-speech tag greatly increases the number of parts-of-speech, and so tagsets can be 4 to 10 times larger than the 50–100 tags we have seen for English. With such large tagsets, each word needs to be morphologically analyzed to generate the list of possible morphological tag sequences (part-of-speech tags) for the word. The role of the tagger is then to disambiguate among these tags. This method also helps with unknown words since morphological parsers can accept unknown stems and still segment the affixes properly.

For non-word-space languages like Chinese, word segmentation (Chapter 2) is either applied before tagging or done jointly. Although Chinese words are on average very short (around 2.4 characters per unknown word compared with 7.7 for English) the problem of unknown words is still large. While English unknown words tend to be proper nouns in Chinese the majority of unknown words are common nouns and verbs because of extensive compounding. Tagging models for Chinese use similar unknown word features to English, including character prefix and suffix features, as well as novel features like the **radicals** of each character in a word. (Tseng et al., 2005b).

A stanford for multilingual tagging is the Universal POS tag set of the Universal Dependencies project, which contains 16 tags plus a wide variety of features that can be added to them to create a large tagset for any language (Nivre et al., 2016a).

8.8 Summary

This chapter introduced **parts-of-speech** and **part-of-speech tagging**:

- Languages generally have a small set of **closed class** words that are highly frequent, ambiguous, and act as **function words**, and **open-class** words like **nouns**, **verbs**, **adjectives**. Various part-of-speech **tagsets** exist, of between 40 and 200 tags.
- **Part-of-speech tagging** is the process of assigning a part-of-speech label to each of a sequence of words.
- Two common approaches to **sequence modeling** are a **generative** approach, **HMM** tagging, and a **discriminative** approach, **MEMM** tagging. We will see a third, discriminative neural approach in Chapter 9.
- The probabilities in HMM taggers are estimated by maximum likelihood estimation on tag-labeled training corpora. The Viterbi algorithm is used for **decoding**, finding the most likely tag sequence
- Beam search is a variant of Viterbi decoding that maintains only a fraction of high scoring states rather than all states during decoding.
- **Maximum entropy Markov model** or **MEMM taggers** train logistic regression models to pick the best tag given an observation word and its context and the previous tags, and then use Viterbi to choose the best sequence of tags.
- Modern taggers are generally run **bidirectionally**.

Bibliographical and Historical Notes

What is probably the earliest part-of-speech tagger was part of the parser in Zellig Harris's Transformations and Discourse Analysis Project (TDAP), implemented between June 1958 and July 1959 at the University of Pennsylvania (Harris, 1962), although earlier systems had used part-of-speech dictionaries. TDAP used 14 hand-written rules for part-of-speech disambiguation; the use of part-of-speech tag sequences and the relative frequency of tags for a word prefigures all modern algorithms. The parser was implemented essentially as a cascade of finite-state transducers; see Joshi and Hopely (1999) and Karttunen (1999) for a reimplementation.

The Computational Grammar Coder (CGC) of Klein and Simmons (1963) had three components: a lexicon, a morphological analyzer, and a context disambiguator. The small 1500-word lexicon listed only function words and other irregular words. The morphological analyzer used inflectional and derivational suffixes to assign part-of-speech classes. These were run over words to produce candidate parts-of-speech which were then disambiguated by a set of 500 context rules by relying on surrounding islands of unambiguous words. For example, one rule said that between an ARTICLE and a VERB, the only allowable sequences were ADJ-NOUN, NOUN-ADVERB, or NOUN-NOUN. The TAGGIT tagger (Greene and Rubin, 1971) used the same architecture as Klein and Simmons (1963), with a bigger dictionary and more tags (87). TAGGIT was applied to the Brown corpus and, according to Francis and Kučera (1982, p. 9), accurately tagged 77% of the corpus; the remainder of the Brown corpus was then tagged by hand. All these early algorithms were based on a two-stage architecture in which a dictionary was first used to assign each word a set of potential parts-of-speech, and then lists of hand-written disambiguation rules winnowed the set down to a single part-of-speech per word.

Soon afterwards probabilistic architectures began to be developed. Probabilities were used in tagging by Stolz et al. (1965) and a complete probabilistic tagger with Viterbi decoding was sketched by Bahl and Mercer (1976). The Lancaster-Oslo/Bergen (LOB) corpus, a British English equivalent of the Brown corpus, was tagged in the early 1980's with the CLAWS tagger (Marshall 1983; Marshall 1987; Garside 1987), a probabilistic algorithm that approximated a simplified HMM tagger. The algorithm used tag bigram probabilities, but instead of storing the word likelihood of each tag, the algorithm marked tags either as *rare* ($P(\text{tag}|\text{word}) < .01$) *infrequent* ($P(\text{tag}|\text{word}) < .10$) or *normally frequent* ($P(\text{tag}|\text{word}) > .10$).

DeRose (1988) developed a quasi-HMM algorithm, including the use of dynamic programming, although computing $P(t|w)P(w)$ instead of $P(w|t)P(w)$. The same year, the probabilistic PARTS tagger of Church (1988), (1989) was probably the first implemented HMM tagger, described correctly in Church (1989), although Church (1988) also described the computation incorrectly as $P(t|w)P(w)$ instead of $P(w|t)P(w)$. Church (p.c.) explained that he had simplified for pedagogical purposes because using the probability $P(t|w)$ made the idea seem more understandable as "storing a lexicon in an almost standard form".

Later taggers explicitly introduced the use of the hidden Markov model (Kupiec 1992; Weischedel et al. 1993; Schütze and Singer 1994). Merialdo (1994) showed that fully unsupervised EM didn't work well for the tagging task and that reliance on hand-labeled data was important. Charniak et al. (1993) showed the importance of the most frequent tag baseline; the 92.3% number we give above was from Abney et al. (1999). See Brants (2000) for many implementation details of an HMM tagger whose performance is still roughly close to state of the art taggers.

Ratnaparkhi (1996) introduced the MEMM tagger, called MXPOST, and the modern formulation is very much based on his work.

The idea of using letter suffixes for unknown words is quite old; the early Klein and Simmons (1963) system checked all final letter suffixes of lengths 1-5. The probabilistic formulation we described for HMMs comes from Samuelsson (1993). The unknown word features described on page 163 come mainly from (Ratnaparkhi, 1996), with augmentations from Toutanova et al. (2003) and Manning (2011).

State of the art taggers use neural algorithms or (bidirectional) log-linear models Toutanova et al. (2003). HMM (Brants 2000; Thede and Harper 1999) and MEMM tagger accuracies are likely just a tad lower.

An alternative modern formalism, the English Constraint Grammar systems (Karlsson et al. 1995; Voutilainen 1995; Voutilainen 1999), uses a two-stage formalism much like the early taggers from the 1950s and 1960s. A morphological analyzer with tens of thousands of English word stem entries returns all parts-of-speech for a word, using a large feature-based tagset. So the word *occurred* is tagged with the options ⟨V PCP2 SV⟩ and ⟨V PAST VFIN SV⟩, meaning it can be a participle (PCP2) for an intransitive (SV) verb, or a past (PAST) finite (VFIN) form of an intransitive (SV) verb. A set of 3,744 constraints are then applied to the input sentence to rule out parts-of-speech inconsistent with the context. For example here's a rule for the ambiguous word *that* that eliminates all tags except the ADV (adverbial intensifier) sense (this is the sense in the sentence *it isn't that odd*):

ADVERBIAL-THAT RULE **Given input**: "that"
if (+1 A/ADV/QUANT); /* *if next word is adj, adverb, or quantifier* */
(+2 SENT-LIM); /* *and following which is a sentence boundary,* */
(NOT -1 SVOC/A); /* *and the previous word is not a verb like* */
/* *'consider' which allows adjs as object complements* */
then eliminate non-ADV tags **else** eliminate ADV tag

Manning (2011) investigates the remaining 2.7% of errors in a state-of-the-art tagger, the bidirectional MEMM-style model described above (Toutanova et al., 2003). He suggests that a third or half of these remaining errors are due to errors or inconsistencies in the training data, a third might be solvable with richer linguistic models, and for the remainder the task is underspecified or unclear.

Supervised tagging relies heavily on in-domain training data hand-labeled by experts. Ways to relax this assumption include unsupervised algorithms for clustering words into part-of-speech-like classes, summarized in Christodoulopoulos et al. (2010), and ways to combine labeled and unlabeled data, for example by co-training (Clark et al. 2003; Søgaard 2010).

See Householder (1995) for historical notes on parts-of-speech, and Sampson (1987) and Garside et al. (1997) on the provenance of the Brown and other tagsets.

Exercises

8.1 Find one tagging error in each of the following sentences that are tagged with the Penn Treebank tagset:

1. I/PRP need/VBP a/DT flight/NN from/IN Atlanta/NN
2. Does/VBZ this/DT flight/NN serve/VB dinner/NNS
3. I/PRP have/VB a/DT friend/NN living/VBG in/IN Denver/NNP
4. Can/VBP you/PRP list/VB the/DT nonstop/JJ afternoon/NN flights/NNS

8.2 Use the Penn Treebank tagset to tag each word in the following sentences from Damon Runyon's short stories. You may ignore punctuation. Some of these are quite difficult; do your best.

1. It is a nice night.
2. This crap game is over a garage in Fifty-second Street...
3. ...Nobody ever takes the newspapers she sells ...
4. He is a tall, skinny guy with a long, sad, mean-looking kisser, and a mournful voice.
5. ...I am sitting in Mindy's restaurant putting on the gefillte fish, which is a dish I am very fond of, ...
6. When a guy and a doll get to taking peeks back and forth at each other, why there you are indeed.

8.3 Now compare your tags from the previous exercise with one or two friend's answers. On which words did you disagree the most? Why?

8.4 Implement the "most likely tag" baseline. Find a POS-tagged training set, and use it to compute for each word the tag that maximizes $p(t|w)$. You will need to implement a simple tokenizer to deal with sentence boundaries. Start by assuming that all unknown words are NN and compute your error rate on known and unknown words. Now write at least five rules to do a better job of tagging unknown words, and show the difference in error rates.

8.5 Build a bigram HMM tagger. You will need a part-of-speech-tagged corpus. First split the corpus into a training set and test set. From the labeled training set, train the transition and observation probabilities of the HMM tagger directly on the hand-tagged data. Then implement the Viterbi algorithm so that you can label an arbitrary test sentence. Now run your algorithm on the test set. Report its error rate and compare its performance to the most frequent tag baseline.

8.6 Do an error analysis of your tagger. Build a confusion matrix and investigate the most frequent errors. Propose some features for improving the performance of your tagger on these errors.

CHAPTER

9 Recurrent Neural Networks

Placeholder

CHAPTER

10 Formal Grammars of English

The study of grammar has an ancient pedigree; Panini's grammar of Sanskrit was written over two thousand years ago and is still referenced today in teaching Sanskrit. Despite this history, knowledge of grammar remains spotty at best. In this chapter, we make a preliminary stab at addressing some of these gaps in our knowledge of grammar and syntax, as well as introducing some of the formal mechanisms that are available for capturing this knowledge in a computationally useful manner.

syntax

The word **syntax** comes from the Greek *sýntaxis*, meaning "setting out together or arrangement", and refers to the way words are arranged together. We have seen various syntactic notions in previous chapters. The regular languages introduced in Chapter 2 offered a simple way to represent the ordering of strings of words, and Chapter 3 showed how to compute probabilities for these word sequences. Chapter 8 showed that part-of-speech categories could act as a kind of equivalence class for words. In this chapter and next few we introduce a variety of syntactic phenomena and models for syntax and grammar that go well beyond these simpler approaches.

The bulk of this chapter is devoted to the topic of context-free grammars. Context-free grammars are the backbone of many formal models of the syntax of natural language (and, for that matter, of computer languages). As such, they are integral to many computational applications, including grammar checking, semantic interpretation, dialogue understanding, and machine translation. They are powerful enough to express sophisticated relations among the words in a sentence, yet computationally tractable enough that efficient algorithms exist for parsing sentences with them (as we show in Chapter 11). In Chapter 12, we show that adding probability to context-free grammars gives us a powerful model of disambiguation. And in Chapter 15 we show how they provide a systematic framework for semantic interpretation.

In addition to an introduction to this grammar formalism, this chapter also provides a brief overview of the grammar of English. To illustrate our grammars, we have chosen a domain that has relatively simple sentences, the Air Traffic Information System (ATIS) domain (Hemphill et al., 1990). ATIS systems were an early example of spoken language systems for helping book airline reservations. Users try to book flights by conversing with the system, specifying constraints like *I'd like to fly from Atlanta to Denver*.

10.1 Constituency

The fundamental notion underlying the idea of constituency is that of abstraction — groups of words behaving as a single units, or constituents. A significant part of developing a grammar involves discovering the inventory of constituents present in the language.

noun phrase

How do words group together in English? Consider the **noun phrase**, a sequence of words surrounding at least one noun. Here are some examples of noun phrases

(thanks to Damon Runyon):

Harry the Horse	a high-class spot such as Mindy's
the Broadway coppers	the reason he comes into the Hot Box
they	three parties from Brooklyn

What evidence do we have that these words group together (or "form constituents")? One piece of evidence is that they can all appear in similar syntactic environments, for example, before a verb.

three parties from Brooklyn *arrive*...
a high-class spot such as Mindy's *attracts*...
the Broadway coppers *love*...
they *sit*

But while the whole noun phrase can occur before a verb, this is not true of each of the individual words that make up a noun phrase. The following are not grammatical sentences of English (recall that we use an asterisk (*) to mark fragments that are not grammatical English sentences):

*from *arrive*...	*as *attracts*...
*the *is*...	*spot *sat*...

Thus, to correctly describe facts about the ordering of these words in English, we must be able to say things like "*Noun Phrases can occur before verbs*".

preposed postposed

Other kinds of evidence for constituency come from what are called **preposed** or **postposed** constructions. For example, the prepositional phrase *on September seventeenth* can be placed in a number of different locations in the following examples, including at the beginning (preposed) or at the end (postposed):

On September seventeenth, I'd like to fly from Atlanta to Denver
I'd like to fly *on September seventeenth* from Atlanta to Denver
I'd like to fly from Atlanta to Denver *on September seventeenth*

But again, while the entire phrase can be placed differently, the individual words making up the phrase cannot be

*On September, I'd like to fly seventeenth from Atlanta to Denver
*On I'd like to fly September seventeenth from Atlanta to Denver
*I'd like to fly on September from Atlanta to Denver seventeenth

See Radford (1988) for further examples of groups of words behaving as a single constituent.

10.2 Context-Free Grammars

CFG

The most widely used formal system for modeling constituent structure in English and other natural languages is the **Context-Free Grammar**, or **CFG**. Context-free grammars are also called **Phrase-Structure Grammars**, and the formalism is equivalent to **Backus-Naur Form**, or **BNF**. The idea of basing a grammar on constituent structure dates back to the psychologist Wilhelm Wundt (1900) but was not formalized until Chomsky (1956) and, independently, Backus (1959).

rules

A context-free grammar consists of a set of **rules** or **productions**, each of which

lexicon
NP

expresses the ways that symbols of the language can be grouped and ordered together, and a **lexicon** of words and symbols. For example, the following productions express that an **NP** (or **noun phrase**) can be composed of either a *ProperNoun* or a determiner (*Det*) followed by a *Nominal*; a *Nominal* in turn can consist of one or more *Noun*s.

$$\begin{aligned} NP &\rightarrow Det\ Nominal \\ NP &\rightarrow ProperNoun \\ Nominal &\rightarrow Noun \mid Nominal\ Noun \end{aligned}$$

Context-free rules can be hierarchically embedded, so we can combine the previous rules with others, like the following, that express facts about the lexicon:

$$\begin{aligned} Det &\rightarrow a \\ Det &\rightarrow the \\ Noun &\rightarrow flight \end{aligned}$$

terminal

non-terminal

The symbols that are used in a CFG are divided into two classes. The symbols that correspond to words in the language ("the", "nightclub") are called **terminal** symbols; the lexicon is the set of rules that introduce these terminal symbols. The symbols that express abstractions over these terminals are called **non-terminals**. In each context-free rule, the item to the right of the arrow ($\rightarrow$) is an ordered list of one or more terminals and non-terminals; to the left of the arrow is a single non-terminal symbol expressing some cluster or generalization. Notice that in the lexicon, the non-terminal associated with each word is its lexical category, or part-of-speech, which we defined in Chapter 8.

A CFG can be thought of in two ways: as a device for generating sentences and as a device for assigning a structure to a given sentence. Viewing a CFG as a generator, we can read the $\rightarrow$ arrow as "rewrite the symbol on the left with the string of symbols on the right".

So starting from the symbol:	*NP*
we can use our first rule to rewrite *NP* as:	*Det Nominal*
and then rewrite *Nominal* as:	*Det Noun*
and finally rewrite these parts-of-speech as:	*a flight*

derivation
parse tree

We say the string *a flight* can be derived from the non-terminal *NP*. Thus, a CFG can be used to generate a set of strings. This sequence of rule expansions is called a **derivation** of the string of words. It is common to represent a derivation by a **parse tree** (commonly shown inverted with the root at the top). Figure 10.1 shows the tree representation of this derivation.

dominates

In the parse tree shown in Fig. 10.1, we can say that the node *NP* **dominates** all the nodes in the tree (*Det*, *Nom*, *Noun*, *a*, *flight*). We can say further that it immediately dominates the nodes *Det* and *Nom*.

start symbol

The formal language defined by a CFG is the set of strings that are derivable from the designated **start symbol**. Each grammar must have one designated start symbol, which is often called *S*. Since context-free grammars are often used to define sentences, *S* is usually interpreted as the "sentence" node, and the set of strings that are derivable from *S* is the set of sentences in some simplified version of English.

verb phrase

Let's add a few additional rules to our inventory. The following rule expresses the fact that a sentence can consist of a noun phrase followed by a **verb phrase**:

$$S \rightarrow NP\ VP \quad \text{I prefer a morning flight}$$

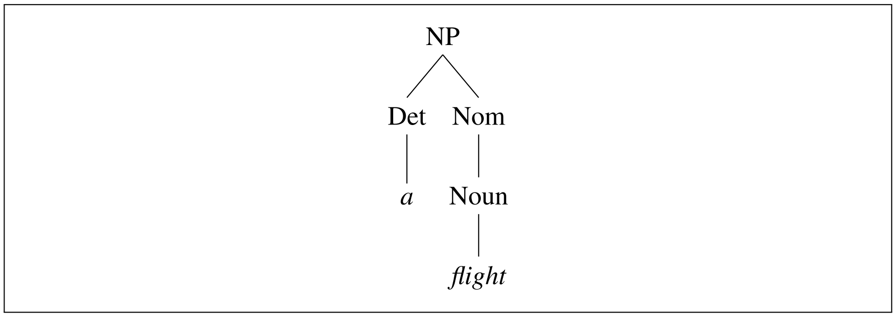


Figure 10.1 A parse tree for “a flight”.

A verb phrase in English consists of a verb followed by assorted other things; for example, one kind of verb phrase consists of a verb followed by a noun phrase:

$$VP \rightarrow Verb\ NP \quad \text{prefer a morning flight}$$

Or the verb may be followed by a noun phrase and a prepositional phrase:

$$VP \rightarrow Verb\ NP\ PP \quad \text{leave Boston in the morning}$$

Or the verb phrase may have a verb followed by a prepositional phrase alone:

$$VP \rightarrow Verb\ PP \quad \text{leaving on Thursday}$$

A prepositional phrase generally has a preposition followed by a noun phrase. For example, a common type of prepositional phrase in the ATIS corpus is used to indicate location or direction:

$$PP \rightarrow Preposition\ NP \quad \text{from Los Angeles}$$

The *NP* inside a *PP* need not be a location; *PPs* are often used with times and dates, and with other nouns as well; they can be arbitrarily complex. Here are ten examples from the ATIS corpus:

to Seattle	on these flights
in Minneapolis	about the ground transportation in Chicago
on Wednesday	of the round trip flight on United Airlines
in the evening	of the AP fifty seven flight
on the ninth of July	with a stopover in Nashville

Figure 10.2 gives a sample lexicon, and Fig. 10.3 summarizes the grammar rules we’ve seen so far, which we’ll call $\mathscr{L}_0$. Note that we can use the or-symbol | to indicate that a non-terminal has alternate possible expansions.

We can use this grammar to generate sentences of this “ATIS-language”. We start with *S*, expand it to *NP VP*, then choose a random expansion of *NP* (let’s say, to *I*), and a random expansion of *VP* (let’s say, to *Verb NP*), and so on until we generate the string *I prefer a morning flight*. Figure 10.4 shows a parse tree that represents a complete derivation of *I prefer a morning flight*.

bracketed notation

It is sometimes convenient to represent a parse tree in a more compact format called **bracketed notation**; here is the bracketed representation of the parse tree of Fig. 10.4:

(10.1) $[_S\ [_{NP}\ [_{Pro}\ \text{I}]]\ [_{VP}\ [_V\ \text{prefer}]\ [_{NP}\ [_{Det}\ \text{a}]\ [_{Nom}\ [_N\ \text{morning}]\ [_{Nom}\ [_N\ \text{flight}]]]]]]$

Noun →	*flights* | *breeze* | *trip* | *morning*
Verb →	*is* | *prefer* | *like* | *need* | *want* | *fly*
Adjective →	*cheapest* | *non-stop* | *first* | *latest*
	| *other* | *direct*
Pronoun →	*me* | *I* | *you* | *it*
Proper-Noun →	*Alaska* | *Baltimore* | *Los Angeles*
	| *Chicago* | *United* | *American*
Determiner →	*the* | *a* | *an* | *this* | *these* | *that*
Preposition →	*from* | *to* | *on* | *near*
Conjunction →	*and* | *or* | *but*

Figure 10.2 The lexicon for $\mathscr{L}_0$.

Grammar Rules			Examples
S	→	*NP VP*	I + want a morning flight
NP	→	*Pronoun*	I
	|	*Proper-Noun*	Los Angeles
	|	*Det Nominal*	a + flight
Nominal	→	*Nominal Noun*	morning + flight
	|	*Noun*	flights
VP	→	*Verb*	do
	|	*Verb NP*	want + a flight
	|	*Verb NP PP*	leave + Boston + in the morning
	|	*Verb PP*	leaving + on Thursday
PP	→	*Preposition NP*	from + Los Angeles

Figure 10.3 The grammar for $\mathscr{L}_0$, with example phrases for each rule.

A CFG like that of $\mathscr{L}_0$ defines a formal language. We saw in Chapter 2 that a formal language is a set of strings. Sentences (strings of words) that can be derived by a grammar are in the formal language defined by that grammar, and are called **grammatical** sentences. Sentences that cannot be derived by a given formal grammar are not in the language defined by that grammar and are referred to as **ungrammatical**. This hard line between "in" and "out" characterizes all formal languages but is only a very simplified model of how natural languages really work. This is because determining whether a given sentence is part of a given natural language (say, English) often depends on the context. In linguistics, the use of formal languages to model natural languages is called **generative grammar** since the language is defined by the set of possible sentences "generated" by the grammar.

grammatical

ungrammatical

generative grammar

10.2.1 Formal Definition of Context-Free Grammar

We conclude this section with a quick, formal description of a context-free grammar and the language it generates. A context-free grammar G is defined by four parameters: N, Σ, R, S (technically this is a "4-tuple").

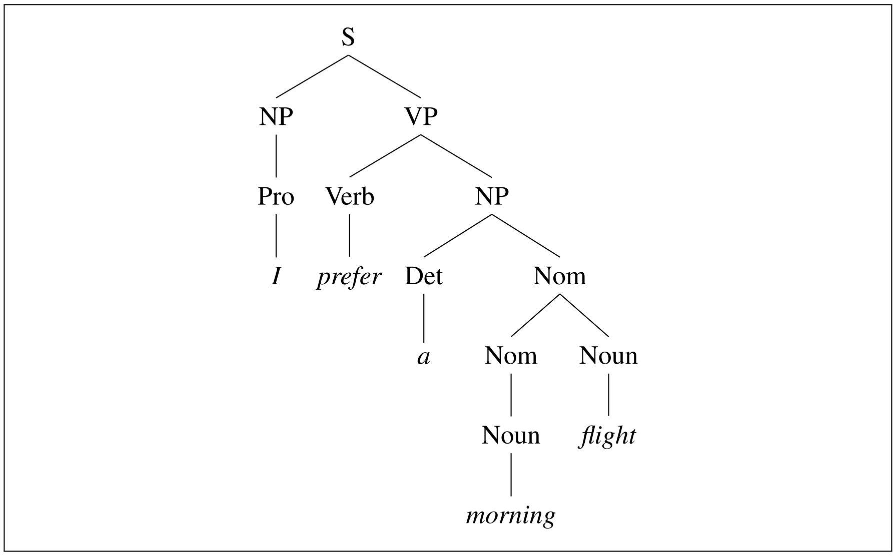


Figure 10.4 The parse tree for "I prefer a morning flight" according to grammar $\mathscr{L}_0$.

N	a set of **non-terminal symbols** (or **variables**)
Σ	a set of **terminal symbols** (disjoint from N)
R	a set of **rules** or productions, each of the form $A \rightarrow \beta$,
	where A is a non-terminal,
	β is a string of symbols from the infinite set of strings $(\Sigma \cup N)*$
S	a designated **start symbol** and a member of N

For the remainder of the book we adhere to the following conventions when discussing the formal properties of context-free grammars (as opposed to explaining particular facts about English or other languages).

Capital letters like A, B, and S	Non-terminals
S	The start symbol
Lower-case Greek letters like α, β, and γ	Strings drawn from $(\Sigma \cup N)*$
Lower-case Roman letters like u, v, and w	Strings of terminals

A language is defined through the concept of derivation. One string derives another one if it can be rewritten as the second one by some series of rule applications. More formally, following Hopcroft and Ullman (1979),

> if $A \rightarrow \beta$ is a production of R and α and γ are any strings in the set $(\Sigma \cup N)*$, then we say that $\alpha A \gamma$ **directly derives** $\alpha \beta \gamma$, or $\alpha A \gamma \Rightarrow \alpha \beta \gamma$.

directly derives

Derivation is then a generalization of direct derivation:

> Let $\alpha_1, \alpha_2, \ldots, \alpha_m$ be strings in $(\Sigma \cup N)*, m \geq 1$, such that
>
> $$\alpha_1 \Rightarrow \alpha_2, \alpha_2 \Rightarrow \alpha_3, \ldots, \alpha_{m-1} \Rightarrow \alpha_m$$
>
> We say that α_1 **derives** α_m, or $\alpha_1 \overset{*}{\Rightarrow} \alpha_m$.

derives

We can then formally define the language $\mathscr{L}_G$ generated by a grammar G as the set of strings composed of terminal symbols that can be derived from the designated

start symbol S.

$$\mathscr{L}_G = \{w|w \text{ is in } \Sigma* \text{ and } S \stackrel{*}{\Rightarrow} w\}$$

syntactic parsing

The problem of mapping from a string of words to its parse tree is called **syntactic parsing**; we define algorithms for parsing in Chapter 11.

10.3 Some Grammar Rules for English

In this section, we introduce a few more aspects of the phrase structure of English; for consistency we will continue to focus on sentences from the ATIS domain. Because of space limitations, our discussion is necessarily limited to highlights. Readers are strongly advised to consult a good reference grammar of English, such as Huddleston and Pullum (2002).

10.3.1 Sentence-Level Constructions

In the small grammar $\mathscr{L}_0$, we provided only one sentence-level construction for declarative sentences like *I prefer a morning flight*. Among the large number of constructions for English sentences, four are particularly common and important: declaratives, imperatives, yes-no questions, and wh-questions.

declarative

Sentences with **declarative** structure have a subject noun phrase followed by a verb phrase, like "I prefer a morning flight". Sentences with this structure have a great number of different uses that we follow up on in Chapter 25. Here are a number of examples from the ATIS domain:

I want a flight from Ontario to Chicago
The flight should be eleven a.m. tomorrow
The return flight should leave at around seven p.m.

imperative

Sentences with **imperative** structure often begin with a verb phrase and have no subject. They are called imperative because they are almost always used for commands and suggestions; in the ATIS domain they are commands to the system.

Show the lowest fare
Give me Sunday's flights arriving in Las Vegas from New York City
List all flights between five and seven p.m.

We can model this sentence structure with another rule for the expansion of S:

$$S \rightarrow VP$$

yes-no question

Sentences with **yes-no question** structure are often (though not always) used to ask questions; they begin with an auxiliary verb, followed by a subject *NP*, followed by a *VP*. Here are some examples. Note that the third example is not a question at all but a request; Chapter 25 discusses the uses of these question forms to perform different **pragmatic** functions such as asking, requesting, or suggesting.

Do any of these flights have stops?
Does American's flight eighteen twenty five serve dinner?
Can you give me the same information for United?

Here's the rule:

$$S \rightarrow Aux\ NP\ VP$$

wh-phrase
wh-word

The most complex sentence-level structures we examine here are the various **wh-** structures. These are so named because one of their constituents is a **wh-phrase**, that is, one that includes a **wh-word** (*who, whose, when, where, what, which, how, why*). These may be broadly grouped into two classes of sentence-level structures. The **wh-subject-question** structure is identical to the declarative structure, except that the first noun phrase contains some wh-word.

What airlines fly from Burbank to Denver?
Which flights depart Burbank after noon and arrive in Denver by six p.m?
Whose flights serve breakfast?

Here is a rule. Exercise 10.7 discusses rules for the constituents that make up the *Wh-NP*.

$$S \rightarrow \textit{Wh-NP}\ \ VP$$

wh-non-subject-question

In the **wh-non-subject-question** structure, the wh-phrase is not the subject of the sentence, and so the sentence includes another subject. In these types of sentences the auxiliary appears before the subject *NP*, just as in the yes-no question structures. Here is an example followed by a sample rule:

What flights do you have from Burbank to Tacoma Washington?

$$S \rightarrow \textit{Wh-NP Aux NP VP}$$

long-distance dependencies

Constructions like the **wh-non-subject-question** contain what are called **long-distance dependencies** because the *Wh-NP what flights* is far away from the predicate that it is semantically related to, the main verb *have* in the *VP*. In some models of parsing and understanding compatible with the grammar rule above, long-distance dependencies like the relation between *flights* and *have* are thought of as a semantic relation. In such models, the job of figuring out that *flights* is the argument of *have* is done during semantic interpretation. In other models of parsing, the relationship between *flights* and *have* is considered to be a syntactic relation, and the grammar is modified to insert a small marker called a **trace** or **empty category** after the verb. We return to such empty-category models when we introduce the Penn Treebank on page 186.

10.3.2 Clauses and Sentences

Before we move on, we should clarify the status of the *S* rules in the grammars we just described. *S* rules are intended to account for entire sentences that stand alone as fundamental units of discourse. However, *S* can also occur on the right-hand side of grammar rules and hence can be embedded within larger sentences. Clearly then, there's more to being an *S* than just standing alone as a unit of discourse.

clause

What differentiates sentence constructions (i.e., the *S* rules) from the rest of the grammar is the notion that they are in some sense *complete*. In this way they correspond to the notion of a **clause**, which traditional grammars often describe as forming a complete thought. One way of making this notion of "complete thought" more precise is to say an *S* is a node of the parse tree below which the main verb of the *S* has all of its **arguments**. We define verbal arguments later, but for now let's just see an illustration from the tree for *I prefer a morning flight* in Fig. 10.4 on page 177. The verb *prefer* has two arguments: the subject *I* and the object *a morning flight*. One of the arguments appears below the *VP* node, but the other one, the subject *NP*, appears only below the *S* node.

10.3.3 The Noun Phrase

Our $\mathscr{L}_0$ grammar introduced three of the most frequent types of noun phrases that occur in English: pronouns, proper nouns and the $NP \rightarrow Det\ \ Nominal$ construction. The central focus of this section is on the last type since that is where the bulk of the syntactic complexity resides. These noun phrases consist of a head, the central noun in the noun phrase, along with various modifiers that can occur before or after the head noun. Let's take a close look at the various parts.

The Determiner

Noun phrases can begin with simple lexical determiners, as in the following examples:

a stop	the flights	this flight
those flights	any flights	some flights

The role of the determiner in English noun phrases can also be filled by more complex expressions, as follows:

United's flight
United's pilot's union
Denver's mayor's mother's canceled flight

In these examples, the role of the determiner is filled by a possessive expression consisting of a noun phrase followed by an *'s* as a possessive marker, as in the following rule.

$$Det \rightarrow NP\ 's$$

The fact that this rule is recursive (since an *NP* can start with a *Det*) helps us model the last two examples above, in which a sequence of possessive expressions serves as a determiner.

Under some circumstances determiners are optional in English. For example, determiners may be omitted if the noun they modify is plural:

(10.2) Show me *flights* from San Francisco to Denver on weekdays

As we saw in Chapter 8, **mass nouns** also don't require determination. Recall that mass nouns often (not always) involve something that is treated like a substance (including e.g., *water* and *snow*), don't take the indefinite article "*a*", and don't tend to pluralize. Many abstract nouns are mass nouns (*music*, *homework*). Mass nouns in the ATIS domain include *breakfast*, *lunch*, and *dinner*:

(10.3) Does this flight serve dinner?

The Nominal

The nominal construction follows the determiner and contains any pre- and post-head noun modifiers. As indicated in grammar $\mathscr{L}_0$, in its simplest form a nominal can consist of a single noun.

$$Nominal \rightarrow Noun$$

As we'll see, this rule also provides the basis for the bottom of various recursive rules used to capture more complex nominal constructions.

Before the Head Noun

Cardinal numbers

ordinal numbers

quantifiers

A number of different kinds of word classes can appear before the head noun (the "postdeterminers") in a nominal. These include **cardinal numbers**, **ordinal numbers**, **quantifiers**, and adjectives. Examples of cardinal numbers:

two friends	one stop

Ordinal numbers include *first*, *second*, *third*, and so on, but also words like *next*, *last*, *past*, *other*, and *another*:

the first one	the next day	the second leg
the last flight	the other American flight	

Some quantifiers (*many*, *(a) few*, *several*) occur only with plural count nouns:

many fares

Adjectives occur after quantifiers but before nouns.

a *first-class* fare	a *non-stop* flight
the *longest* layover	the *earliest* lunch flight

adjective phrase

Adjectives can also be grouped into a phrase called an **adjective phrase** or AP. APs can have an adverb before the adjective (see Chapter 8 for definitions of adjectives and adverbs):

the *least expensive* fare

After the Head Noun

A head noun can be followed by **postmodifiers**. Three kinds of nominal postmodifiers are common in English:

prepositional phrases	all flights *from Cleveland*
non-finite clauses	any flights *arriving after eleven a.m.*
relative clauses	a flight *that serves breakfast*

common in the ATIS corpus since they are used to mark the origin and destination of flights.

Here are some examples of prepositional phrase postmodifiers, with brackets inserted to show the boundaries of each PP; note that two or more PPs can be strung together within a single NP:

all flights *[from Cleveland] [to Newark]*
arrival *[in San Jose] [before seven p.m.]*
a reservation *[on flight six oh six] [from Tampa] [to Montreal]*

Here's a new nominal rule to account for postnominal *PP*s:

$$Nominal \rightarrow Nominal\ PP$$

non-finite

The three most common kinds of **non-finite** postmodifiers are the gerundive (-*ing*), -*ed*, and infinitive forms.

gerundive

Gerundive postmodifiers are so called because they consist of a verb phrase that begins with the gerundive (-*ing*) form of the verb. Here are some examples:

any of those *[leaving on Thursday]*
any flights *[arriving after eleven a.m.]*
flights *[arriving within thirty minutes of each other]*

We can define the *Nominals* with gerundive modifiers as follows, making use of a new non-terminal *GerundVP*:

$$Nominal \rightarrow Nominal\ GerundVP$$

We can make rules for *GerundVP* constituents by duplicating all of our VP productions, substituting *GerundV* for *V*.

$$\begin{aligned} GerundVP &\rightarrow GerundV\ NP \\ &\mid \quad GerundV\ PP \mid GerundV \mid GerundV\ NP\ PP \end{aligned}$$

GerundV can then be defined as

$$GerundV \rightarrow being \mid arriving \mid leaving \mid \ldots$$

The phrases in italics below are examples of the two other common kinds of non-finite clauses, infinitives and *-ed* forms:

the last flight *to arrive in Boston*
I need to have dinner *served*
Which is the aircraft *used by this flight*?

relative pronoun

A postnominal relative clause (more correctly a **restrictive relative clause**), is a clause that often begins with a **relative pronoun** (*that* and *who* are the most common). The relative pronoun functions as the subject of the embedded verb in the following examples:

a flight *that serves breakfast*
flights *that leave in the morning*
the one *that leaves at ten thirty five*

We might add rules like the following to deal with these:

$$\begin{aligned} Nominal &\rightarrow Nominal\ RelClause \\ RelClause &\rightarrow (who \mid that)\ VP \end{aligned}$$

The relative pronoun may also function as the object of the embedded verb, as in the following example; we leave for the reader the exercise of writing grammar rules for more complex relative clauses of this kind.

the earliest American Airlines flight that I can get

Various postnominal modifiers can be combined, as the following examples show:

a flight *[from Phoenix to Detroit] [leaving Monday evening]*
evening flights *[from Nashville to Houston] [that serve dinner]*
a friend *[living in Denver] [that would like to visit me here in Washington DC]*

Before the Noun Phrase

predeterminers

Word classes that modify and appear before *NP*s are called **predeterminers**. Many of these have to do with number or amount; a common predeterminer is *all*:

all the flights   all flights   all non-stop flights

The example noun phrase given in Fig. 10.5 illustrates some of the complexity that arises when these rules are combined.

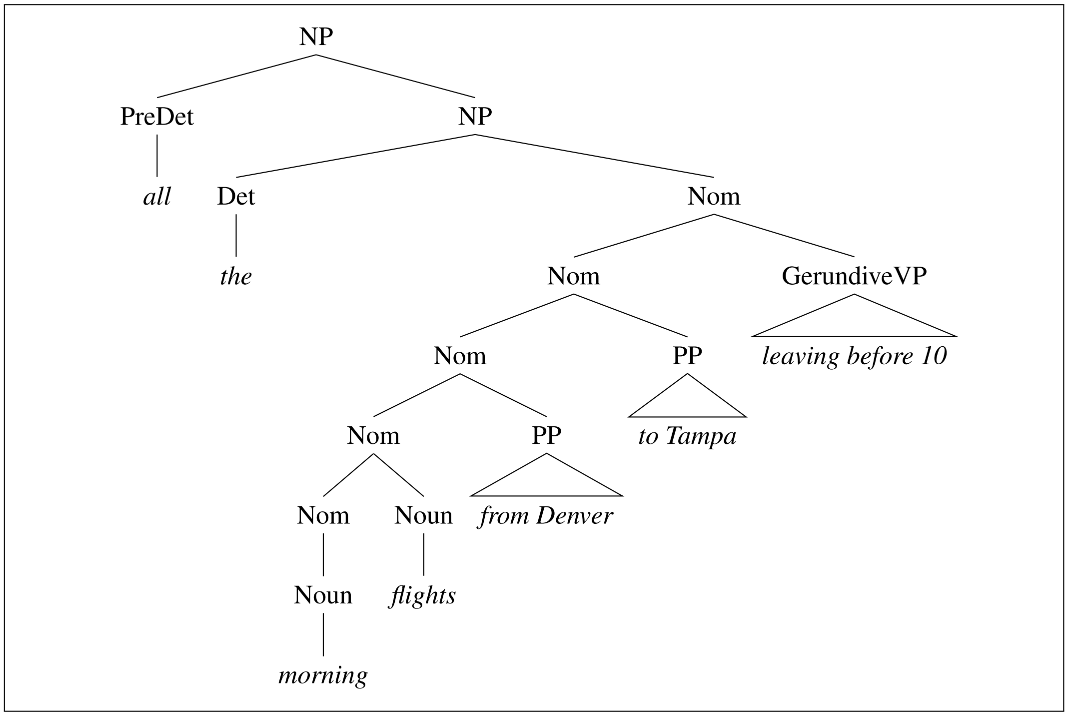


Figure 10.5 A parse tree for “all the morning flights from Denver to Tampa leaving before 10”.

10.3.4 The Verb Phrase

The verb phrase consists of the verb and a number of other constituents. In the simple rules we have built so far, these other constituents include *NP*s and *PP*s and combinations of the two:

$VP \rightarrow Verb$ disappear
$VP \rightarrow Verb\ NP$ prefer a morning flight
$VP \rightarrow Verb\ NP\ PP$ leave Boston in the morning
$VP \rightarrow Verb\ PP$ leaving on Thursday

Verb phrases can be significantly more complicated than this. Many other kinds of constituents, such as an entire embedded sentence, can follow the verb. These are called **sentential complements**:

sentential complements

You [$_{VP}$ [$_V$ said [$_S$ you had a two hundred sixty six dollar fare]]
[$_{VP}$ [$_V$ Tell] [$_{NP}$ me] [$_S$ how to get from the airport in Philadelphia to downtown]]
I [$_{VP}$ [$_V$ think [$_S$ I would like to take the nine thirty flight]]

Here’s a rule for these:

$$VP \rightarrow Verb\ S$$

Similarly, another potential constituent of the *VP* is another VP. This is often the case for verbs like *want*, *would like*, *try*, *intend*, *need*:

I want [$_{VP}$ to fly from Milwaukee to Orlando]
Hi, I want [$_{VP}$ to arrange three flights]

Frame	Verb	Example
$\emptyset$	eat, sleep	I ate
NP	prefer, find, leave	Find [$_{NP}$ the flight from Pittsburgh to Boston]
NP NP	show, give	Show [$_{NP}$ me] [$_{NP}$ airlines with flights from Pittsburgh]
PP_{from} PP_{to}	fly, travel	I would like to fly [$_{PP}$ from Boston] [$_{PP}$ to Philadelphia]
NP PP_{with}	help, load	Can you help [$_{NP}$ me] [$_{PP}$ with a flight]
VPto	prefer, want, need	I would prefer [$_{VPto}$ to go by United airlines]
VPbrst	can, would, might	I can [$_{VPbrst}$ go from Boston]
S	mean	Does this mean [$_{S}$ AA has a hub in Boston]

Figure 10.6 Subcategorization frames for a set of example verbs.

While a verb phrase can have many possible kinds of constituents, not every verb is compatible with every verb phrase. For example, the verb *want* can be used either with an *NP* complement (*I want a flight* ...) or with an infinitive *VP* complement (*I want to fly to* ...). By contrast, a verb like *find* cannot take this sort of *VP* complement (* *I found to fly to Dallas*).

transitive

intransitive

This idea that verbs are compatible with different kinds of complements is a very old one; traditional grammar distinguishes between **transitive** verbs like *find*, which take a direct object *NP* (*I found a flight*), and **intransitive** verbs like *disappear*, which do not (**I disappeared a flight*).

subcategorize

Subcategorizes for

complements

Subcategorization frame

Where traditional grammars **subcategorize** verbs into these two categories (transitive and intransitive), modern grammars distinguish as many as 100 subcategories. We say that a verb like *find* **subcategorizes for** an *NP*, and a verb like *want* subcategorizes for either an *NP* or a non-finite *VP*. We also call these constituents the **complements** of the verb (hence our use of the term **sentential complement** above). So we say that *want* can take a *VP* complement. These possible sets of complements are called the **subcategorization frame** for the verb. Another way of talking about the relation between the verb and these other constituents is to think of the verb as a logical predicate and the constituents as logical arguments of the predicate. So we can think of such predicate-argument relations as FIND(I, A FLIGHT) or WANT(I, TO FLY). We talk more about this view of verbs and arguments in Chapter 14 when we talk about predicate calculus representations of verb semantics. Subcategorization frames for a set of example verbs are given in Fig. 10.6.

We can capture the association between verbs and their complements by making separate subtypes of the class Verb (e.g., *Verb-with-NP-complement*, *Verb-with-Inf-VP-complement*, *Verb-with-S-complement*, and so on):

$$\begin{aligned} \textit{Verb-with-NP-complement} &\rightarrow \textit{find} \mid \textit{leave} \mid \textit{repeat} \mid \ldots \\ \textit{Verb-with-S-complement} &\rightarrow \textit{think} \mid \textit{believe} \mid \textit{say} \mid \ldots \\ \textit{Verb-with-Inf-VP-complement} &\rightarrow \textit{want} \mid \textit{try} \mid \textit{need} \mid \ldots \end{aligned}$$

Each *VP* rule could then be modified to require the appropriate verb subtype:

$$\begin{aligned} VP &\rightarrow \textit{Verb-with-no-complement} \quad \text{disappear} \\ VP &\rightarrow \textit{Verb-with-NP-comp NP} \quad \text{prefer a morning flight} \\ VP &\rightarrow \textit{Verb-with-S-comp S} \quad \text{said there were two flights} \end{aligned}$$

A problem with this approach is the significant increase in the number of rules and the associated loss of generality.

10.3.5 Coordination

conjunctions
coordinate

The major phrase types discussed here can be conjoined with **conjunctions** like *and*, *or*, and *but* to form larger constructions of the same type. For example, a **coordinate** noun phrase can consist of two other noun phrases separated by a conjunction:

Please repeat [$_{NP}$ [$_{NP}$ the flights] *and* [$_{NP}$ the costs]]
I need to know [$_{NP}$ [$_{NP}$ the aircraft] *and* [$_{NP}$ the flight number]]

Here's a rule that allows these structures:

$$NP \rightarrow NP \textit{ and } NP$$

Note that the ability to form coordinate phrases through conjunctions is often used as a test for constituency. Consider the following examples, which differ from the ones given above in that they lack the second determiner.

Please repeat the [$_{Nom}$ [$_{Nom}$ flights] *and* [$_{Nom}$ costs]]
I need to know the [$_{Nom}$ [$_{Nom}$ aircraft] *and* [$_{Nom}$ flight number]]

The fact that these phrases can be conjoined is evidence for the presence of the underlying *Nominal* constituent we have been making use of. Here's a new rule for this:

$$Nominal \rightarrow Nominal \textit{ and } Nominal$$

The following examples illustrate conjunctions involving *VP*s and *S*s.

What flights do you have [$_{VP}$ [$_{VP}$ leaving Denver] *and* [$_{VP}$ arriving in San Francisco]]
[$_S$ [$_S$ I'm interested in a flight from Dallas to Washington] *and* [$_S$ I'm also interested in going to Baltimore]]

The rules for *VP* and *S* conjunctions mirror the *NP* one given above.

$$VP \rightarrow VP \textit{ and } VP$$
$$S \rightarrow S \textit{ and } S$$

metarules

Since all the major phrase types can be conjoined in this fashion, it is also possible to represent this conjunction fact more generally; a number of grammar formalisms such as GPSG ((Gazdar et al., 1985)) do this using **metarules** such as the following:

$$X \rightarrow X \textit{ and } X$$

This metarule simply states that any non-terminal can be conjoined with the same non-terminal to yield a constituent of the same type. Of course, the variable *X* must be designated as a variable that stands for any non-terminal rather than a non-terminal itself.

10.4 Treebanks

Sufficiently robust grammars consisting of context-free grammar rules can be used to assign a parse tree to any sentence. This means that it is possible to build a corpus where every sentence in the collection is paired with a corresponding parse

treebank

tree. Such a syntactically annotated corpus is called a **treebank**. Treebanks play an important role in parsing, as we discuss in Chapter 11, as well as in linguistic investigations of syntactic phenomena.

Penn Treebank

A wide variety of treebanks have been created, generally through the use of parsers (of the sort described in the next few chapters) to automatically parse each sentence, followed by the use of humans (linguists) to hand-correct the parses. The **Penn Treebank** project (whose POS tagset we introduced in Chapter 8) has produced treebanks from the Brown, Switchboard, ATIS, and *Wall Street Journal* corpora of English, as well as treebanks in Arabic and Chinese. A number of treebanks use the dependency representation we will introduce in Chapter 13, including many that are part of the **Universal Dependencies** project (Nivre et al., 2016b).

10.4.1 Example: The Penn Treebank Project

Figure 10.7 shows sentences from the Brown and ATIS portions of the Penn Treebank.[1] Note the formatting differences for the part-of-speech tags; such small differences are common and must be dealt with in processing treebanks. The Penn Treebank part-of-speech tagset was defined in Chapter 8. The use of LISP-style parenthesized notation for trees is extremely common and resembles the bracketed notation we saw earlier in (10.1). For those who are not familiar with it we show a standard node-and-line tree representation in Fig. 10.8.

```
((S
   (NP-SBJ (DT That)
     (JJ cold) (, ,)
     (JJ empty) (NN sky) )
   (VP (VBD was)
     (ADJP-PRD (JJ full)
       (PP (IN of)
         (NP (NN fire)
           (CC and)
           (NN light) ))))
   (. .) ))
```
(a)

```
((S
   (NP-SBJ The/DT flight/NN )
   (VP should/MD
     (VP arrive/VB
       (PP-TMP at/IN
         (NP eleven/CD a.m/RB ))
       (NP-TMP tomorrow/NN )))))
```
(b)

Figure 10.7 Parsed sentences from the LDC Treebank3 version of the Brown (a) and ATIS (b) corpora.

traces
syntactic movement

Figure 10.9 shows a tree from the *Wall Street Journal*. This tree shows another feature of the Penn Treebanks: the use of **traces** (-NONE- nodes) to mark long-distance dependencies or **syntactic movement**. For example, quotations often follow a quotative verb like *say*. But in this example, the quotation "We would have to wait until we have collected on those assets" precedes the words *he said*. An empty *S* containing only the node -NONE- marks the position after *said* where the quotation sentence often occurs. This empty node is marked (in Treebanks II and III) with the index 2, as is the quotation *S* at the beginning of the sentence. Such co-indexing may make it easier for some parsers to recover the fact that this fronted or topicalized quotation is the complement of the verb *said*. A similar -NONE- node

[1] The Penn Treebank project released treebanks in multiple languages and in various stages; for example, there were Treebank I (Marcus et al., 1993), Treebank II (Marcus et al., 1994), and Treebank III releases of English treebanks. We use Treebank III for our examples.

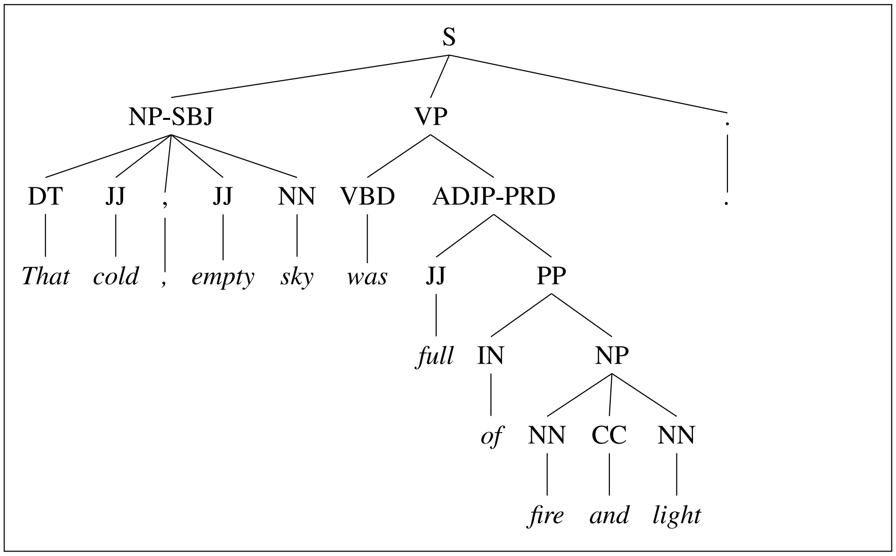


Figure 10.8 The tree corresponding to the Brown corpus sentence in the previous figure.

marks the fact that there is no syntactic subject right before the verb *to wait*; instead, the subject is the earlier *NP We*. Again, they are both co-indexed with the index 1.

```
( (S (‘‘ ‘‘)
    (S-TPC-2
      (NP-SBJ-1 (PRP We) )
      (VP (MD would)
        (VP (VB have)
          (S
            (NP-SBJ (-NONE- *-1) )
            (VP (TO to)
              (VP (VB wait)
                (SBAR-TMP (IN until)
                  (S
                    (NP-SBJ (PRP we) )
                    (VP (VBP have)
                      (VP (VBN collected)
                        (PP-CLR (IN on)
                          (NP (DT those)(NNS assets)))))))))))))
    (, ,) (’’ ’’)
    (NP-SBJ (PRP he) )
    (VP (VBD said)
      (S (-NONE- *T*-2) ))
    (. .) ))
```

Figure 10.9 A sentence from the *Wall Street Journal* portion of the LDC Penn Treebank. Note the use of the empty `-NONE-` nodes.

The Penn Treebank II and Treebank III releases added further information to make it easier to recover the relationships between predicates and arguments. Cer-

Grammar	Lexicon
$S \rightarrow NP\ VP$.	$PRP \rightarrow we \mid he$
$S \rightarrow NP\ VP$	$DT \rightarrow the \mid that \mid those$
$S \rightarrow$ " S " , $NP\ VP$.	$JJ \rightarrow cold \mid empty \mid full$
$S \rightarrow$ -NONE-	$NN \rightarrow sky \mid fire \mid light \mid flight \mid tomorrow$
$NP \rightarrow DT\ NN$	$NNS \rightarrow assets$
$NP \rightarrow DT\ NNS$	$CC \rightarrow and$
$NP \rightarrow NN\ CC\ NN$	$IN \rightarrow of \mid at \mid until \mid on$
$NP \rightarrow CD\ RB$	$CD \rightarrow eleven$
$NP \rightarrow DT\ JJ$, $JJ\ NN$	$RB \rightarrow a.m.$
$NP \rightarrow PRP$	$VB \rightarrow arrive \mid have \mid wait$
$NP \rightarrow$ -NONE-	$VBD \rightarrow was \mid said$
$VP \rightarrow MD\ VP$	$VBP \rightarrow have$
$VP \rightarrow VBD\ ADJP$	$VBN \rightarrow collected$
$VP \rightarrow VBD\ S$	$MD \rightarrow should \mid would$
$VP \rightarrow VBN\ PP$	$TO \rightarrow to$
$VP \rightarrow VB\ S$	
$VP \rightarrow VB\ SBAR$	
$VP \rightarrow VBP\ VP$	
$VP \rightarrow VBN\ PP$	
$VP \rightarrow TO\ VP$	
$SBAR \rightarrow IN\ S$	
$ADJP \rightarrow JJ\ PP$	
$PP \rightarrow IN\ NP$	

Figure 10.10 A sample of the CFG grammar rules and lexical entries that would be extracted from the three treebank sentences in Fig. 10.7 and Fig. 10.9.

tain phrases were marked with tags indicating the grammatical function of the phrase (as surface subject, logical topic, cleft, non-VP predicates) its presence in particular text categories (headlines, titles), and its semantic function (temporal phrases, locations) (Marcus et al. 1994, Bies et al. 1995). Figure 10.9 shows examples of the `-SBJ` (surface subject) and `-TMP` (temporal phrase) tags. Figure 10.8 shows in addition the `-PRD` tag, which is used for predicates that are not VPs (the one in Fig. 10.8 is an ADJP). We'll return to the topic of grammatical function when we consider dependency grammars and parsing in Chapter 13.

10.4.2 Treebanks as Grammars

The sentences in a treebank implicitly constitute a grammar of the language represented by the corpus being annotated. For example, from the three parsed sentences in Fig. 10.7 and Fig. 10.9, we can extract each of the CFG rules in them. For simplicity, let's strip off the rule suffixes (`-SBJ` and so on). The resulting grammar is shown in Fig. 10.10.

The grammar used to parse the Penn Treebank is relatively flat, resulting in very many and very long rules. For example, among the approximately 4,500 different rules for expanding VPs are separate rules for PP sequences of any length and every possible arrangement of verb arguments:

```
VP → VBD PP
VP → VBD PP PP
VP → VBD PP PP PP
VP → VBD PP PP PP PP
VP → VB ADVP PP
VP → VB PP ADVP
VP → ADVP VB PP
```

as well as even longer rules, such as

```
VP → VBP PP PP PP PP PP ADVP PP
```

which comes from the *VP* marked in italics:

> This mostly happens because we *go from football in the fall to lifting in the winter to football again in the spring*.

Some of the many thousands of *NP* rules include

```
NP → DT JJ NN
NP → DT JJ NNS
NP → DT JJ NN NN
NP → DT JJ JJ NN
NP → DT JJ CD NNS
NP → RB DT JJ NN NN
NP → RB DT JJ JJ NNS
NP → DT JJ JJ NNP NNS
NP → DT NNP NNP NNP NNP JJ NN
NP → DT JJ NNP CC JJ JJ NN NNS
NP → RB DT JJS NN NN SBAR
NP → DT VBG JJ NNP NNP CC NNP
NP → DT JJ NNS , NNS CC NN NNS NN
NP → DT JJ JJ VBG NN NNP NNP FW NNP
NP → NP JJ , JJ ‘‘ SBAR ’’ NNS
```

The last two of those rules, for example, come from the following two noun phrases:

> [$_{\text{DT}}$ The] [$_{\text{JJ}}$ state-owned] [$_{\text{JJ}}$ industrial] [$_{\text{VBG}}$ holding] [$_{\text{NN}}$ company] [$_{\text{NNP}}$ Instituto] [$_{\text{NNP}}$ Nacional] [$_{\text{FW}}$ de] [$_{\text{NNP}}$ Industria]
> [$_{\text{NP}}$ Shearson's] [$_{\text{JJ}}$ easy-to-film], [$_{\text{JJ}}$ black-and-white] "[$_{\text{SBAR}}$ Where We Stand]" [$_{\text{NNS}}$ commercials]

Viewed as a large grammar in this way, the Penn Treebank III *Wall Street Journal* corpus, which contains about 1 million words, also has about 1 million non-lexical rule tokens, consisting of about 17,500 distinct rule types.

Various facts about the treebank grammars, such as their large numbers of flat rules, pose problems for probabilistic parsing algorithms. For this reason, it is common to make various modifications to a grammar extracted from a treebank. We discuss these further in Chapter 12.

10.4.3 Heads and Head Finding

We suggested informally earlier that syntactic constituents could be associated with a lexical **head**; *N* is the head of an *NP*, *V* is the head of a *VP*. This idea of a head for each constituent dates back to Bloomfield (1914). It is central to constituent-based grammar formalisms such as Head-Driven Phrase Structure Grammar (Pollard and Sag, 1994), as well as the dependency-based approaches to grammar we'll discuss in Chapter 13. Heads and head-dependent relations have also come to play a central role in computational linguistics with their use in probabilistic parsing (Chapter 12) and in dependency parsing (Chapter 13).

In one simple model of lexical heads, each context-free rule is associated with a head (Charniak 1997, Collins 1999). The head is the word in the phrase that is grammatically the most important. Heads are passed up the parse tree; thus, each non-terminal in a parse tree is annotated with a single word, which is its lexical head.

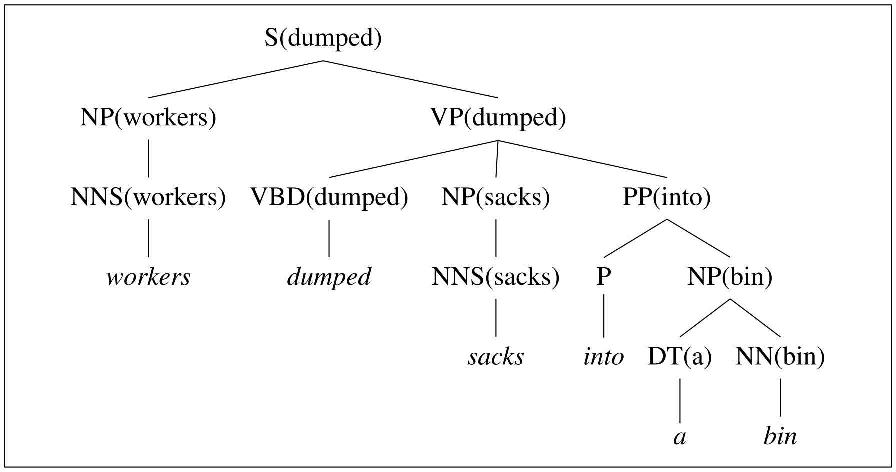


Figure 10.11 A lexicalized tree from Collins (1999).

Figure 10.11 shows an example of such a tree from Collins (1999), in which each non-terminal is annotated with its head.

For the generation of such a tree, each CFG rule must be augmented to identify one right-side constituent to be the head daughter. The headword for a node is then set to the headword of its head daughter. Choosing these head daughters is simple for textbook examples (*NN* is the head of *NP*) but is complicated and indeed controversial for most phrases. (Should the complementizer *to* or the verb be the head of an infinite verb-phrase?) Modern linguistic theories of syntax generally include a component that defines heads (see, e.g., (Pollard and Sag, 1994)).

An alternative approach to finding a head is used in most practical computational systems. Instead of specifying head rules in the grammar itself, heads are identified dynamically in the context of trees for specific sentences. In other words, once a sentence is parsed, the resulting tree is walked to decorate each node with the appropriate head. Most current systems rely on a simple set of hand-written rules, such as a practical one for Penn Treebank grammars given in Collins (1999) but developed originally by Magerman (1995). For example, the rule for finding the head of an *NP* is as follows (Collins, 1999, p. 238):

- If the last word is tagged POS, return last-word.
- Else search from right to left for the first child which is an NN, NNP, NNPS, NX, POS, or JJR.
- Else search from left to right for the first child which is an NP.
- Else search from right to left for the first child which is a $, ADJP, or PRN.
- Else search from right to left for the first child which is a CD.
- Else search from right to left for the first child which is a JJ, JJS, RB or QP.
- Else return the last word

Selected other rules from this set are shown in Fig. 10.12. For example, for *VP* rules of the form $VP \rightarrow Y_1 \cdots Y_n$, the algorithm would start from the left of $Y_1 \cdots Y_n$ looking for the first Y_i of type TO; if no TOs are found, it would search for the first Y_i of type VBD; if no VBDs are found, it would search for a VBN, and so on. See Collins (1999) for more details.

Parent	Direction	Priority List
ADJP	Left	NNS QP NN $ ADVP JJ VBN VBG ADJP JJR NP JJS DT FW RBR RBS SBAR RB
ADVP	Right	RB RBR RBS FW ADVP TO CD JJR JJ IN NP JJS NN
PRN	Left	
PRT	Right	RP
QP	Left	$ IN NNS NN JJ RB DT CD NCD QP JJR JJS
S	Left	TO IN VP S SBAR ADJP UCP NP
SBAR	Left	WHNP WHPP WHADVP WHADJP IN DT S SQ SINV SBAR FRAG
VP	Left	TO VBD VBN MD VBZ VB VBG VBP VP ADJP NN NNS NP

Figure 10.12 Selected head rules from Collins (1999). The set of head rules is often called a **head percolation table**.

10.5 Grammar Equivalence and Normal Form

A formal language is defined as a (possibly infinite) set of strings of words. This suggests that we could ask if two grammars are equivalent by asking if they generate the same set of strings. In fact, it is possible to have two distinct context-free grammars generate the same language.

We usually distinguish two kinds of grammar equivalence: **weak equivalence** and **strong equivalence**. Two grammars are strongly equivalent if they generate the same set of strings *and* if they assign the same phrase structure to each sentence (allowing merely for renaming of the non-terminal symbols). Two grammars are weakly equivalent if they generate the same set of strings but do not assign the same phrase structure to each sentence.

normal form

It is sometimes useful to have a **normal form** for grammars, in which each of the productions takes a particular form. For example, a context-free grammar is in **Chomsky normal form** (CNF) (Chomsky, 1963) if it is ϵ-free and if in addition each production is either of the form $A \rightarrow B\ C$ or $A \rightarrow a$. That is, the right-hand side of each rule either has two non-terminal symbols or one terminal symbol. Chomsky normal form grammars are **binary branching**, that is they have binary trees (down to the prelexical nodes). We make use of this binary branching property in the CKY parsing algorithm in Chapter 11.

Chomsky normal form

binary branching

Any context-free grammar can be converted into a weakly equivalent Chomsky normal form grammar. For example, a rule of the form

$$A \rightarrow B\ C\ D$$

can be converted into the following two CNF rules (Exercise 10.8 asks the reader to formulate the complete algorithm):

$$\begin{aligned} A &\rightarrow B\ X \\ X &\rightarrow C\ D \end{aligned}$$

Sometimes using binary branching can actually produce smaller grammars. For example, the sentences that might be characterized as

```
VP -> VBD NP PP*
```

are represented in the Penn Treebank by this series of rules:

```
VP → VBD NP PP
VP → VBD NP PP PP
```

```
VP → VBD NP PP PP PP
VP → VBD NP PP PP PP PP
...
```

but could also be generated by the following two-rule grammar:

```
VP → VBD NP PP
VP → VP PP
```

Chomsky-adjunction

The generation of a symbol A with a potentially infinite sequence of symbols B with a rule of the form `A → A B` is known as **Chomsky-adjunction**.

10.6 Lexicalized Grammars

The approach to grammar presented thus far emphasizes phrase-structure rules while minimizing the role of the lexicon. However, as we saw in the discussions of agreement, subcategorization, and long distance dependencies, this approach leads to solutions that are cumbersome at best, yielding grammars that are redundant, hard to manage, and brittle. To overcome these issues, numerous alternative approaches have been developed that all share the common theme of making better use of the lexicon. Among the more computationally relevant approaches are Lexical-Functional Grammar (LFG) (Bresnan, 1982), Head-Driven Phrase Structure Grammar (HPSG) (Pollard and Sag, 1994), Tree-Adjoining Grammar (TAG) (Joshi, 1985), and Combinatory Categorial Grammar (CCG). These approaches differ with respect to how *lexicalized* they are — the degree to which they rely on the lexicon as opposed to phrase structure rules to capture facts about the language.

The following section provides an introduction to CCG, a heavily lexicalized approach motivated by both syntactic and semantic considerations, which we will return to in Chapter 14. Chapter 13 discusses dependency grammars, an approach that eliminates phrase-structure rules entirely.

10.6.1 Combinatory Categorial Grammar

categorial grammar

combinatory categorial grammar

In this section, we provide an overview of **categorial grammar** (Ajdukiewicz 1935, Bar-Hillel 1953), an early lexicalized grammar model, as well as an important modern extension, **combinatory categorial grammar**, or CCG (Steedman 1996,Steedman 1989,Steedman 2000).

The categorial approach consists of three major elements: a set of categories, a lexicon that associates words with categories, and a set of rules that govern how categories combine in context.

Categories

Categories are either atomic elements or single-argument functions that return a category as a value when provided with a desired category as argument. More formally, we can define $\mathscr{C}$, a set of categories for a grammar as follows:

- $\mathscr{A} \subseteq \mathscr{C}$, where $\mathscr{A}$ is a given set of atomic elements
- (X/Y), $(X\backslash Y) \in \mathscr{C}$, if X, $Y \in \mathscr{C}$

The slash notation shown here is used to define the functions in the grammar. It specifies the type of the expected argument, the direction it is expected be found, and the type of the result. Thus, (X/Y) is a function that seeks a constituent of type

Y to its right and returns a value of X; $(X\backslash Y)$ is the same except it seeks its argument to the left.

The set of atomic categories is typically very small and includes familiar elements such as sentences and noun phrases. Functional categories include verb phrases and complex noun phrases among others.

The Lexicon

The lexicon in a categorial approach consists of assignments of categories to words. These assignments can either be to atomic or functional categories, and due to lexical ambiguity words can be assigned to multiple categories. Consider the following sample lexical entries.

$$
\begin{aligned}
\textit{flight}: &\quad N \\
\textit{Miami}: &\quad NP \\
\textit{cancel}: &\quad (S\backslash NP)/NP
\end{aligned}
$$

Nouns and proper nouns like *flight* and *Miami* are assigned to atomic categories, reflecting their typical role as arguments to functions. On the other hand, a transitive verb like *cancel* is assigned the category $(S\backslash NP)/NP$: a function that seeks an NP on its right and returns as its value a function with the type $(S\backslash NP)$. This function can, in turn, combine with an NP on the left, yielding an S as the result. This captures the kind of subcategorization information discussed in Section 10.3.4, however here the information has a rich, computationally useful, internal structure.

Ditransitive verbs like *give*, which expect two arguments after the verb, would have the category $((S\backslash NP)/NP)/NP$: a function that combines with an NP on its right to yield yet another function corresponding to the transitive verb $(S\backslash NP)/NP$ category such as the one given above for *cancel*.

Rules

The rules of a categorial grammar specify how functions and their arguments combine. The following two rule templates constitute the basis for all categorial grammars.

$$X/Y \;\; Y \;\Rightarrow\; X \tag{10.4}$$

$$Y \;\; X\backslash Y \;\Rightarrow\; X \tag{10.5}$$

The first rule applies a function to its argument on the right, while the second looks to the left for its argument. We'll refer to the first as **forward function application**, and the second as **backward function application**. The result of applying either of these rules is the category specified as the value of the function being applied.

Given these rules and a simple lexicon, let's consider an analysis of the sentence *United serves Miami*. Assume that *serves* is a transitive verb with the category $(S\backslash NP)/NP$ and that *United* and *Miami* are both simple NPs. Using both forward and backward function application, the derivation would proceed as follows:

$$
\begin{array}{ccc}
\textit{United} & \textit{serves} & \textit{Miami} \\
\hline
\mathrm{NP} & \mathrm{(S\backslash NP)/NP} & \mathrm{NP} \\
\cline{2-3}
 & \multicolumn{2}{c}{\mathrm{S\backslash NP}} \;> \\
\hline
\multicolumn{3}{c}{\mathrm{S}} \;<
\end{array}
$$

Categorial grammar derivations are illustrated growing down from the words, rule applications are illustrated with a horizontal line that spans the elements involved, with the type of the operation indicated at the right end of the line. In this example, there are two function applications: one forward function application indicated by the > that applies the verb *serves* to the *NP* on its right, and one backward function application indicated by the < that applies the result of the first to the *NP United* on its left.

With the addition of another rule, the categorial approach provides a straightforward way to implement the coordination metarule described earlier on page 185. Recall that English permits the coordination of two constituents of the same type, resulting in a new constituent of the same type. The following rule provides the mechanism to handle such examples.

$$X \;\; CONJ \;\; X \;\; \Rightarrow \;\; X \tag{10.6}$$

This rule states that when two constituents of the same category are separated by a constituent of type *CONJ* they can be combined into a single larger constituent of the same type. The following derivation illustrates the use of this rule.

<table>
<tr><td>We</td><td>flew</td><td>to</td><td>Geneva</td><td>and</td><td>drove</td><td>to</td><td>Chamonix</td><td></td></tr>
<tr><td>NP</td><td>(S\NP)/PP</td><td>PP/NP</td><td>NP</td><td>CONJ</td><td>(S\NP)/PP</td><td>PP/NP</td><td>NP</td><td></td></tr>
<tr><td></td><td></td><td colspan="2">PP (>)</td><td></td><td></td><td colspan="2">PP (>)</td><td></td></tr>
<tr><td></td><td colspan="3">S\NP (>)</td><td></td><td colspan="3">S\NP (>)</td><td></td></tr>
<tr><td></td><td colspan="7">S\NP (<Φ>)</td><td></td></tr>
<tr><td colspan="8">S (<)</td><td></td></tr>
</table>

Here the two *S\NP* constituents are combined via the conjunction operator <Φ> to form a larger constituent of the same type, which can then be combined with the subject *NP* via backward function application.

These examples illustrate the lexical nature of the categorial grammar approach. The grammatical facts about a language are largely encoded in the lexicon, while the rules of the grammar are boiled down to a set of three rules. Unfortunately, the basic categorial approach does not give us any more expressive power than we had with traditional CFG rules; it just moves information from the grammar to the lexicon. To move beyond these limitations CCG includes operations that operate over functions.

The first pair of operators permit us to **compose** adjacent functions.

$$X/Y \;\; Y/Z \;\; \Rightarrow \;\; X/Z \tag{10.7}$$

$$Y\backslash Z \;\; X\backslash Y \;\; \Rightarrow \;\; X\backslash Z \tag{10.8}$$

forward composition

The first rule, called **forward composition**, can be applied to adjacent constituents where the first is a function seeking an argument of type *Y* to its right, and the second is a function that provides*Y* as a result. This rule allows us to compose these two functions into a single one with the type of the first constituent and the argument of the second. Although the notation is a little awkward, the second rule, **backward composition** is the same, except that we're looking to the left instead of to the right for the relevant arguments. Both kinds of composition are signalled by a **B** in CCG diagrams, accompanied by a < or > to indicate the direction.

backward composition

type raising

The next operator is **type raising**. Type raising elevates simple categories to the status of functions. More specifically, type raising takes a category and converts it to function that seeks as an argument a function that takes the original category

as its argument. The following schema show two versions of type raising: one for arguments to the right, and one for the left.

$$X \;\Rightarrow\; T/(T\backslash X) \tag{10.9}$$

$$X \;\Rightarrow\; T\backslash(T/X) \tag{10.10}$$

The category T in these rules can correspond to any of the atomic or functional categories already present in the grammar.

A particularly useful example of type raising transforms a simple *NP* argument in subject position to a function that can compose with a following *VP*. To see how this works, let's revisit our earlier example of *United serves Miami*. Instead of classifying *United* as an *NP* which can serve as an argument to the function attached to *serve*, we can use type raising to reinvent it as a function in its own right as follows.

$$NP \;\Rightarrow\; S/(S\backslash NP)$$

Combining this type-raised constituent with the forward composition rule (10.7) permits the following alternative to our previous derivation.

$$\begin{array}{ccc}
\textit{United} & \textit{serves} & \textit{Miami} \\
\hline
\mathrm{NP} & \mathrm{(S\backslash NP)/NP} & \mathrm{NP} \\
\overline{\mathrm{S/(S\backslash NP)}}^{\,>\mathbf{T}} & & \\
\multicolumn{2}{c}{\overline{\mathrm{S/NP}}^{\,>\mathbf{B}}} & \\
\multicolumn{3}{c}{\overline{\mathrm{S}}^{\,>}}
\end{array}$$

By type raising *United* to $S/(S\backslash NP)$, we can compose it with the transitive verb *serves* to yield the (S/NP) function needed to complete the derivation.

There are several interesting things to note about this derivation. First, is it provides a left-to-right, word-by-word derivation that more closely mirrors the way humans process language. This makes CCG a particularly apt framework for psycholinguistic studies. Second, this derivation involves the use of an intermediate unit of analysis, *United serves*, that does not correspond to a traditional constituent in English. This ability to make use of such non-constituent elements provides CCG with the ability to handle the coordination of phrases that are not proper constituents, as in the following example.

(10.11) We flew IcelandAir to Geneva and SwissAir to London.

Here, the segments that are being coordinated are *IcelandAir to Geneva* and *SwissAir to London*, phrases that would not normally be considered constituents, as can be seen in the following standard derivation for the verb phrase *flew IcelandAir to Geneva*.

$$\begin{array}{cccc}
\textit{flew} & \textit{IcelandAir} & \textit{to} & \textit{Geneva} \\
\hline
\mathrm{(VP/PP)/NP} & \mathrm{NP} & \mathrm{PP/NP} & \mathrm{NP} \\
\multicolumn{2}{c}{\overline{\mathrm{VP/PP}}^{\,>}} & \multicolumn{2}{c}{\overline{\mathrm{PP}}^{\,>}} \\
\multicolumn{4}{c}{\overline{\mathrm{VP}}^{\,>}}
\end{array}$$

In this derivation, there is no single constituent that corresponds to *IcelandAir to Geneva*, and hence no opportunity to make use of the <Φ> operator. Note that complex CCG categories can can get a little cumbersome, so we'll use *VP* as a shorthand for $(S\backslash NP)$ in this and the following derivations.

The following alternative derivation provides the required element through the use of both backward type raising (10.10) and backward function composition (10.8).

$$
\begin{array}{cccc}
\text{flew} & \text{IcelandAir} & \text{to} & \text{Geneva} \\
\hline
(VP/PP)/NP & NP & PP/NP & NP \\
 & \overline{(VP/PP)\backslash((VP/PP)/NP)}^{<\mathbf{T}} & \multicolumn{2}{c}{\overline{PP}^{>}} \\
 & & \multicolumn{2}{c}{\overline{VP\backslash(VP/PP)}^{<\mathbf{T}}} \\
 & \multicolumn{3}{c}{\overline{VP\backslash((VP/PP)/NP)}^{<\mathbf{B}}} \\
\multicolumn{4}{c}{\overline{VP}^{<}}
\end{array}
$$

Applying the same analysis to *SwissAir to London* satisfies the requirements for the <Φ> operator, yielding the following derivation for our original example (10.11).

$$
\begin{array}{cccccccc}
\textit{flew} & \textit{IcelandAir} & \textit{to} & \textit{Geneva} & \textit{and} & \textit{SwissAir} & \textit{to} & \textit{London} \\
\hline
(VP/PP)/NP & NP & PP/NP & NP & CONJ & NP & PP/NP & NP \\
 & \overline{(VP/PP)\backslash((VP/PP)/NP)}^{<\mathbf{T}} & \multicolumn{2}{c}{\overline{PP}^{>}} & & \overline{(VP/PP)\backslash((VP/PP)/NP)}^{<\mathbf{T}} & \multicolumn{2}{c}{\overline{PP}^{>}} \\
 & & \multicolumn{2}{c}{\overline{VP\backslash(VP/PP)}^{<\mathbf{T}}} & & & \multicolumn{2}{c}{\overline{VP\backslash(VP/PP)}^{<\mathbf{T}}} \\
 & \multicolumn{3}{c}{\overline{VP\backslash((VP/PP)/NP)}^{<}} & & \multicolumn{3}{c}{\overline{VP\backslash((VP/PP)/NP)}^{<}} \\
 & \multicolumn{7}{c}{\overline{VP\backslash((VP/PP)/NP)}^{<\Phi>}} \\
\multicolumn{8}{c}{\overline{VP}^{<}}
\end{array}
$$

Finally, let's examine how these advanced operators can be used to handle **long-distance dependencies** (also referred to as syntactic movement or extraction). As mentioned in Section 10.3.1, long-distance dependencies arise from many English constructions including wh-questions, relative clauses, and topicalization. What these constructions have in common is a constituent that appears somewhere distant from its usual, or expected, location. Consider the following relative clause as an example.

the flight that United diverted

Here, *divert* is a transitive verb that expects two *NP* arguments, a subject *NP* to its left and a direct object *NP* to its right; its category is therefore $(S\backslash NP)/NP$. However, in this example the direct object *the flight* has been "moved" to the beginning of the clause, while the subject *United* remains in its normal position. What is needed is a way to incorporate the subject argument, while dealing with the fact that *the flight* is not in its expected location.

The following derivation accomplishes this, again through the combined use of type raising and function composition.

$$
\begin{array}{ccccc}
\textit{the} & \textit{flight} & \textit{that} & \textit{United} & \textit{diverted} \\
\hline
\mathrm{NP/N} & \mathrm{N} & \mathrm{(NP\backslash NP)/(S/NP)} & \mathrm{NP} & \mathrm{(S\backslash NP)/NP} \\
\multicolumn{2}{c}{\overline{\mathrm{NP}}^{>}} & & \overline{\mathrm{S/(S\backslash NP)}}^{>\mathbf{T}} & \\
 & & & \multicolumn{2}{c}{\overline{\mathrm{S/NP}}^{>\mathbf{B}}} \\
 & & \multicolumn{3}{c}{\overline{\mathrm{NP\backslash NP}}^{>}} \\
\multicolumn{5}{c}{\overline{\mathrm{NP}}^{<}}
\end{array}
$$

As we saw with our earlier examples, the first step of this derivation is type raising *United* to the category $S/(S\backslash NP)$ allowing it to combine with *diverted* via forward composition. The result of this composition is S/NP which preserves the fact that we are still looking for an *NP* to fill the missing direct object. The second critical piece is the lexical category assigned to the word *that*: $(NP\backslash NP)/(S/NP)$. This function seeks a verb phrase missing an argument to its right, and transforms it into an *NP* seeking a missing element to its left, precisely where we find *the flight*.

CCGBank

As with phrase-structure approaches, treebanks play an important role in CCG-based approaches to parsing. CCGBank (Hockenmaier and Steedman, 2007) is the largest and most widely used CCG treebank. It was created by automatically translating phrase-structure trees from the Penn Treebank via a rule-based approach. The method produced successful translations of over 99% of the trees in the Penn Treebank resulting in 48,934 sentences paired with CCG derivations. It also provides a lexicon of 44,000 words with over 1200 categories. Chapter 12 will discuss how these resources can be used to train CCG parsers.

10.7 Summary

This chapter has introduced a number of fundamental concepts in syntax through the use of **context-free grammars**.

- In many languages, groups of consecutive words act as a group or a **constituent**, which can be modeled by **context-free grammars** (which are also known as **phrase-structure grammars**).
- A context-free grammar consists of a set of **rules** or **productions**, expressed over a set of **non-terminal** symbols and a set of **terminal** symbols. Formally, a particular **context-free language** is the set of strings that can be **derived** from a particular **context-free grammar**.
- A **generative grammar** is a traditional name in linguistics for a formal language that is used to model the grammar of a natural language.
- There are many sentence-level grammatical constructions in English; **declarative**, **imperative**, **yes-no question**, and **wh-question** are four common types; these can be modeled with context-free rules.
- An English **noun phrase** can have **determiners**, **numbers**, **quantifiers**, and **adjective phrases** preceding the **head noun**, which can be followed by a number of **postmodifiers**; **gerundive** VPs, **infinitives** VPs, and **past participial** VPs are common possibilities.
- **Subjects** in English **agree** with the main verb in person and number.
- Verbs can be **subcategorized** by the types of **complements** they expect. Simple subcategories are **transitive** and **intransitive**; most grammars include many more categories than these.
- **Treebanks** of parsed sentences exist for many genres of English and for many languages. Treebanks can be searched with tree-search tools.
- Any context-free grammar can be converted to **Chomsky normal form**, in which the right-hand side of each rule has either two non-terminals or a single terminal.
- Lexicalized grammars place more emphasis on the structure of the lexicon, lessening the burden on pure phrase-structure rules.
- Combinatorial categorial grammar (CCG) is an important computationally relevant lexicalized approach.

Bibliographical and Historical Notes

[The origin of the idea of phrasal constituency, cited in Percival (1976)]:
den sprachlichen Ausdruck für die willkürliche
Gliederung einer Gesammtvorstellung in ihre
in logische Beziehung zueinander gesetzten Bestandteile'
[the linguistic expression for the arbitrary division of a total idea
into its constituent parts placed in logical relations to one another]
W. Wundt

According to Percival (1976), the idea of breaking up a sentence into a hierarchy of constituents appeared in the *Völkerpsychologie* of the groundbreaking psychologist Wilhelm Wundt (Wundt, 1900). Wundt's idea of constituency was taken up into linguistics by Leonard Bloomfield in his early book *An Introduction to the Study of Language* (Bloomfield, 1914). By the time of his later book, *Language* (Bloomfield, 1933a), what was then called "immediate-constituent analysis" was a well-established method of syntactic study in the United States. By contrast, traditional European grammar, dating from the Classical period, defined relations between *words* rather than constituents, and European syntacticians retained this emphasis on such **dependency** grammars, the subject of Chapter 13.

American Structuralism saw a number of specific definitions of the immediate constituent, couched in terms of their search for a "discovery procedure": a methodological algorithm for describing the syntax of a language. In general, these attempt to capture the intuition that "The primary criterion of the immediate constituent is the degree in which combinations behave as simple units" (Bazell, 1966, p. 284). The most well known of the specific definitions is Harris' idea of distributional similarity to individual units, with the *substitutability* test. Essentially, the method proceeded by breaking up a construction into constituents by attempting to substitute simple structures for possible constituents—if a substitution of a simple form, say, *man*, was substitutable in a construction for a more complex set (like *intense young man*), then the form *intense young man* was probably a constituent. Harris's test was the beginning of the intuition that a constituent is a kind of equivalence class.

The first formalization of this idea of hierarchical constituency was the **phrase-structure grammar** defined in Chomsky (1956) and further expanded upon (and argued against) in Chomsky (1957) and Chomsky (1975). From this time on, most generative linguistic theories were based at least in part on context-free grammars or generalizations of them (such as Head-Driven Phrase Structure Grammar (Pollard and Sag, 1994), Lexical-Functional Grammar (Bresnan, 1982), Government and Binding (Chomsky, 1981), and Construction Grammar (Kay and Fillmore, 1999), inter alia); many of these theories used schematic context-free templates known as **X-bar schemata**, which also relied on the notion of syntactic head.

X-bar schemata

Shortly after Chomsky's initial work, the context-free grammar was reinvented by Backus (1959) and independently by Naur et al. (1960) in their descriptions of the ALGOL programming language; Backus (1996) noted that he was influenced by the productions of Emil Post and that Naur's work was independent of his (Backus') own. (Recall the discussion on page **??** of multiple invention in science.) After this early work, a great number of computational models of natural language processing were based on context-free grammars because of the early development of efficient algorithms to parse these grammars (see Chapter 11).

As we have already noted, grammars based on context-free rules are not ubiquitous. Various classes of extensions to CFGs are designed specifically to handle long-distance dependencies. We noted earlier that some grammars treat long-distance-dependent items as being related semantically but not syntactically; the surface syntax does not represent the long-distance link (Kay and Fillmore 1999, Culicover and Jackendoff 2005). But there are alternatives.

One extended formalism is **Tree Adjoining Grammar** (TAG) (Joshi, 1985). The primary TAG data structure is the tree, rather than the rule. Trees come in two kinds: **initial trees** and **auxiliary trees**. Initial trees might, for example, represent simple sentential structures, and auxiliary trees add recursion into a tree. Trees are combined by two operations called **substitution** and **adjunction**. The adjunction operation handles long-distance dependencies. See Joshi (1985) for more details. An extension of Tree Adjoining Grammar, called Lexicalized Tree Adjoining Grammars is discussed in Chapter 12. Tree Adjoining Grammar is a member of the family of **mildly context-sensitive languages**.

We mentioned on page 186 another way of handling long-distance dependencies, based on the use of empty categories and co-indexing. The Penn Treebank uses this model, which draws (in various Treebank corpora) from the Extended Standard Theory and Minimalism (Radford, 1997).

Readers interested in the grammar of English should get one of the three large reference grammars of English: Huddleston and Pullum (2002), Biber et al. (1999), and Quirk et al. (1985). Another useful reference is McCawley (1998).

generative

functional

There are many good introductory textbooks on syntax from different perspectives. Sag et al. (2003) is an introduction to syntax from a **generative** perspective, focusing on the use of phrase-structure rules, unification, and the type hierarchy in Head-Driven Phrase Structure Grammar. Van Valin, Jr. and La Polla (1997) is an introduction from a **functional** perspective, focusing on cross-linguistic data and on the functional motivation for syntactic structures.

Exercises

10.1 Draw tree structures for the following ATIS phrases:

1. Dallas
2. from Denver
3. after five p.m.
4. arriving in Washington
5. early flights
6. all redeye flights
7. on Thursday
8. a one-way fare
9. any delays in Denver

10.2 Draw tree structures for the following ATIS sentences:

1. Does American airlines have a flight between five a.m. and six a.m.?
2. I would like to fly on American airlines.
3. Please repeat that.
4. Does American 487 have a first-class section?
5. I need to fly between Philadelphia and Atlanta.
6. What is the fare from Atlanta to Denver?

7. Is there an American airlines flight from Philadelphia to Dallas?

10.3 Assume a grammar that has many *VP* rules for different subcategorizations, as expressed in Section 10.3.4, and differently subcategorized verb rules like *Verb-with-NP-complement*. How would the rule for postnominal relative clauses (10.4) need to be modified if we wanted to deal properly with examples like *the earliest flight that you have*? Recall that in such examples the pronoun *that* is the object of the verb *get*. Your rules should allow this noun phrase but should correctly rule out the ungrammatical *S *I get*.

10.4 Does your solution to the previous problem correctly model the NP *the earliest flight that I can get*? How about *the earliest flight that I think my mother wants me to book for her*? Hint: this phenomenon is called **long-distance dependency**.

10.5 Write rules expressing the verbal subcategory of English auxiliaries; for example, you might have a rule *verb-with-bare-stem-VP-complement* → *can*.

possessive
genitive

10.6 *NP*s like *Fortune's office* or *my uncle's marks* are called **possessive** or **genitive** noun phrases. We can model possessive noun phrases by treating the sub-NP like *Fortune's* or *my uncle's* as a determiner of the following head noun. Write grammar rules for English possessives. You may treat *'s* as if it were a separate word (i.e., as if there were always a space before *'s*).

10.7 Page 179 discussed the need for a *Wh-NP* constituent. The simplest *Wh-NP* is one of the *Wh-pronouns* (*who, whom, whose, which*). The Wh-words *what* and *which* can be determiners: *which four will you have?*, *what credit do you have with the Duke?* Write rules for the different types of *Wh-NP*s.

10.8 Write an algorithm for converting an arbitrary context-free grammar into Chomsky normal form.

CHAPTER

11 Syntactic Parsing

One morning I shot an elephant in my pajamas.
How he got into my pajamas I don't know.
Groucho Marx, *Animal Crackers*, 1930

Syntactic parsing is the task of recognizing a sentence and assigning a syntactic structure to it. This chapter focuses on the structures assigned by context-free grammars of the kind described in Chapter 10. Since they are based on a purely declarative formalism, context-free grammars don't specify *how* the parse tree for a given sentence should be computed. We therefore need to specify algorithms that employ these grammars to efficiently produce correct trees.

Parse trees are directly useful in applications such as **grammar checking** in word-processing systems: a sentence that cannot be parsed may have grammatical errors (or at least be hard to read). More typically, however, parse trees serve as an important intermediate stage of representation for **semantic analysis** (as we show in Chapter 15) and thus play an important role in applications like **question answering** and **information extraction**. For example, to answer the question

What books were written by British women authors before 1800?

we'll need to know that the subject of the sentence was *what books* and that the by-adjunct was *British women authors* to help us figure out that the user wants a list of books (and not a list of authors).

Before presenting any algorithms, we begin by discussing how the ambiguity arises again in this context and the problems it presents. The section that follows then presents the Cocke-Kasami-Younger (CKY) algorithm (Kasami 1965, Younger 1967), the standard dynamic programming approach to syntactic parsing. Recall that we've already seen applications of dynamic programming algorithms in the Minimum-Edit-Distance and Viterbi algorithms of earlier chapters. Finally, we discuss **partial parsing methods**, for use in situations in which a superficial syntactic analysis of an input may be sufficient.

11.1 Ambiguity

Structural ambiguity

Ambiguity is perhaps the most serious problem faced by syntactic parsers. Chapter 8 introduced the notions of **part-of-speech ambiguity** and **part-of-speech disambiguation**. Here, we introduce a new kind of ambiguity, called **structural ambiguity**, which arises from many commonly used rules in phrase-structure grammars. To illustrate the issues associated with structural ambiguity, we'll make use of a new toy grammar $\mathscr{L}_1$, shown in Figure 11.1, which consists of the $\mathscr{L}_0$ grammar from the last chapter augmented with a few additional rules.

Structural ambiguity occurs when the grammar can assign more than one parse to a sentence. Groucho Marx's well-known line as Captain Spaulding in *Animal*

Grammar	**Lexicon**
$S \rightarrow NP\ VP$	$Det \rightarrow that \mid this \mid the \mid a$
$S \rightarrow Aux\ NP\ VP$	$Noun \rightarrow book \mid flight \mid meal \mid money$
$S \rightarrow VP$	$Verb \rightarrow book \mid include \mid prefer$
$NP \rightarrow Pronoun$	$Pronoun \rightarrow I \mid she \mid me$
$NP \rightarrow Proper\text{-}Noun$	$Proper\text{-}Noun \rightarrow Houston \mid NWA$
$NP \rightarrow Det\ Nominal$	$Aux \rightarrow does$
$Nominal \rightarrow Noun$	$Preposition \rightarrow from \mid to \mid on \mid near \mid through$
$Nominal \rightarrow Nominal\ Noun$	
$Nominal \rightarrow Nominal\ PP$	
$VP \rightarrow Verb$	
$VP \rightarrow Verb\ NP$	
$VP \rightarrow Verb\ NP\ PP$	
$VP \rightarrow Verb\ PP$	
$VP \rightarrow VP\ PP$	
$PP \rightarrow Preposition\ NP$	

Figure 11.1 The $\mathscr{L}_1$ miniature English grammar and lexicon.

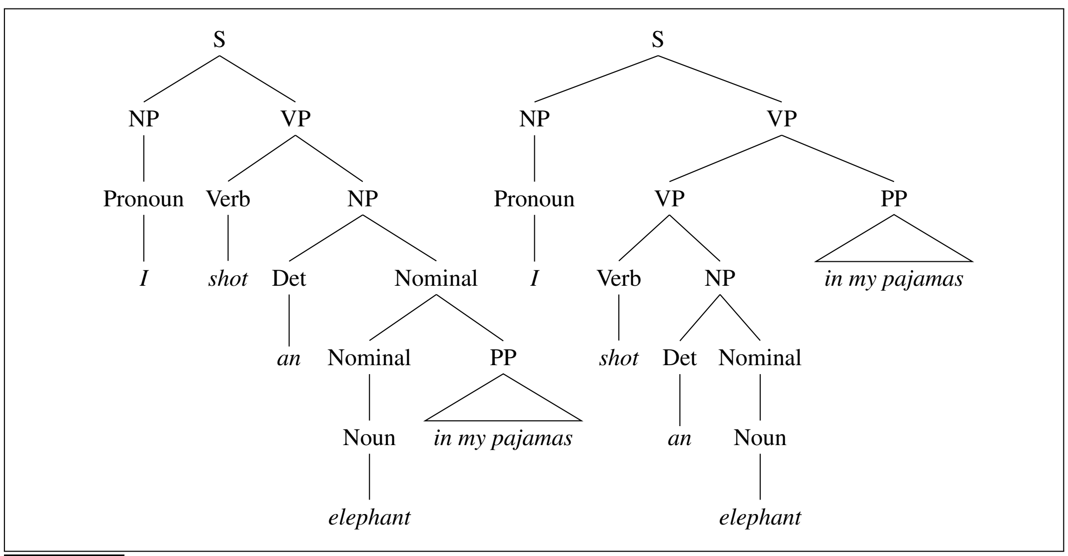


Figure 11.2 Two parse trees for an ambiguous sentence. The parse on the left corresponds to the humorous reading in which the elephant is in the pajamas, the parse on the right corresponds to the reading in which Captain Spaulding did the shooting in his pajamas.

Crackers is ambiguous because the phrase *in my pajamas* can be part of the *NP* headed by *elephant* or a part of the verb phrase headed by *shot*. Figure 11.2 illustrates these two analyses of Marx's line using rules from $\mathscr{L}_1$.

Structural ambiguity, appropriately enough, comes in many forms. Two common kinds of ambiguity are **attachment ambiguity** and **coordination ambiguity**.

Attachment ambiguity

A sentence has an **attachment ambiguity** if a particular constituent can be attached to the parse tree at more than one place. The Groucho Marx sentence is an example of *PP*-attachment ambiguity. Various kinds of adverbial phrases are also subject to this kind of ambiguity. For instance, in the following example the gerundive-*VP flying to Paris* can be part of a gerundive sentence whose subject is *the Eiffel Tower* or it can be an adjunct modifying the *VP* headed by *saw*:

(11.1) We saw the Eiffel Tower flying to Paris.

Coordination ambiguity

In **coordination ambiguity** different sets of phrases can be conjoined by a conjunction like *and*. For example, the phrase *old men and women* can be bracketed as *[old [men and women]]*, referring to *old men* and *old women*, or as *[old men] and [women]*, in which case it is only the men who are old.

These ambiguities combine in complex ways in real sentences. A program that summarized the news, for example, would need to be able to parse sentences like the following from the Brown corpus:

(11.2) President Kennedy today pushed aside other White House business to devote all his time and attention to working on the Berlin crisis address he will deliver tomorrow night to the American people over nationwide television and radio.

This sentence has a number of ambiguities, although since they are semantically unreasonable, it requires a careful reading to see them. The last noun phrase could be parsed *[nationwide [television and radio]]* or *[[nationwide television] and radio]*. The direct object of *pushed aside* should be *other White House business* but could also be the bizarre phrase *[other White House business to devote all his time and attention to working]* (i.e., a structure like *Kennedy affirmed [his intention to propose a new budget to address the deficit]*). Then the phrase *on the Berlin crisis address he will deliver tomorrow night to the American people* could be an adjunct modifying the verb *pushed*. A *PP* like *over nationwide television and radio* could be attached to any of the higher *VP*s or *NP*s (e.g., it could modify *people* or *night*).

The fact that there are many grammatically correct but semantically unreasonable parses for naturally occurring sentences is an irksome problem that affects all parsers. Ultimately, most natural language processing systems need to be able to choose a single correct parse from the multitude of possible parses through a process of **syntactic disambiguation**. Effective disambiguation algorithms require statistical, semantic, and contextual knowledge sources that vary in how well they can be integrated into parsing algorithms.

Syntactic disambiguation

Fortunately, the CKY algorithm presented in the next section is designed to efficiently handle structural ambiguities of the kind we've been discussing. And as we'll see in Chapter 12, there are straightforward ways to integrate statistical techniques into the basic CKY framework to produce highly accurate parsers.

11.2 CKY Parsing: A Dynamic Programming Approach

The previous section introduced some of the problems associated with ambiguous grammars. Fortunately, **dynamic programming** provides a powerful framework for addressing these problems, just as it did with the Minimum Edit Distance, Viterbi, and Forward algorithms. Recall that dynamic programming approaches systematically fill in tables of solutions to sub-problems. When complete, the tables contain the solution to all the sub-problems needed to solve the problem as a whole. In the case of syntactic parsing, these sub-problems represent parse trees for all the constituents detected in the input.

The dynamic programming advantage arises from the context-free nature of our grammar rules — once a constituent has been discovered in a segment of the input we can record its presence and make it available for use in any subsequent derivation that might require it. This provides both time and storage efficiencies since subtrees

can be looked up in a table, not reanalyzed. This section presents the Cocke-Kasami-Younger (CKY) algorithm, the most widely used dynamic-programming based approach to parsing. Related approaches include the **Earley algorithm** (Earley, 1970) and **chart parsing** (Kaplan 1973, Kay 1982).

11.2.1 Conversion to Chomsky Normal Form

We begin our investigation of the CKY algorithm by examining the requirement that grammars used with it must be in Chomsky Normal Form (CNF). Recall from Chapter 10 that grammars in CNF are restricted to rules of the form $A \rightarrow B\ C$ or $A \rightarrow w$. That is, the right-hand side of each rule must expand either to two non-terminals or to a single terminal. Restricting a grammar to CNF does not lead to any loss in expressiveness, since any context-free grammar can be converted into a corresponding CNF grammar that accepts exactly the same set of strings as the original grammar.

Let's start with the process of converting a generic CFG into one represented in CNF. Assuming we're dealing with an ϵ-free grammar, there are three situations we need to address in any generic grammar: rules that mix terminals with non-terminals on the right-hand side, rules that have a single non-terminal on the right-hand side, and rules in which the length of the right-hand side is greater than 2.

The remedy for rules that mix terminals and non-terminals is to simply introduce a new dummy non-terminal that covers only the original terminal. For example, a rule for an infinitive verb phrase such as $\textit{INF-VP} \rightarrow \textit{to VP}$ would be replaced by the two rules $\textit{INF-VP} \rightarrow \textit{TO VP}$ and $\textit{TO} \rightarrow \textit{to}$.

Unit productions

Rules with a single non-terminal on the right are called **unit productions**. We can eliminate unit productions by rewriting the right-hand side of the original rules with the right-hand side of all the non-unit production rules that they ultimately lead to. More formally, if $A \stackrel{*}{\Rightarrow} B$ by a chain of one or more unit productions and $B \rightarrow \gamma$ is a non-unit production in our grammar, then we add $A \rightarrow \gamma$ for each such rule in the grammar and discard all the intervening unit productions. As we demonstrate with our toy grammar, this can lead to a substantial *flattening* of the grammar and a consequent promotion of terminals to fairly high levels in the resulting trees.

Rules with right-hand sides longer than 2 are normalized through the introduction of new non-terminals that spread the longer sequences over several new rules. Formally, if we have a rule like

$$A \rightarrow B\ C\ \gamma$$

we replace the leftmost pair of non-terminals with a new non-terminal and introduce a new production result in the following new rules:

$$\begin{aligned} A &\rightarrow X1\ \gamma \\ X1 &\rightarrow B\ C \end{aligned}$$

In the case of longer right-hand sides, we simply iterate this process until the offending rule has been replaced by rules of length 2. The choice of replacing the leftmost pair of non-terminals is purely arbitrary; any systematic scheme that results in binary rules would suffice.

In our current grammar, the rule $S \rightarrow \textit{Aux NP VP}$ would be replaced by the two rules $S \rightarrow \textit{X1 VP}$ and $\textit{X1} \rightarrow \textit{Aux NP}$.

The entire conversion process can be summarized as follows:

1. Copy all conforming rules to the new grammar unchanged.

$\mathscr{L}_1$ Grammar	$\mathscr{L}_1$ in CNF
$S \rightarrow NP\ VP$	$S \rightarrow NP\ VP$
$S \rightarrow Aux\ NP\ VP$	$S \rightarrow X1\ VP$
	$X1 \rightarrow Aux\ NP$
$S \rightarrow VP$	$S \rightarrow book \mid include \mid prefer$
	$S \rightarrow Verb\ NP$
	$S \rightarrow X2\ PP$
	$S \rightarrow Verb\ PP$
	$S \rightarrow VP\ PP$
$NP \rightarrow Pronoun$	$NP \rightarrow I \mid she \mid me$
$NP \rightarrow Proper\text{-}Noun$	$NP \rightarrow TWA \mid Houston$
$NP \rightarrow Det\ Nominal$	$NP \rightarrow Det\ Nominal$
$Nominal \rightarrow Noun$	$Nominal \rightarrow book \mid flight \mid meal \mid money$
$Nominal \rightarrow Nominal\ Noun$	$Nominal \rightarrow Nominal\ Noun$
$Nominal \rightarrow Nominal\ PP$	$Nominal \rightarrow Nominal\ PP$
$VP \rightarrow Verb$	$VP \rightarrow book \mid include \mid prefer$
$VP \rightarrow Verb\ NP$	$VP \rightarrow Verb\ NP$
$VP \rightarrow Verb\ NP\ PP$	$VP \rightarrow X2\ PP$
	$X2 \rightarrow Verb\ NP$
$VP \rightarrow Verb\ PP$	$VP \rightarrow Verb\ PP$
$VP \rightarrow VP\ PP$	$VP \rightarrow VP\ PP$
$PP \rightarrow Preposition\ NP$	$PP \rightarrow Preposition\ NP$

Figure 11.3 $\mathscr{L}_1$ Grammar and its conversion to CNF. Note that although they aren't shown here, all the original lexical entries from $\mathscr{L}_1$ carry over unchanged as well.

2. Convert terminals within rules to dummy non-terminals.
3. Convert unit-productions.
4. Make all rules binary and add them to new grammar.

Figure 11.3 shows the results of applying this entire conversion procedure to the $\mathscr{L}_1$ grammar introduced earlier on page 202. Note that this figure doesn't show the original lexical rules; since these original lexical rules are already in CNF, they all carry over unchanged to the new grammar. Figure 11.3 does, however, show the various places where the process of eliminating unit productions has, in effect, created new lexical rules. For example, all the original verbs have been promoted to both *VP*s and to *S*s in the converted grammar.

11.2.2 CKY Recognition

With our grammar now in CNF, each non-terminal node above the part-of-speech level in a parse tree will have exactly two daughters. A two-dimensional matrix can be used to encode the structure of an entire tree. For a sentence of length n, we will work with the upper-triangular portion of an $(n+1) \times (n+1)$ matrix. Each cell $[i, j]$ in this matrix contains the set of non-terminals that represent all the constituents that span positions i through j of the input. Since our indexing scheme begins with 0, it's natural to think of the indexes as pointing at the gaps between the input words (as in $_0$ *Book* $_1$ *that* $_2$ *flight* $_3$). It follows then that the cell that represents the entire input resides in position $[0, n]$ in the matrix.

Since each non-terminal entry in our table has two daughters in the parse, it follows that for each constituent represented by an entry $[i, j]$, there must be a position in the input, k, where it can be split into two parts such that $i < k < j$. Given such

a position k, the first constituent $[i,k]$ must lie to the left of entry $[i,j]$ somewhere along row i, and the second entry $[k,j]$ must lie beneath it, along column j.

To make this more concrete, consider the following example with its completed parse matrix, shown in Fig. 11.4.

(11.3) Book the flight through Houston.

The superdiagonal row in the matrix contains the parts of speech for each input word in the input. The subsequent diagonals above that superdiagonal contain constituents that cover all the spans of increasing length in the input.

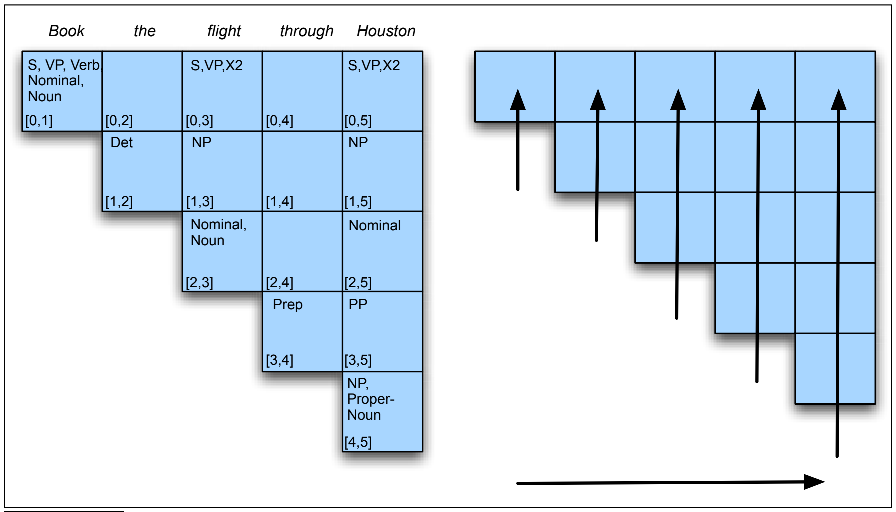


Figure 11.4 Completed parse table for *Book the flight through Houston.*

Given this setup, CKY recognition consists of filling the parse table in the right way. To do this, we'll proceed in a bottom-up fashion so that at the point where we are filling any cell $[i,j]$, the cells containing the parts that could contribute to this entry (i.e., the cells to the left and the cells below) have already been filled. The algorithm given in Fig. 11.5 fills the upper-triangular matrix a column at a time working from left to right, with each column filled from bottom to top, as the right side of Fig. 11.4 illustrates. This scheme guarantees that at each point in time we have all the information we need (to the left, since all the columns to the left have already been filled, and below since we're filling bottom to top). It also mirrors on-line parsing since filling the columns from left to right corresponds to processing each word one at a time.

function CKY-PARSE(*words, grammar*) **returns** *table*

  for $j \leftarrow$ **from** 1 **to** LENGTH(*words*) **do**
    for all $\{A \mid A \rightarrow words[j] \in grammar\}$
      $table[j-1,j] \leftarrow table[j-1,j] \cup A$
    for $i \leftarrow$ **from** $j-2$ **downto** 0 **do**
      for $k \leftarrow i+1$ **to** $j-1$ **do**
        for all $\{A \mid A \rightarrow BC \in grammar$ **and** $B \in table[i,k]$ **and** $C \in table[k,j]\}$
          $table[i,j] \leftarrow table[i,j] \cup A$

Figure 11.5 The CKY algorithm.

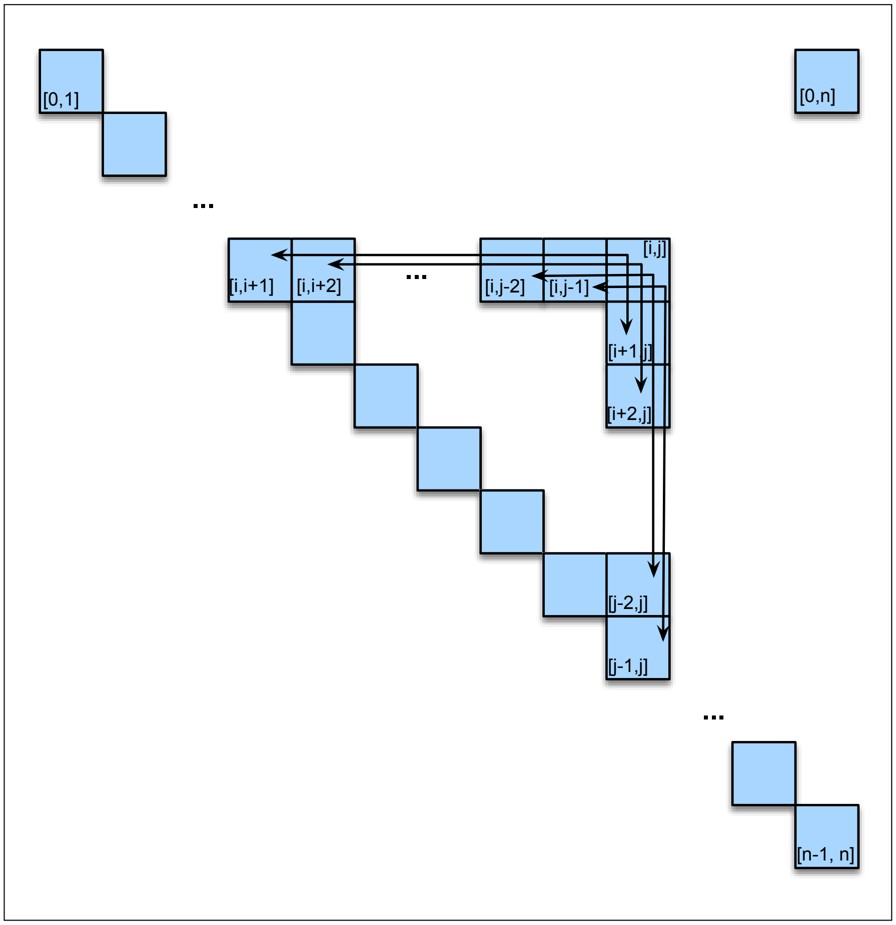


Figure 11.6 All the ways to fill the $[i, j]$th cell in the CKY table.

The outermost loop of the algorithm given in Fig. 11.5 iterates over the columns, and the second loop iterates over the rows, from the bottom up. The purpose of the innermost loop is to range over all the places where a substring spanning i to j in the input might be split in two. As k ranges over the places where the string can be split, the pairs of cells we consider move, in lockstep, to the right along row i and down along column j. Figure 11.6 illustrates the general case of filling cell $[i, j]$. At each such split, the algorithm considers whether the contents of the two cells can be combined in a way that is sanctioned by a rule in the grammar. If such a rule exists, the non-terminal on its left-hand side is entered into the table.

Figure 11.7 shows how the five cells of column 5 of the table are filled after the word *Houston* is read. The arrows point out the two spans that are being used to add an entry to the table. Note that the action in cell $[0,5]$ indicates the presence of three alternative parses for this input, one where the *PP* modifies the *flight*, one where it modifies the booking, and one that captures the second argument in the original *VP* $\rightarrow$ *Verb NP PP* rule, now captured indirectly with the *VP* $\rightarrow$ *X2 PP* rule.

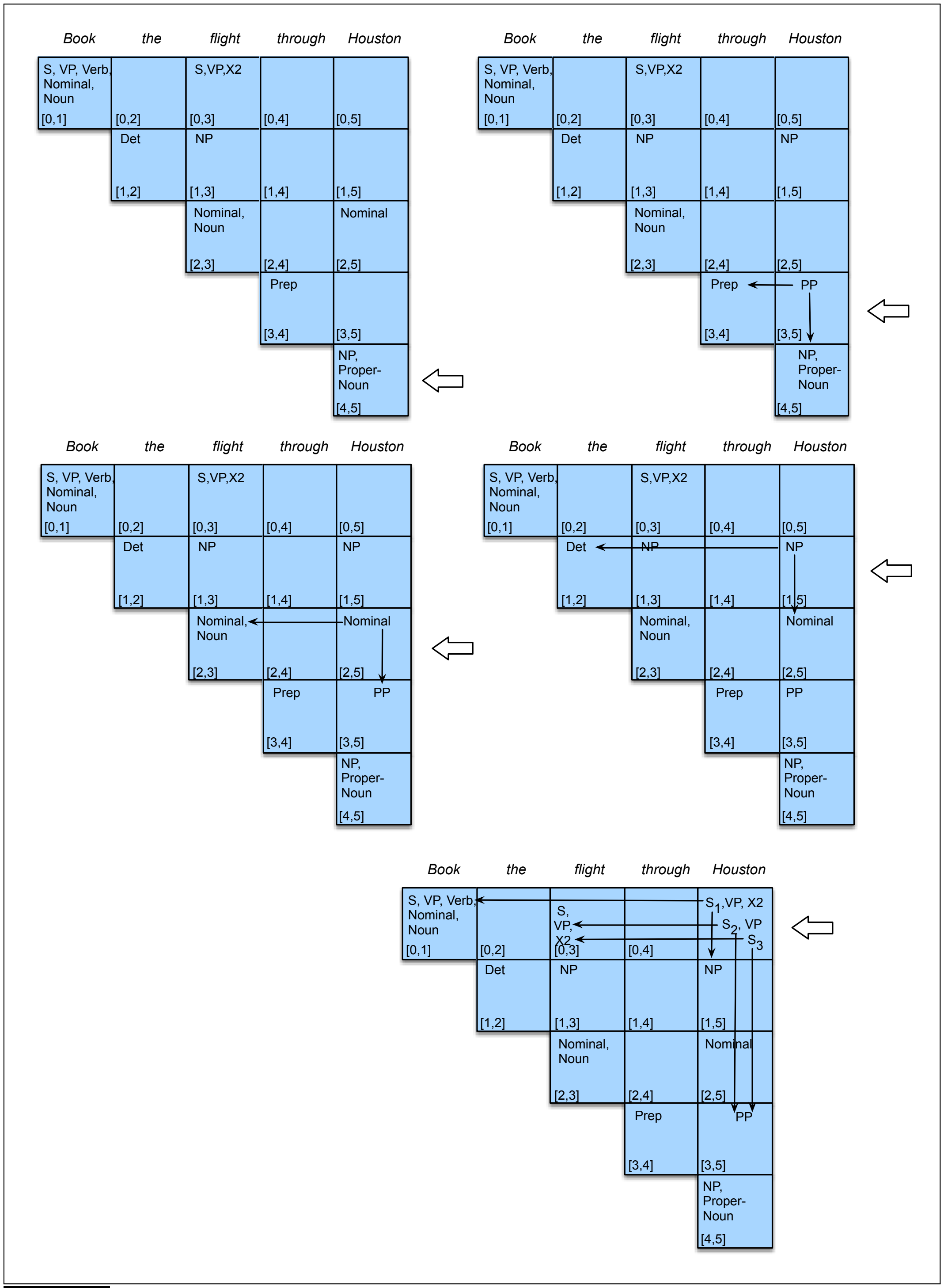


Figure 11.7 Filling the cells of column 5 after reading the word *Houston*.

11.2.3 CKY Parsing

The algorithm given in Fig. 11.5 is a recognizer, not a parser; for it to succeed, it simply has to find an S in cell $[0,n]$. To turn it into a parser capable of returning all possible parses for a given input, we can make two simple changes to the algorithm: the first change is to augment the entries in the table so that each non-terminal is paired with pointers to the table entries from which it was derived (more or less as shown in Fig. 11.7), the second change is to permit multiple versions of the same non-terminal to be entered into the table (again as shown in Fig. 11.7). With these changes, the completed table contains all the possible parses for a given input. Returning an arbitrary single parse consists of choosing an S from cell $[0,n]$ and then recursively retrieving its component constituents from the table.

Of course, returning all the parses for a given input may incur considerable cost since an exponential number of parses may be associated with a given input. In such cases, returning all the parses will have an unavoidable exponential cost. Looking forward to Chapter 12, we can also think about retrieving the best parse for a given input by further augmenting the table to contain the probabilities of each entry. Retrieving the most probable parse consists of running a suitably modified version of the Viterbi algorithm from Chapter 8 over the completed parse table.

11.2.4 CKY in Practice

Finally, we should note that while the restriction to CNF does not pose a problem theoretically, it does pose some non-trivial problems in practice. Obviously, as things stand now, our parser isn't returning trees that are consistent with the grammar given to us by our friendly syntacticians. In addition to making our grammar developers unhappy, the conversion to CNF will complicate any syntax-driven approach to semantic analysis.

One approach to getting around these problems is to keep enough information around to transform our trees back to the original grammar as a post-processing step of the parse. This is trivial in the case of the transformation used for rules with length greater than 2. Simply deleting the new dummy non-terminals and promoting their daughters restores the original tree.

In the case of unit productions, it turns out to be more convenient to alter the basic CKY algorithm to handle them directly than it is to store the information needed to recover the correct trees. Exercise 11.3 asks you to make this change. Many of the probabilistic parsers presented in Chapter 12 use the CKY algorithm altered in just this manner. Another solution is to adopt a more complex dynamic programming solution that simply accepts arbitrary CFGs. The next section presents such an approach.

11.3 Partial Parsing

partial parse
shallow parse

Many language processing tasks do not require complex, complete parse trees for all inputs. For these tasks, a **partial parse**, or **shallow parse**, of input sentences may be sufficient. For example, information extraction systems generally do not extract *all* the possible information from a text: they simply identify and classify the segments in a text that are likely to contain valuable information. Similarly, information retrieval systems may index texts according to a subset of the constituents found in

them.

There are many different approaches to partial parsing. Some make use of cascades of finite state transducers to produce tree-like representations. These approaches typically produce flatter trees than the ones we've been discussing in this chapter and the previous one. This flatness arises from the fact that finite state transducer approaches generally defer decisions that may require semantic or contextual factors, such as prepositional phrase attachments, coordination ambiguities, and nominal compound analyses. Nevertheless, the intent is to produce parse trees that link all the major constituents in an input.

chunking

An alternative style of partial parsing is known as **chunking**. Chunking is the process of identifying and classifying the flat, non-overlapping segments of a sentence that constitute the basic non-recursive phrases corresponding to the major content-word parts-of-speech: noun phrases, verb phrases, adjective phrases, and prepositional phrases. THe task of finding all the base noun phrases in a text is particularly common. Since chunked texts lack a hierarchical structure, a simple bracketing notation is sufficient to denote the location and the type of the chunks in a given example:

(11.4) [$_{NP}$ The morning flight] [$_{PP}$ from] [$_{NP}$ Denver] [$_{VP}$ has arrived.]

This bracketing notation makes clear the two fundamental tasks that are involved in chunking: segmenting (finding the non-overlapping extents of the chunks) and labeling (assigning the correct tag to the discovered chunks).

Some input words may not be part of any chunk, particularly in tasks like base *NP*:

(11.5) [$_{NP}$ The morning flight] from [$_{NP}$ Denver] has arrived.

What constitutes a syntactic base phrase depends on the application (and whether the phrases come from a treebank). Nevertheless, some standard guidelines are followed in most systems. First and foremost, base phrases of a given type do not recursively contain any constituents of the same type. Eliminating this kind of recursion leaves us with the problem of determining the boundaries of the non-recursive phrases. In most approaches, base phrases include the headword of the phrase, along with any pre-head material within the constituent, while crucially excluding any post-head material. Eliminating post-head modifiers obviates the need to resolve attachment ambiguities. This exclusion does lead to certain oddities, such as *PP*s and *VP*s often consisting solely of their heads. Thus, our earlier example *a flight from Indianapolis to Houston on NWA* is reduced to the following:

(11.6) [$_{NP}$ a flight] [$_{PP}$ from] [$_{NP}$ Indianapolis][$_{PP}$ to][$_{NP}$ Houston][$_{PP}$ on][$_{NP}$ NWA]

11.3.1 Machine Learning-Based Approaches to Chunking

IOB

State-of-the-art approaches to chunking use supervised machine learning to train a chunker by using annotated data as a training set and training any sequence labeler. It's common to model chunking as **IOB** tagging. In IOB tagging we introduce a tag for the beginning (B) and inside (I) of each chunk type, and one for tokens outside (O) any chunk. The number of tags is thus $2n+1$ tags, where n is the number of chunk types. IOB tagging can represent exactly the same information as the bracketed notation. The following example shows the bracketing notation of (11.4) on page 210 reframed as a tagging task:

(11.7)
The	*morning*	*flight*	*from*	*Denver*	*has*	*arrived*
B_NP	I_NP	I_NP	B_PP	B_NP	B_VP	I_VP

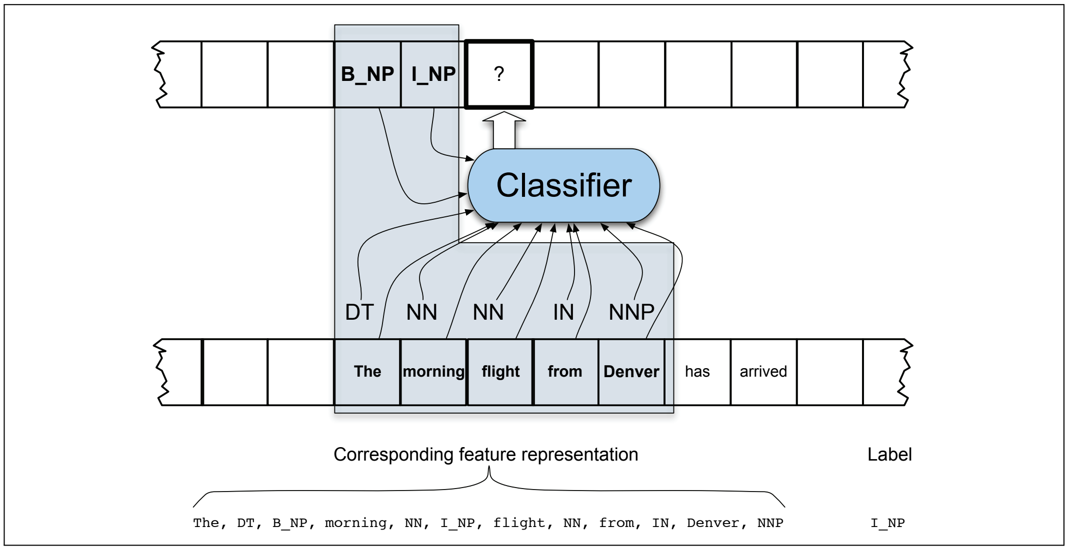


Figure 11.8 A sequence model for chunking. The chunker slides a context window over the sentence, classifying words as it proceeds. At this point, the classifier is attempting to label *flight*, using features like words, embeddings, part-of-speech tags and previously assigned chunk tags.

The same sentence with only the base-NPs tagged illustrates the role of the O tags.

(11.8) *The morning flight from Denver has arrived.*
B_NP I_NP I_NP O B_NP O O

There is no explicit encoding of the end of a chunk in IOB tagging; the end of any chunk is implicit in any transition from an I or B to a B or O tag. This encoding reflects the notion that when sequentially labeling words, it is generally easier (at least in English) to detect the beginning of a new chunk than it is to know when a chunk has ended.

Since annotation efforts are expensive and time consuming, chunkers usually rely on existing treebanks like the Penn Treebank (Chapter 10), extracting syntactic phrases from the full parse constituents of a sentence, finding the appropriate heads and then including the material to the left of the head, ignoring the text to the right. This is somewhat error-prone since it relies on the accuracy of the head-finding rules described in Chapter 10.

Given a training set, any sequence model can be used. Figure 11.8 shows an illustration of a simple feature-based model, using features like the words and parts-of-speech within a 2 word window, and the chunk tags of the preceding inputs in the window. In training, each training vector would consist of the values of 13 features; the two words to the left of the decision point, their parts-of-speech and chunk tags, the word to be tagged along with its part-of-speech, the two words that follow along with their parts-of speech, and the correct chunk tag, in this case, I_NP. During classification, the classifier is given the same vector without the answer and assigns the most appropriate tag from its tagset. Viterbi decoding is commonly used.

11.3.2 Chunking-System Evaluations

As with the evaluation of part-of-speech taggers, the evaluation of chunkers proceeds by comparing chunker output with gold-standard answers provided by human annotators. However, unlike part-of-speech tagging, word-by-word accuracy measures are not appropriate. Instead, chunkers are evaluated according to the notions of precision, recall, and the F-measure borrowed from the field of information retrieval.

precision

Precision measures the percentage of system-provided chunks that were correct. Correct here means that both the boundaries of the chunk and the chunk's label are correct. Precision is therefore defined as

$$\textbf{Precision:} = \frac{\text{Number of correct chunks given by system}}{\text{Total number of chunks given by system}}$$

recall

Recall measures the percentage of chunks actually present in the input that were correctly identified by the system. Recall is defined as

$$\textbf{Recall:} = \frac{\text{Number of correct chunks given by system}}{\text{Total number of actual chunks in the text}}$$

F-measure

The ***F*-measure** (van Rijsbergen, 1975) provides a way to combine these two measures into a single metric. The F-measure is defined as

$$F_\beta = \frac{(\beta^2+1)PR}{\beta^2 P + R}$$

The β parameter differentially weights the importance of recall and precision, based perhaps on the needs of an application. Values of $\beta > 1$ favor recall, while values of $\beta < 1$ favor precision. When $\beta = 1$, precision and recall are equally balanced; this is sometimes called $F_{\beta=1}$ or just F_1:

$$F_1 = \frac{2PR}{P+R} \tag{11.9}$$

F-measure comes from a weighted harmonic mean of precision and recall. The harmonic mean of a set of numbers is the reciprocal of the arithmetic mean of reciprocals:

$$\text{HarmonicMean}(\mathrm{a}_1, \mathrm{a}_2, \mathrm{a}_3, \mathrm{a}_4, \ldots, \mathrm{a}_\mathrm{n}) = \frac{\mathrm{n}}{\frac{1}{\mathrm{a}_1} + \frac{1}{\mathrm{a}_2} + \frac{1}{\mathrm{a}_3} + \ldots + \frac{1}{\mathrm{a}_\mathrm{n}}} \tag{11.10}$$

and hence F-measure is

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} \quad \text{or} \left(\text{with } \beta^2 = \frac{1-\alpha}{\alpha}\right) \quad F = \frac{(\beta^2+1)PR}{\beta^2 P + R} \tag{11.11}$$

11.4 Summary

The two major ideas introduced in this chapter are those of **parsing** and **partial parsing**. Here's a summary of the main points we covered about these ideas:

- **Structural ambiguity** is a significant problem for parsers. Common sources of structural ambiguity include **PP-attachment**, **coordination ambiguity**, and **noun-phrase bracketing ambiguity**.
- **Dynamic programming** parsing algorithms, such as **CKY**, use a table of partial parses to efficiently parse ambiguous sentences.

- **CKY** restricts the form of the grammar to Chomsky normal form (CNF).
- Many practical problems, including **information extraction** problems, can be solved without full parsing.
- **Partial parsing** and **chunking** are methods for identifying shallow syntactic constituents in a text.
- State-of-the-art methods for partial parsing use **supervised machine learning** techniques.

Bibliographical and Historical Notes

Writing about the history of compilers, Knuth notes:

> In this field there has been an unusual amount of parallel discovery of the same technique by people working independently.

Well, perhaps not unusual, since multiple discovery is the norm in science (see page **??**). But there has certainly been enough parallel publication that this history errs on the side of succinctness in giving only a characteristic early mention of each algorithm; the interested reader should see Aho and Ullman (1972).

Bottom-up parsing seems to have been first described by Yngve (1955), who gave a breadth-first, bottom-up parsing algorithm as part of an illustration of a machine translation procedure. Top-down approaches to parsing and translation were described (presumably independently) by at least Glennie (1960), Irons (1961), and Kuno and Oettinger (1963). Dynamic programming parsing, once again, has a history of independent discovery. According to Martin Kay (personal communication), a dynamic programming parser containing the roots of the CKY algorithm was first implemented by John Cocke in 1960. Later work extended and formalized the algorithm, as well as proving its time complexity (Kay 1967,Younger 1967,Kasami 1965).
WFST The related **well-formed substring table** (**WFST**) seems to have been independently proposed by Kuno (1965) as a data structure that stores the results of all previous computations in the course of the parse. Based on a generalization of Cocke's work, a similar data structure had been independently described in Kay 1967, Kay 1973. The top-down application of dynamic programming to parsing was described in Earley's Ph.D. dissertation (Earley 1968, Earley 1970). Sheil (1976) showed the equivalence of the WFST and the Earley algorithm. Norvig (1991) shows that the efficiency offered by dynamic programming can be captured in any language with a *memoization* function (such as in LISP) simply by wrapping the *memoization* operation around a simple top-down parser.

While parsing via cascades of finite-state automata had been common in the early history of parsing (Harris, 1962), the focus shifted to full CFG parsing quite soon afterward. Church (1980) argued for a return to finite-state grammars as a processing model for natural language understanding; other early finite-state parsing models include Ejerhed (1988). Abney (1991) argued for the important practical role of shallow parsing.

The classic reference for parsing algorithms is Aho and Ullman (1972); although the focus of that book is on computer languages, most of the algorithms have been applied to natural language. A good programming languages textbook such as Aho et al. (1986) is also useful.

Exercises

11.1 Implement the algorithm to convert arbitrary context-free grammars to CNF. Apply your program to the $\mathscr{L}_1$ grammar.

11.2 Implement the CKY algorithm and test it with your converted $\mathscr{L}_1$ grammar.

11.3 Rewrite the CKY algorithm given in Fig. 11.5 on page 206 so that it can accept grammars that contain unit productions.

11.4 Discuss the relative advantages and disadvantages of partial versus full parsing.

11.5 Discuss how to augment a parser to deal with input that may be incorrect, for example, containing spelling errors or mistakes arising from automatic speech recognition.

CHAPTER

12 Statistical Parsing

The characters in Damon Runyon's short stories are willing to bet "on any proposition whatever", as Runyon says about Sky Masterson in *The Idyll of Miss Sarah Brown*, from the probability of getting aces back-to-back to the odds against a man being able to throw a peanut from second base to home plate. There is a moral here for language processing: with enough knowledge we can figure the probability of just about anything. The last two chapters have introduced sophisticated models of syntactic structure and its parsing. Here, we show that it is possible to build probabilistic models of syntactic knowledge and use some of this probabilistic knowledge to build efficient probabilistic parsers.

One crucial use of probabilistic parsing is to solve the problem of **disambiguation**. Recall from Chapter 11 that sentences on average tend to be syntactically ambiguous because of phenomena like **coordination ambiguity** and **attachment ambiguity**. The CKY parsing algorithm can represent these ambiguities in an efficient way but is not equipped to resolve them. A probabilistic parser offers a solution to the problem: compute the probability of each interpretation and choose the most probable interpretation. Thus, due to the prevalence of ambiguity, most modern parsers used for natural language understanding tasks (semantic analysis, summarization, question-answering, machine translation) are of necessity probabilistic.

The most commonly used probabilistic grammar formalism is the **probabilistic context-free grammar** (PCFG), a probabilistic augmentation of context-free grammars in which each rule is associated with a probability. We introduce PCFGs in the next section, showing how they can be trained on Treebank grammars and how they can be parsed. We present the most basic parsing algorithm for PCFGs, which is the probabilistic version of the **CKY algorithm** that we saw in Chapter 11.

We then show a number of ways that we can improve on this basic probability model (PCFGs trained on Treebank grammars). One method of improving a trained Treebank grammar is to change the names of the non-terminals. By making the non-terminals sometimes more specific and sometimes more general, we can come up with a grammar with a better probability model that leads to improved parsing scores. Another augmentation of the PCFG works by adding more sophisticated conditioning factors, extending PCFGs to handle probabilistic **subcategorization** information and probabilistic **lexical dependencies**.

Heavily lexicalized grammar formalisms such as Lexical-Functional Grammar (LFG) (Bresnan, 1982), Head-Driven Phrase Structure Grammar (HPSG) (Pollard and Sag, 1994), Tree-Adjoining Grammar (TAG) (Joshi, 1985), and Combinatory Categorial Grammar (CCG) pose additional problems for probabilistic parsers. Section 12.7 introduces the task of **supertagging** and the use of heuristic search methods based on the **A* algorithm** in the context of CCG parsing.

Finally, we describe the standard techniques and metrics for evaluating parsers and discuss some relevant psychological results on human parsing.

12.1 Probabilistic Context-Free Grammars

PCFG
SCFG

The simplest augmentation of the context-free grammar is the **Probabilistic Context-Free Grammar** (**PCFG**), also known as the **Stochastic Context-Free Grammar** (**SCFG**), first proposed by Booth (1969). Recall that a context-free grammar G is defined by four parameters (N, Σ, R, S); a probabilistic context-free grammar is also defined by four parameters, with a slight augmentation to each of the rules in R:

N	a set of **non-terminal symbols** (or **variables**)
Σ	a set of **terminal symbols** (disjoint from N)
R	a set of **rules** or productions, each of the form $A \rightarrow \beta\ [p]$, where A is a non-terminal, β is a string of symbols from the infinite set of strings $(\Sigma \cup N)*$, and p is a number between 0 and 1 expressing $P(\beta\|A)$
S	a designated **start symbol**

That is, a PCFG differs from a standard CFG by augmenting each rule in R with a conditional probability:

$$A \rightarrow \beta \quad [p] \tag{12.1}$$

Here p expresses the probability that the given non-terminal A will be expanded to the sequence β. That is, p is the conditional probability of a given expansion β given the left-hand-side (LHS) non-terminal A. We can represent this probability as

$$P(A \rightarrow \beta)$$

or as

$$P(A \rightarrow \beta|A)$$

or as

$$P(RHS|LHS)$$

Thus, if we consider all the possible expansions of a non-terminal, the sum of their probabilities must be 1:

$$\sum_{\beta} P(A \rightarrow \beta) = 1$$

Figure 12.1 shows a PCFG: a probabilistic augmentation of the $\mathscr{L}_1$ miniature English CFG grammar and lexicon. Note that the probabilities of all of the expansions of each non-terminal sum to 1. Also note that these probabilities were made up for pedagogical purposes. A real grammar has a great many more rules for each non-terminal; hence, the probabilities of any particular rule would tend to be much smaller.

consistent

A PCFG is said to be **consistent** if the sum of the probabilities of all sentences in the language equals 1. Certain kinds of recursive rules cause a grammar to be inconsistent by causing infinitely looping derivations for some sentences. For example, a rule $S \rightarrow S$ with probability 1 would lead to lost probability mass due to derivations that never terminate. See Booth and Thompson (1973) for more details on consistent and inconsistent grammars.

Grammar		Lexicon
$S \rightarrow NP\ VP$	[.80]	$Det \rightarrow that$ [.10] | a [.30] | the [.60]
$S \rightarrow Aux\ NP\ VP$	[.15]	$Noun \rightarrow book$ [.10] | $flight$ [.30]
$S \rightarrow VP$	[.05]	| $meal$ [.015] | $money$ [.05]
$NP \rightarrow Pronoun$	[.35]	| $flight$ [.40] | $dinner$ [.10]
$NP \rightarrow Proper\text{-}Noun$	[.30]	$Verb \rightarrow book$ [.30] | $include$ [.30]
$NP \rightarrow Det\ Nominal$	[.20]	| $prefer$ [.40]
$NP \rightarrow Nominal$	[.15]	$Pronoun \rightarrow I$ [.40] | she [.05]
$Nominal \rightarrow Noun$	[.75]	| me [.15] | you [.40]
$Nominal \rightarrow Nominal\ Noun$	[.20]	$Proper\text{-}Noun \rightarrow Houston$ [.60]
$Nominal \rightarrow Nominal\ PP$	[.05]	| NWA [.40]
$VP \rightarrow Verb$	[.35]	$Aux \rightarrow does$ [.60] | can [40]
$VP \rightarrow Verb\ NP$	[.20]	$Preposition \rightarrow from$ [.30] | to [.30]
$VP \rightarrow Verb\ NP\ PP$	[.10]	| on [.20] | $near$ [.15]
$VP \rightarrow Verb\ PP$	[.15]	| $through$ [.05]
$VP \rightarrow Verb\ NP\ NP$	[.05]	
$VP \rightarrow VP\ PP$	[.15]	
$PP \rightarrow Preposition\ NP$	[1.0]	

Figure 12.1 A PCFG that is a probabilistic augmentation of the $\mathscr{L}_1$ miniature English CFG grammar and lexicon of Fig. 11.1. These probabilities were made up for pedagogical purposes and are not based on a corpus (since any real corpus would have many more rules, so the true probabilities of each rule would be much smaller).

How are PCFGs used? A PCFG can be used to estimate a number of useful probabilities concerning a sentence and its parse tree(s), including the probability of a particular parse tree (useful in disambiguation) and the probability of a sentence or a piece of a sentence (useful in language modeling). Let's see how this works.

12.1.1 PCFGs for Disambiguation

A PCFG assigns a probability to each parse tree T (i.e., each **derivation**) of a sentence S. This attribute is useful in **disambiguation**. For example, consider the two parses of the sentence "Book the dinner flight" shown in Fig. 12.2. The sensible parse on the left means "Book a flight that serves dinner". The nonsensical parse on the right, however, would have to mean something like "Book a flight on behalf of 'the dinner"' just as a structurally similar sentence like "Can you book John a flight?" means something like "Can you book a flight on behalf of John?"

The probability of a particular parse T is defined as the product of the probabilities of all the n rules used to expand each of the n non-terminal nodes in the parse tree T, where each rule i can be expressed as $LHS_i \rightarrow RHS_i$:

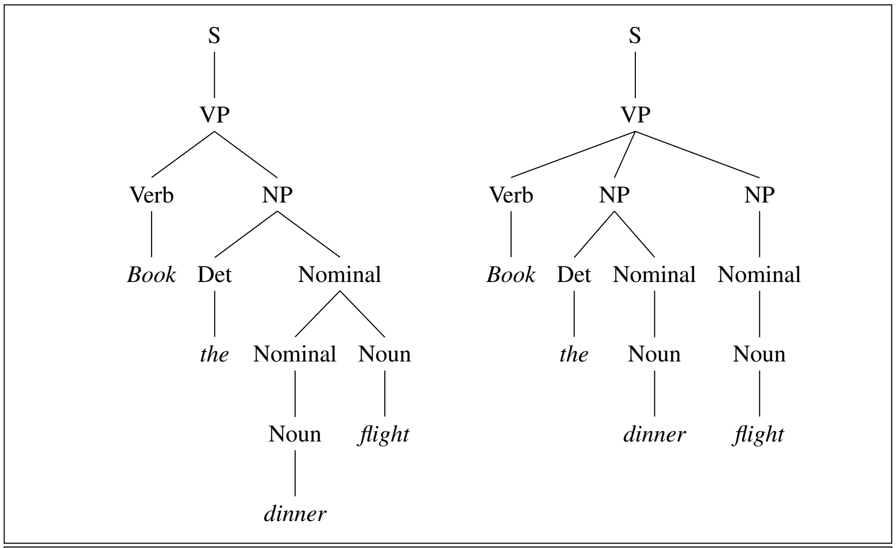
$$P(T,S) = \prod_{i=1}^{n} P(RHS_i|LHS_i) \tag{12.2}$$

The resulting probability $P(T,S)$ is both the joint probability of the parse and the sentence and also the probability of the parse $P(T)$. How can this be true? First, by the definition of joint probability:

$$P(T,S) = P(T)P(S|T) \tag{12.3}$$

But since a parse tree includes all the words of the sentence, $P(S|T)$ is 1. Thus,

$$P(T,S) = P(T)P(S|T) = P(T) \qquad (12.4)$$



Rules			P	Rules			P
S	→	VP	.05	S	→	VP	.05
VP	→	Verb NP	.20	VP	→	Verb NP NP	.10
NP	→	Det Nominal	.20	NP	→	Det Nominal	.20
Nominal	→	Nominal Noun	.20	NP	→	Nominal	.15
Nominal	→	Noun	.75	Nominal	→	Noun	.75
				Nominal	→	Noun	.75
Verb	→	book	.30	Verb	→	book	.30
Det	→	the	.60	Det	→	the	.60
Noun	→	dinner	.10	Noun	→	dinner	.10
Noun	→	flight	.40	Noun	→	flight	.40

Figure 12.2 Two parse trees for an ambiguous sentence. The parse on the left corresponds to the sensible meaning "Book a flight that serves dinner", while the parse on the right corresponds to the nonsensical meaning "Book a flight on behalf of 'the dinner' ".

We can compute the probability of each of the trees in Fig. 12.2 by multiplying the probabilities of each of the rules used in the derivation. For example, the probability of the left tree in Fig. 12.2a (call it T_{left}) and the right tree (Fig. 12.2b or T_{right}) can be computed as follows:

$$\begin{aligned} P(T_{left}) &= .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}} \\ P(T_{right}) &= .05 * .10 * .20 * .15 * .75 * .75 * .30 * .60 * .10 * .40 = \mathbf{6.1 \times 10^{-7}} \end{aligned}$$

We can see that the left tree in Fig. 12.2 has a much higher probability than the tree on the right. Thus, this parse would correctly be chosen by a disambiguation algorithm that selects the parse with the highest PCFG probability.

Let's formalize this intuition that picking the parse with the highest probability is the correct way to do disambiguation. Consider all the possible parse trees for a given sentence S. The string of words S is called the **yield** of any parse tree over S.

yield

Thus, out of all parse trees with a yield of S, the disambiguation algorithm picks the parse tree that is most probable given S:

$$\hat{T}(S) = \underset{T s.t. S=\text{yield}(T)}{\operatorname{argmax}} P(T|S) \tag{12.5}$$

By definition, the probability $P(T|S)$ can be rewritten as $P(T,S)/P(S)$, thus leading to

$$\hat{T}(S) = \underset{T s.t. S=\text{yield}(T)}{\operatorname{argmax}} \frac{P(T,S)}{P(S)} \tag{12.6}$$

Since we are maximizing over all parse trees for the same sentence, $P(S)$ will be a constant for each tree, so we can eliminate it:

$$\hat{T}(S) = \underset{T s.t. S=\text{yield}(T)}{\operatorname{argmax}} P(T,S) \tag{12.7}$$

Furthermore, since we showed above that $P(T,S) = P(T)$, the final equation for choosing the most likely parse neatly simplifies to choosing the parse with the highest probability:

$$\hat{T}(S) = \underset{T s.t. S=\text{yield}(T)}{\operatorname{argmax}} P(T) \tag{12.8}$$

12.1.2 PCFGs for Language Modeling

A second attribute of a PCFG is that it assigns a probability to the string of words constituting a sentence. This is important in **language modeling**, whether for use in speech recognition, machine translation, spelling correction, augmentative communication, or other applications. The probability of an unambiguous sentence is $P(T,S) = P(T)$ or just the probability of the single parse tree for that sentence. The probability of an ambiguous sentence is the sum of the probabilities of all the parse trees for the sentence:

$$\begin{aligned} P(S) &= \sum_{T s.t. S=\text{yield}(T)} P(T,S) && (12.9)\\ &= \sum_{T s.t. S=\text{yield}(T)} P(T) && (12.10) \end{aligned}$$

An additional feature of PCFGs that is useful for language modeling is their ability to assign a probability to substrings of a sentence. For example, suppose we want to know the probability of the next word w_i in a sentence given all the words we've seen so far $w_1,...,w_{i-1}$. The general formula for this is

$$P(w_i|w_1,w_2,...,w_{i-1}) = \frac{P(w_1,w_2,...,w_{i-1},w_i)}{P(w_1,w_2,...,w_{i-1})} \tag{12.11}$$

We saw in Chapter 3 a simple approximation of this probability using N-grams, conditioning on only the last word or two instead of the entire context; thus, the **bigram approximation** would give us

$$P(w_i|w_1,w_2,...,w_{i-1}) \approx \frac{P(w_{i-1},w_i)}{P(w_{i-1})} \tag{12.12}$$

But the fact that the N-gram model can only make use of a couple words of context means it is ignoring potentially useful prediction cues. Consider predicting the word *after* in the following sentence from Chelba and Jelinek (2000):

(12.13) the contract ended with a loss of 7 cents after trading as low as 9 cents

A trigram grammar must predict *after* from the words *7 cents*, while it seems clear that the verb *ended* and the subject *contract* would be useful predictors that a PCFG-based parser could help us make use of. Indeed, it turns out that PCFGs allow us to condition on the entire previous context $w_1, w_2, ..., w_{i-1}$ shown in Eq. 12.11.

In summary, this section and the previous one have shown that PCFGs can be applied both to disambiguation in syntactic parsing and to word prediction in language modeling. Both of these applications require that we be able to compute the probability of parse tree T for a given sentence S. The next few sections introduce some algorithms for computing this probability.

12.2 Probabilistic CKY Parsing of PCFGs

The parsing problem for PCFGs is to produce the most-likely parse $\hat{T}$ for a given sentence S, that is,

$$\hat{T}(S) = \underset{T s.t. S=\text{yield}(T)}{\text{argmax}} P(T) \tag{12.14}$$

The algorithms for computing the most likely parse are simple extensions of the standard algorithms for parsing; most modern probabilistic parsers are based on the **probabilistic CKY** algorithm, first described by Ney (1991).

probabilistic CKY

As with the CKY algorithm, we assume for the probabilistic CKY algorithm that the PCFG is in Chomsky normal form. Recall from page 191 that grammars in CNF are restricted to rules of the form $A \rightarrow B\ C$, or $A \rightarrow w$. That is, the right-hand side of each rule must expand to either two non-terminals or to a single terminal.

For the CKY algorithm, we represented each sentence as having indices between the words. Thus, an example sentence like

(12.15) Book the flight through Houston.

would assume the following indices between each word:

(12.16) ⓪ Book ① the ② flight ③ through ④ Houston ⑤

Using these indices, each constituent in the CKY parse tree is encoded in a two-dimensional matrix. Specifically, for a sentence of length n and a grammar that contains V non-terminals, we use the upper-triangular portion of an $(n+1) \times (n+1)$ matrix. For CKY, each cell $table[i, j]$ contained a list of constituents that could span the sequence of words from i to j. For probabilistic CKY, it's slightly simpler to think of the constituents in each cell as constituting a third dimension of maximum length V. This third dimension corresponds to each non-terminal that can be placed in this cell, and the value of the cell is then a probability for that non-terminal/constituent rather than a list of constituents. In summary, each cell $[i, j, A]$ in this $(n+1) \times (n+1) \times V$ matrix is the probability of a constituent of type A that spans positions i through j of the input.

Figure 12.3 gives pseudocode for this probabilistic CKY algorithm, extending the basic CKY algorithm from Fig. 11.5.

function PROBABILISTIC-CKY(*words,grammar*) **returns** most probable parse
and its probability
for $j \leftarrow$ **from** 1 **to** LENGTH(*words*) **do**
for all $\{ A \mid A \rightarrow words[j] \in grammar\}$
$table[j-1, j, A] \leftarrow P(A \rightarrow words[j])$
for $i \leftarrow$ **from** $j-2$ **downto** 0 **do**
for $k \leftarrow i+1$ **to** $j-1$ **do**
for all $\{ A \mid A \rightarrow BC \in grammar,$
and $table[i,k,B] > 0$ **and** $table[k,j,C] > 0 \}$
if $(table[i,j,A] < P(A \rightarrow BC) \times table[i,k,B] \times table[k,j,C])$ **then**
$table[i,j,A] \leftarrow P(A \rightarrow BC) \times table[i,k,B] \times table[k,j,C]$
$back[i,j,A] \leftarrow \{k, B, C\}$
return BUILD_TREE(*back*[1, LENGTH(*words*), *S*]), *table*[1, LENGTH(*words*), *S*]

Figure 12.3 The probabilistic CKY algorithm for finding the maximum probability parse of a string of *num_words* words given a PCFG grammar with *num_rules* rules in Chomsky normal form. *back* is an array of backpointers used to recover the best parse. The *build_tree* function is left as an exercise to the reader.

Like the basic CKY algorithm, the probabilistic CKY algorithm as shown in Fig. 12.3 requires a grammar in Chomsky normal form. Converting a probabilistic grammar to CNF requires that we also modify the probabilities so that the probability of each parse remains the same under the new CNF grammar. Exercise 12.2 asks you to modify the algorithm for conversion to CNF in Chapter 11 so that it correctly handles rule probabilities.

In practice, a generalized CKY algorithm that handles unit productions directly is typically used. Recall that Exercise 13.3 asked you to make this change in CKY; Exercise 12.3 asks you to extend this change to probabilistic CKY.

Let's see an example of the probabilistic CKY chart, using the following mini-grammar, which is already in CNF:

S	$\rightarrow$	$NP\ VP$	.80	Det	$\rightarrow$	the	.40
NP	$\rightarrow$	$Det\ N$	.30	Det	$\rightarrow$	a	.40
VP	$\rightarrow$	$V\ NP$	.20	N	$\rightarrow$	$meal$	.01
V	$\rightarrow$	$includes$	.05	N	$\rightarrow$	$flight$	.02

Given this grammar, Fig. 12.4 shows the first steps in the probabilistic CKY parse of the following example:

(12.17) The flight includes a meal

12.3 Ways to Learn PCFG Rule Probabilities

Where do PCFG rule probabilities come from? There are two ways to learn probabilities for the rules of a grammar. The simplest way is to use a treebank, a corpus of already parsed sentences. Recall that we introduced in Chapter 10 the idea of treebanks and the commonly used **Penn Treebank** (Marcus et al., 1993), a collection of parse trees in English, Chinese, and other languages that is distributed by the Linguistic Data Consortium. Given a treebank, we can compute the probability of each expansion of a non-terminal by counting the number of times that expansion

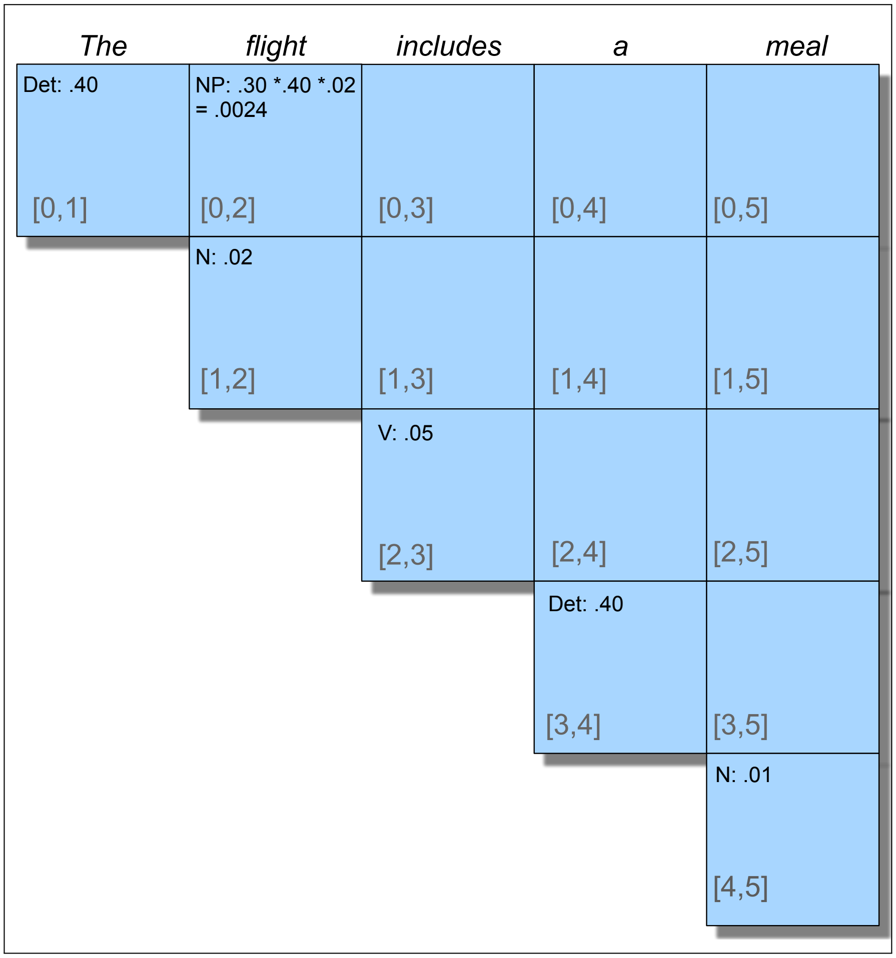


Figure 12.4 The beginning of the probabilistic CKY matrix. Filling out the rest of the chart is left as Exercise 12.4 for the reader.

occurs and then normalizing.

$$P(\alpha \rightarrow \beta|\alpha) = \frac{\text{Count}(\alpha \rightarrow \beta)}{\sum_{\gamma}\text{Count}(\alpha \rightarrow \gamma)} = \frac{\text{Count}(\alpha \rightarrow \beta)}{\text{Count}(\alpha)} \tag{12.18}$$

If we don't have a treebank but we do have a (non-probabilistic) parser, we can generate the counts we need for computing PCFG rule probabilities by first parsing a corpus of sentences with the parser. If sentences were unambiguous, it would be as simple as this: parse the corpus, increment a counter for every rule in the parse, and then normalize to get probabilities.

But wait! Since most sentences are ambiguous, that is, have multiple parses, we don't know which parse to count the rules in. Instead, we need to keep a separate count for each parse of a sentence and weight each of these partial counts by the probability of the parse it appears in. But to get these parse probabilities to weight the rules, we need to already have a probabilistic parser.

The intuition for solving this chicken-and-egg problem is to incrementally improve our estimates by beginning with a parser with equal rule probabilities, then parse the sentence, compute a probability for each parse, use these probabilities to

weight the counts, re-estimate the rule probabilities, and so on, until our probabilities converge. The standard algorithm for computing this solution is called the **inside-outside** algorithm; it was proposed by Baker (1979) as a generalization of the forward-backward algorithm for HMMs. Like forward-backward, inside-outside is a special case of the Expectation Maximization (EM) algorithm, and hence has two steps: the **expectation step**, and the **maximization step**. See Lari and Young (1990) or Manning and Schütze (1999) for a complete description of the algorithm.

inside-outside

expectation step

maximization step

This use of the inside-outside algorithm to estimate the rule probabilities for a grammar is actually a kind of limited use of inside-outside. The inside-outside algorithm can actually be used not only to set the rule probabilities but even to induce the grammar rules themselves. It turns out, however, that grammar induction is so difficult that inside-outside by itself is not a very successful grammar inducer; see the Historical Notes at the end of the chapter for pointers to other grammar induction algorithms.

12.4 Problems with PCFGs

While probabilistic context-free grammars are a natural extension to context-free grammars, they have two main problems as probability estimators:

Poor independence assumptions: CFG rules impose an independence assumption on probabilities, resulting in poor modeling of structural dependencies across the parse tree.

Lack of lexical conditioning: CFG rules don't model syntactic facts about specific words, leading to problems with subcategorization ambiguities, preposition attachment, and coordinate structure ambiguities.

Because of these problems, most current probabilistic parsing models use some augmented version of PCFGs, or modify the Treebank-based grammar in some way. In the next few sections after discussing the problems in more detail we introduce some of these augmentations.

12.4.1 Independence Assumptions Miss Structural Dependencies Between Rules

Let's look at these problems in more detail. Recall that in a CFG the expansion of a non-terminal is independent of the context, that is, of the other nearby non-terminals in the parse tree. Similarly, in a PCFG, the probability of a particular rule like $NP \rightarrow Det\ N$ is also independent of the rest of the tree. By definition, the probability of a group of independent events is the product of their probabilities. These two facts explain why in a PCFG we compute the probability of a tree by just multiplying the probabilities of each non-terminal expansion.

Unfortunately, this CFG independence assumption results in poor probability estimates. This is because in English the choice of how a node expands can after all depend on the location of the node in the parse tree. For example, in English it turns out that *NP*s that are syntactic **subjects** are far more likely to be pronouns, and *NP*s that are syntactic **objects** are far more likely to be non-pronominal (e.g., a proper noun or a determiner noun sequence), as shown by these statistics for *NP*s in the

Switchboard corpus (Francis et al., 1999):[1]

	Pronoun	Non-Pronoun
Subject	91%	9%
Object	34%	66%

Unfortunately, there is no way to represent this contextual difference in the probabilities in a PCFG. Consider two expansions of the non-terminal *NP* as a pronoun or as a determiner+noun. How shall we set the probabilities of these two rules? If we set their probabilities to their overall probability in the Switchboard corpus, the two rules have about equal probability.

$$\begin{array}{lcll} NP & \rightarrow & DT\ NN & .28 \\ NP & \rightarrow & PRP & .25 \end{array}$$

Because PCFGs don't allow a rule probability to be conditioned on surrounding context, this equal probability is all we get; there is no way to capture the fact that in subject position, the probability for $NP \rightarrow PRP$ should go up to .91, while in object position, the probability for $NP \rightarrow DT\ NN$ should go up to .66.

These dependencies could be captured if the probability of expanding an *NP* as a pronoun (e.g., $NP \rightarrow PRP$) versus a lexical *NP* (e.g., $NP \rightarrow DT\ NN$) were *conditioned* on whether the *NP* was a subject or an object. Section 12.5 introduces the technique of **parent annotation** for adding this kind of conditioning.

12.4.2 Lack of Sensitivity to Lexical Dependencies

A second class of problems with PCFGs is their lack of sensitivity to the words in the parse tree. Words do play a role in PCFGs since the parse probability includes the probability of a word given a part-of-speech (i.e., from rules like $V \rightarrow sleep$, $NN \rightarrow book$, etc.).

But it turns out that lexical information is useful in other places in the grammar, such as in resolving prepositional phrase (*PP*) attachment ambiguities. Since prepositional phrases in English can modify a noun phrase or a verb phrase, when a parser finds a prepositional phrase, it must decide where to attach it into the tree. Consider the following example:

(12.19) Workers dumped sacks into a bin.

Figure 12.5 shows two possible parse trees for this sentence; the one on the left is the correct parse; Fig. 12.6 shows another perspective on the preposition attachment problem, demonstrating that resolving the ambiguity in Fig. 12.5 is equivalent to deciding whether to attach the prepositional phrase into the rest of the tree at the *NP* or *VP* nodes; we say that the correct parse requires **VP attachment**, and the incorrect parse implies **NP attachment**.

VP attachment
NP attachment

Why doesn't a PCFG already deal with *PP* attachment ambiguities? Note that the two parse trees in Fig. 12.5 have almost exactly the same rules; they differ only in that the left-hand parse has this rule:

$$VP \rightarrow VBD\ NP\ PP$$

[1] Distribution of subjects from 31,021 declarative sentences; distribution of objects from 7,489 sentences. This tendency is caused by the use of subject position to realize the **topic** or old information in a sentence (Givón, 1990). Pronouns are a way to talk about old information, while non-pronominal ("lexical") noun-phrases are often used to introduce new referents. We talk more about new and old information in Chapter 22.

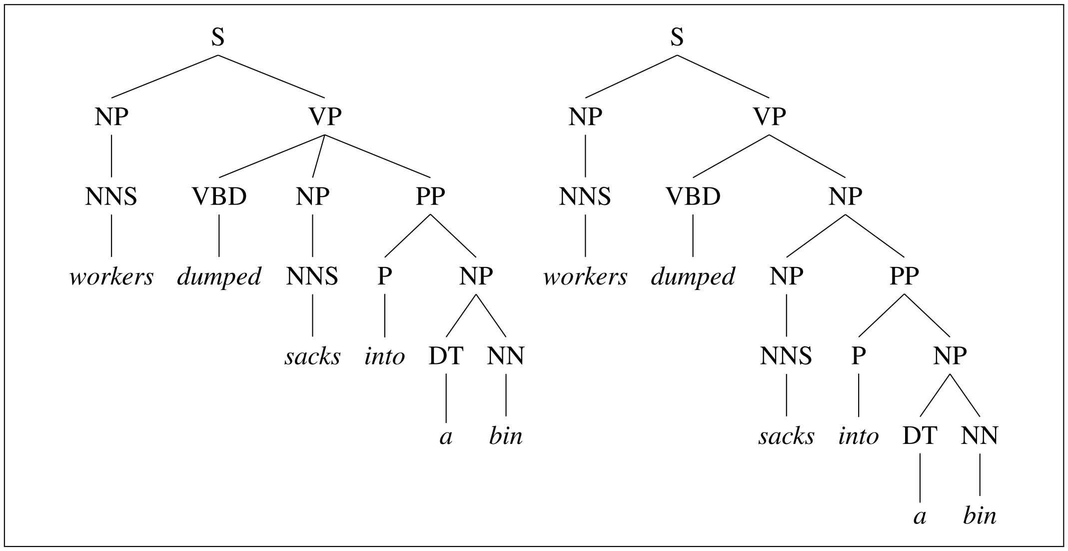


Figure 12.5 Two possible parse trees for a **prepositional phrase attachment ambiguity**. The left parse is the sensible one, in which "into a bin" describes the resulting location of the sacks. In the right incorrect parse, the sacks to be dumped are the ones which are already "into a bin", whatever that might mean.

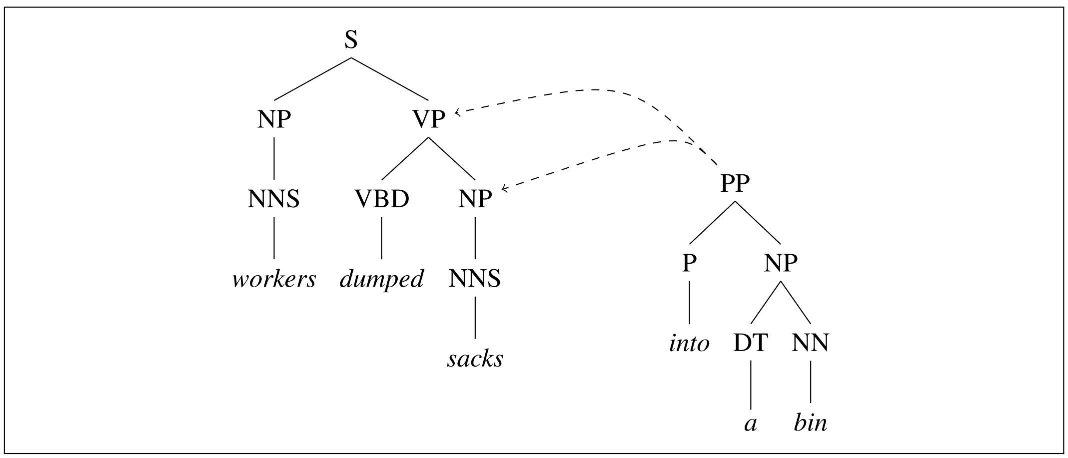


Figure 12.6 Another view of the preposition attachment problem. Should the *PP* on the right attach to the *VP* or *NP* nodes of the partial parse tree on the left?

while the right-hand parse has these:

$$
\begin{aligned}
VP &\rightarrow VBD\ NP \\
NP &\rightarrow NP\ PP
\end{aligned}
$$

Depending on how these probabilities are set, a PCFG will **always** either prefer *NP* attachment or *VP* attachment. As it happens, *NP* attachment is slightly more common in English, so if we trained these rule probabilities on a corpus, we might always prefer *NP* attachment, causing us to misparse this sentence.

But suppose we set the probabilities to prefer the *VP* attachment for this sentence. Now we would misparse the following sentence, which requires *NP* attachment:

(12.20) fishermen caught tons of herring

What information in the input sentence lets us know that (12.20) requires *NP* attachment while (12.19) requires *VP* attachment?

It should be clear that these preferences come from the identities of the verbs, nouns, and prepositions. It seems that the affinity between the verb *dumped* and the preposition *into* is greater than the affinity between the noun *sacks* and the preposition *into*, thus leading to *VP* attachment. On the other hand, in (12.20) the affinity between *tons* and *of* is greater than that between *caught* and *of*, leading to *NP* attachment.

lexical dependency

Thus, to get the correct parse for these kinds of examples, we need a model that somehow augments the PCFG probabilities to deal with these **lexical dependency** statistics for different verbs and prepositions.

Coordination ambiguities are another case in which lexical dependencies are the key to choosing the proper parse. Figure 12.7 shows an example from Collins (1999) with two parses for the phrase *dogs in houses and cats*. Because *dogs* is semantically a better conjunct for *cats* than *houses* (and because most dogs can't fit inside cats), the parse *[dogs in [$_{NP}$ houses and cats]]* is intuitively unnatural and should be dispreferred. The two parses in Fig. 12.7, however, have exactly the same PCFG rules, and thus a PCFG will assign them the same probability.

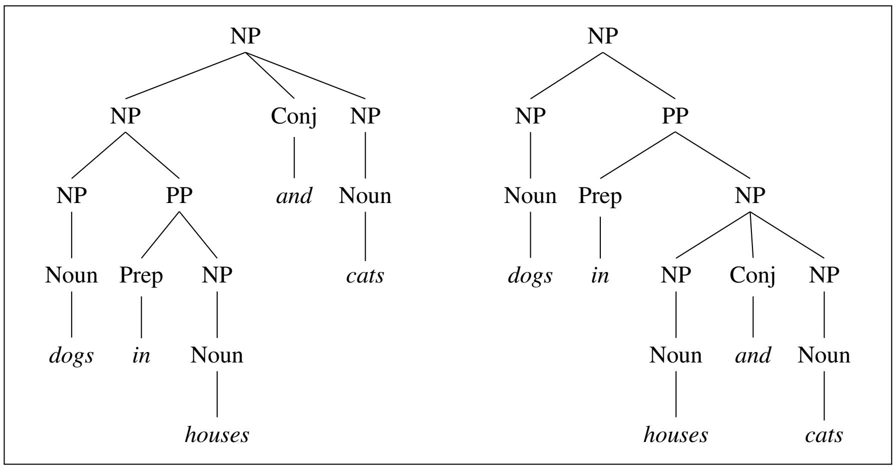


Figure 12.7 An instance of coordination ambiguity. Although the left structure is intuitively the correct one, a PCFG will assign them identical probabilities since both structures use exactly the same set of rules. After Collins (1999).

In summary, we have shown in this section and the previous one that probabilistic context-free grammars are incapable of modeling important **structural** and **lexical** dependencies. In the next two sections we sketch current methods for augmenting PCFGs to deal with both these issues.

12.5 Improving PCFGs by Splitting Non-Terminals

Let's start with the first of the two problems with PCFGs mentioned above: their inability to model structural dependencies, like the fact that NPs in subject position tend to be pronouns, whereas *NP*s in object position tend to have full lexical (non-pronominal) form. How could we augment a PCFG to correctly model this fact?

split

One idea would be to **split** the *NP* non-terminal into two versions: one for sub-

jects, one for objects. Having two nodes (e.g., $NP_{subject}$ and NP_{object}) would allow us to correctly model their different distributional properties, since we would have different probabilities for the rule $NP_{subject} \rightarrow PRP$ and the rule $NP_{object} \rightarrow PRP$.

parent annotation

One way to implement this intuition of splits is to do **parent annotation** (Johnson, 1998), in which we annotate each node with its parent in the parse tree. Thus, an *NP* node that is the subject of the sentence and hence has parent *S* would be annotated *NPˆS*, while a direct object *NP* whose parent is *VP* would be annotated *NPˆVP*. Figure 12.8 shows an example of a tree produced by a grammar that parent-annotates the phrasal non-terminals (like *NP* and *VP*).

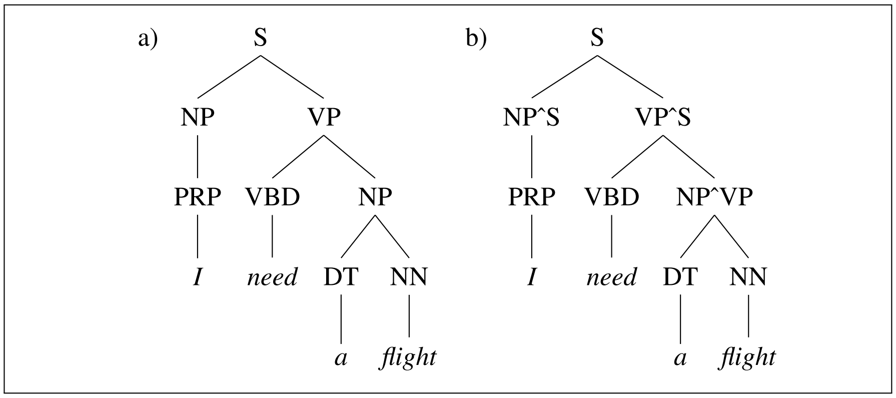


Figure 12.8 A standard PCFG parse tree (a) and one which has **parent annotation** on the nodes which aren't pre-terminal (b). All the non-terminal nodes (except the pre-terminal part-of-speech nodes) in parse (b) have been annotated with the identity of their parent.

In addition to splitting these phrasal nodes, we can also improve a PCFG by splitting the pre-terminal part-of-speech nodes (Klein and Manning, 2003b). For example, different kinds of adverbs (RB) tend to occur in different syntactic positions: the most common adverbs with ADVP parents are *also* and *now*, with *VP* parents *n't* and *not*, and with *NP* parents *only* and *just*. Thus, adding tags like RBˆADVP, RBˆVP, and RBˆNP can be useful in improving PCFG modeling.

Similarly, the Penn Treebank tag IN can mark a wide variety of parts-of-speech, including subordinating conjunctions (*while*, *as*, *if*), complementizers (*that*, *for*), and prepositions (*of*, *in*, *from*). Some of these differences can be captured by parent annotation (subordinating conjunctions occur under S, prepositions under PP), while others require specifically splitting the pre-terminal nodes. Figure 12.9 shows an example from Klein and Manning (2003b) in which even a parent-annotated grammar incorrectly parses *works* as a noun in *to see if advertising works*. Splitting pre-terminals to allow *if* to prefer a sentential complement results in the correct verbal parse.

To deal with cases in which parent annotation is insufficient, we can also hand-write rules that specify a particular node split based on other features of the tree. For example, to distinguish between complementizer IN and subordinating conjunction IN, both of which can have the same parent, we could write rules conditioned on other aspects of the tree such as the lexical identity (the lexeme *that* is likely to be a complementizer, *as* a subordinating conjunction).

Node-splitting is not without problems; it increases the size of the grammar and hence reduces the amount of training data available for each grammar rule, leading to overfitting. Thus, it is important to split to just the correct level of granularity for a particular training set. While early models employed hand-written rules to try to find an optimal number of non-terminals (Klein and Manning, 2003b), modern models

split and merge

automatically search for the optimal splits. The **split and merge** algorithm of Petrov et al. (2006), for example, starts with a simple X-bar grammar, alternately splits the non-terminals, and merges non-terminals, finding the set of annotated nodes that maximizes the likelihood of the training set treebank. As of the time of this writing, the performance of the Petrov et al. (2006) algorithm was the best of any known parsing algorithm on the Penn Treebank.

12.6 Probabilistic Lexicalized CFGs

The previous section showed that a simple probabilistic CKY algorithm for parsing raw PCFGs can achieve extremely high parsing accuracy if the grammar rule symbols are redesigned by automatic splits and merges.

In this section, we discuss an alternative family of models in which instead of modifying the grammar rules, we modify the probabilistic model of the parser to allow for **lexicalized** rules. The resulting family of lexicalized parsers includes the well-known **Collins parser** (Collins, 1999) and **Charniak parser** (Charniak, 1997), both of which are publicly available and widely used throughout natural language processing.

Collins parser

Charniak parser

We saw in Section 10.4.3 that syntactic constituents could be associated with a lexical **head**, and we defined a **lexicalized grammar** in which each non-terminal in the tree is annotated with its lexical head, where a rule like $VP \rightarrow VBD\ NP\ PP$ would be extended as

lexicalized grammar

$$VP(dumped) \rightarrow VBD(dumped)\ NP(sacks)\ PP(into) \tag{12.21}$$

In the standard type of lexicalized grammar, we actually make a further extension, which is to associate the **head tag**, the part-of-speech tags of the headwords, with the non-terminal symbols as well. Each rule is thus lexicalized by both the

head tag

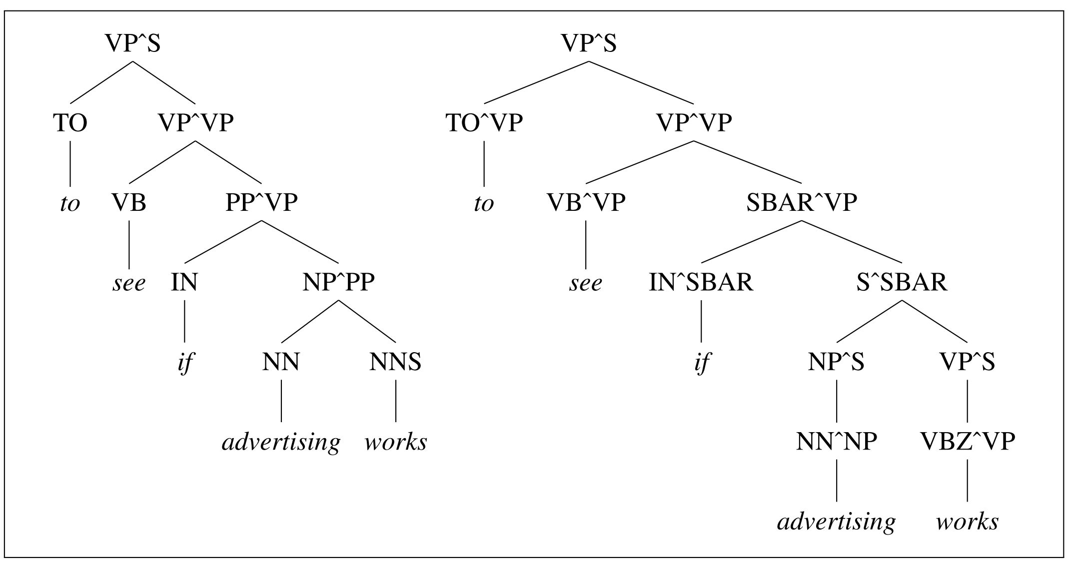
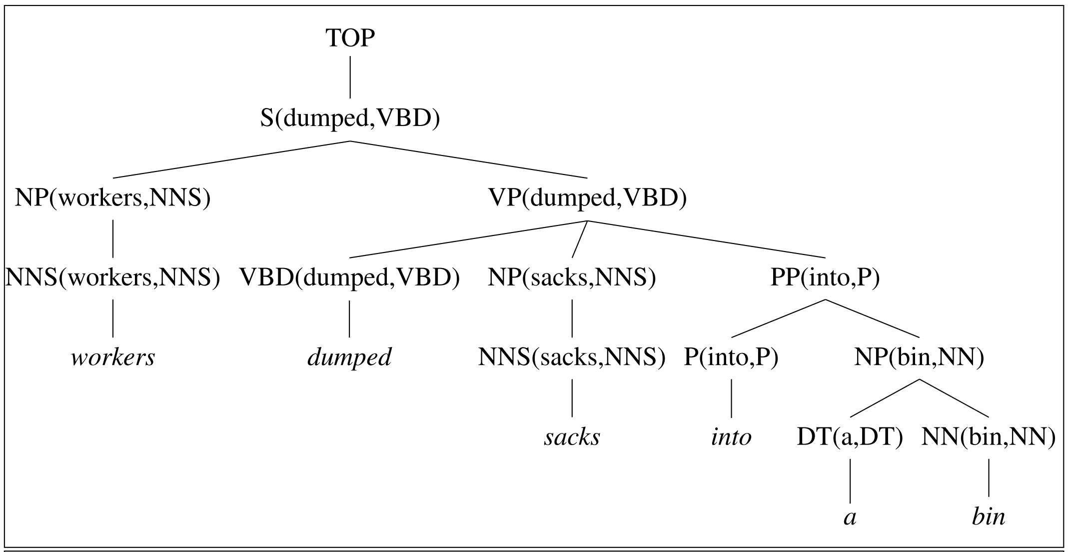


Figure 12.9 An incorrect parse even with a parent-annotated parse (left). The correct parse (right), was produced by a grammar in which the pre-terminal nodes have been split, allowing the probabilistic grammar to capture the fact that *if* prefers sentential complements. Adapted from Klein and Manning (2003b).

headword and the head tag of each constituent resulting in a format for lexicalized rules like

$$VP(dumped,VBD) \rightarrow VBD(dumped,VBD)\ \ NP(sacks,NNS)\ \ PP(into,P) \quad (12.22)$$

We show a lexicalized parse tree with head tags in Fig. 12.10, extended from Fig. 10.11.



Internal Rules					**Lexical Rules**		
TOP	→	S(dumped,VBD)			NNS(workers,NNS)	→	workers
S(dumped,VBD)	→	NP(workers,NNS)	VP(dumped,VBD)		VBD(dumped,VBD)	→	dumped
NP(workers,NNS)	→	NNS(workers,NNS)			NNS(sacks,NNS)	→	sacks
VP(dumped,VBD)	→	VBD(dumped, VBD)	NP(sacks,NNS)	PP(into,P)	P(into,P)	→	into
PP(into,P)	→	P(into,P)	NP(bin,NN)		DT(a,DT)	→	a
NP(bin,NN)	→	DT(a,DT)	NN(bin,NN)		NN(bin,NN)	→	bin

Figure 12.10 A lexicalized tree, including head tags, for a WSJ sentence, adapted from Collins (1999). Below we show the PCFG rules that would be needed for this parse tree, internal rules on the left, and lexical rules on the right.

To generate such a lexicalized tree, each PCFG rule must be augmented to identify one right-hand constituent to be the head daughter. The headword for a node is then set to the headword of its head daughter, and the head tag to the part-of-speech tag of the headword. Recall that we gave in Fig. 10.12 a set of hand-written rules for identifying the heads of particular constituents.

A natural way to think of a lexicalized grammar is as a parent annotation, that is, as a simple context-free grammar with many copies of each rule, one copy for each possible headword/head tag for each constituent. Thinking of a probabilistic lexicalized CFG in this way would lead to the set of simple PCFG rules shown below the tree in Fig. 12.10.

lexical rules
internal rules

Note that Fig. 12.10 shows two kinds of rules: **lexical rules**, which express the expansion of a pre-terminal to a word, and **internal rules**, which express the other rule expansions. We need to distinguish these kinds of rules in a lexicalized grammar because they are associated with very different kinds of probabilities. The lexical rules are deterministic, that is, they have probability 1.0 since a lexicalized pre-terminal like $NN(bin,NN)$ can only expand to the word *bin*. But for the internal rules, we need to estimate probabilities.

Suppose we were to treat a probabilistic lexicalized CFG like a really big CFG that just happened to have lots of very complex non-terminals and estimate the probabilities for each rule from maximum likelihood estimates. Thus, according to Eq. 12.18, the MLE estimate for the probability for the rule *P(VP(dumped,VBD) → VBD(dumped, VBD) NP(sacks,NNS) PP(into,P))* would be

$$\frac{Count(VP(dumped,VBD) \rightarrow VBD(dumped,\ VBD)\ NP(sacks,NNS)\ PP(into,P))}{Count(VP(dumped,VBD))} \tag{12.23}$$

But there's no way we can get good estimates of counts like those in (12.23) because they are so specific: we're unlikely to see many (or even any) instances of a sentence with a verb phrase headed by *dumped* that has one *NP* argument headed by *sacks* and a *PP* argument headed by *into*. In other words, counts of fully lexicalized PCFG rules like this will be far too sparse, and most rule probabilities will come out 0.

The idea of lexicalized parsing is to make some further independence assumptions to break down each rule so that we would estimate the probability

$$P(VP(dumped,VBD) \rightarrow VBD(dumped,\ VBD)\ NP(sacks,NNS)\ PP(into,P))$$

as the product of smaller independent probability estimates for which we could acquire reasonable counts. The next section summarizes one such method, the Collins parsing method.

12.6.1 The Collins Parser

Modern statistical parsers differ in exactly which independence assumptions they make. In this section we describe a simplified version of Collins's worth knowing about; see the summary at the end of the chapter.

The first intuition of the Collins parser is to think of the right-hand side of every (internal) CFG rule as consisting of a head non-terminal, together with the non-terminals to the left of the head and the non-terminals to the right of the head. In the abstract, we think about these rules as follows:

$$LHS \rightarrow L_n L_{n-1} \ldots L_1\, H\, R_1 \ldots R_{n-1}\, R_n \tag{12.24}$$

Since this is a lexicalized grammar, each of the symbols like L_1 or R_3 or H or LHS is actually a complex symbol representing the category and its head and head tag, like *VP(dumped,VP)* or *NP(sacks,NNS)*.

Now, instead of computing a single MLE probability for this rule, we are going to break down this rule via a neat generative story, a slight simplification of what is called Collins Model 1. This new generative story is that given the left-hand side, we first generate the head of the rule and then generate the dependents of the head, one by one, from the inside out. Each of these generation steps will have its own probability.

We also add a special STOP non-terminal at the left and right edges of the rule; this non-terminal allows the model to know when to stop generating dependents on a given side. We generate dependents on the left side of the head until we've generated STOP on the left side of the head, at which point we move to the right side of the head and start generating dependents there until we generate STOP. So it's as if we

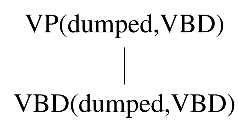
are generating a rule augmented as follows:

$$\begin{aligned}&P(VP(dumped,VBD) \rightarrow \\ &\quad \text{STOP } VBD(dumped,\ VBD)\ NP(sacks,NNS)\ PP(into,P)\ \text{STOP})\end{aligned} \tag{12.25}$$

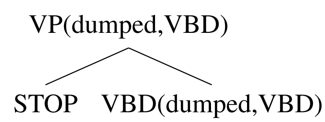
Let's see the generative story for this augmented rule. We make use of three kinds of probabilities: P_H for generating heads, P_L for generating dependents on the left, and P_R for generating dependents on the right.

---

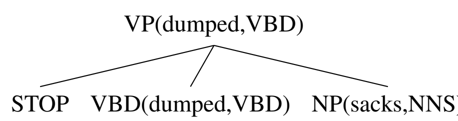
1. Generate the head VBD(dumped,VBD) with probability
P(H|LHS) = P(VBD(dumped,VBD) | VP(dumped,VBD))



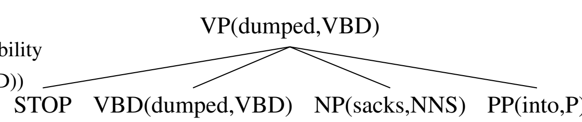
2. Generate the left dependent (which is STOP, since there isn't one) with probability
P(STOP| VP(dumped,VBD) VBD(dumped,VBD))



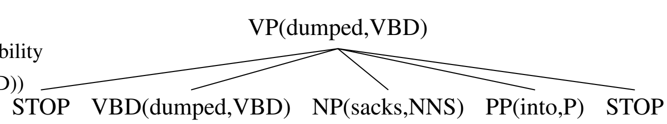
3. Generate right dependent NP(sacks,NNS) with probability
P_r(NP(sacks,NNS| VP(dumped,VBD), VBD(dumped,VBD))



4. Generate the right dependent PP(into,P) with probability
P_r(PP(into,P) | VP(dumped,VBD), VBD(dumped,VBD))



5) Generate the right dependent STOP with probability
P_r(STOP | VP(dumped,VBD), VBD(dumped,VBD))



In summary, the probability of this rule

$$\begin{aligned}&P(VP(dumped,VBD) \rightarrow \\ &\quad VBD(dumped,\ VBD)\ \ NP(sacks,NNS)\ \ PP(into,P))\end{aligned} \tag{12.26}$$

is estimated as

$$\begin{aligned} P_H(VBD|VP,\ dumped)\ &\times\ P_L(STOP|VP,VBD,dumped) \\ &\times\ P_R(NP(sacks,NNS)|VP,VBD,dumped) \\ &\times\ P_R(PP(into,P)|VP,VBD,dumped) \\ &\times\ P_R(STOP|VP,VBD,dumped) \end{aligned} \tag{12.27}$$

Each of these probabilities can be estimated from much smaller amounts of data than the full probability in (12.26). For example, the maximum likelihood estimate

for the component probability $P_R(NP(sacks,NNS)|VP, VBD, dumped)$ is

$$\frac{\text{Count}(\ VP(dumped,VBD)\ \text{ with }\ NNS(sacks)\text{as a daughter somewhere on the right}\ )}{\text{Count}(\ VP(dumped,VBD)\ )} \tag{12.28}$$

These counts are much less subject to sparsity problems than are complex counts like those in (12.26).

More generally, if H is a head with head word hw and head tag ht, lw/lt and rw/rt are the word/tag on the left and right respectively, and P is the parent, then the probability of an entire rule can be expressed as follows:

1. Generate the head of the phrase $H(hw, ht)$ with probability:

$$P_H(H(hw, ht)|P, hw, ht)$$

2. Generate modifiers to the left of the head with total probability

$$\prod_{i=1}^{n+1} P_L(L_i(lw_i, lt_i)|P, H, hw, ht)$$

such that $L_{n+1}(lw_{n+1}, lt_{n+1}) =$STOP, and we stop generating once we've generated a STOP token.

3. Generate modifiers to the right of the head with total probability:

$$\prod_{i=1}^{n+1} P_P(R_i(rw_i, rt_i)|P, H, hw, ht)$$

such that $R_{n+1}(rw_{n+1}, rt_{n+1}) = STOP$, and we stop generating once we've generated a STOP token.

12.6.2 Advanced: Further Details of the Collins Parser

distance

The actual Collins parser models are more complex (in a couple of ways) than the simple model presented in the previous section. Collins Model 1 includes a **distance** feature. Thus, instead of computing P_L and P_R as follows,

$$P_L(L_i(lw_i, lt_i)|P, H, hw, ht) \tag{12.29}$$
$$P_R(R_i(rw_i, rt_i)|P, H, hw, ht) \tag{12.30}$$

Collins Model 1 conditions also on a distance feature:

$$P_L(L_i(lw_i, lt_i)|P, H, hw, ht, distance_L(i-1)) \tag{12.31}$$
$$P_R(R_i(rw_i, rt_i)|P, H, hw, ht, distance_R(i-1)) \tag{12.32}$$

The distance measure is a function of the sequence of words *below* the previous modifiers (i.e., the words that are the yield of each modifier non-terminal we have already generated on the left).

The simplest version of this distance measure is just a tuple of two binary features based on the surface string below these previous dependencies: (1) Is the string of length zero? (i.e., were no previous words generated?) (2) Does the string contain a verb?

Collins Model 2 adds more sophisticated features, conditioning on subcategorization frames for each verb and distinguishing arguments from adjuncts.

Finally, smoothing is as important for statistical parsers as it was for N-gram models. This is particularly true for lexicalized parsers, since the lexicalized rules will otherwise condition on many lexical items that may never occur in training (even using the Collins or other methods of independence assumptions).

Consider the probability $P_R(R_i(rw_i, rt_i)|P, hw, ht)$. What do we do if a particular right-hand constituent never occurs with this head? The Collins model addresses this problem by interpolating three backed-off models: fully lexicalized (conditioning on the headword), backing off to just the head tag, and altogether unlexicalized.

Backoff	$P_R(R_i(rw_i, rt_i\|...)$	**Example**
1	$P_R(R_i(rw_i, rt_i)\|P, hw, ht)$	P_R(NP(sacks,NNS)|VP, VBD, dumped)
2	$P_R(R_i(rw_i, rt_i)\|P, ht)$	$P_R(NP(sacks, NNS)\|VP, VBD)$
3	$P_R(R_i(rw_i, rt_i)\|P)$	$P_R(NP(sacks, NNS)\|VP)$

Similar backoff models are built also for P_L and P_H. Although we've used the word "backoff", in fact these are not backoff models but interpolated models. The three models above are linearly interpolated, where e_1, e_2, and e_3 are the maximum likelihood estimates of the three backoff models above:

$$P_R(...) = \lambda_1 e_1 + (1-\lambda_1)(\lambda_2 e_2 + (1-\lambda_2)e_3) \tag{12.33}$$

The values of $\lambda_1 and \lambda_2$ are set to implement Witten-Bell discounting (Witten and Bell, 1991) following Bikel et al. (1997).

The Collins model deals with unknown words by replacing any unknown word in the test set, and any word occurring less than six times in the training set, with a special UNKNOWN word token. Unknown words in the test set are assigned a part-of-speech tag in a preprocessing step by the Ratnaparkhi (1996) tagger; all other words are tagged as part of the parsing process.

The parsing algorithm for the Collins model is an extension of probabilistic CKY; see Collins (2003a). Extending the CKY algorithm to handle basic lexicalized probabilities is left as Exercises 14.5 and 14.6 for the reader.

12.7 Probabilistic CCG Parsing

Lexicalized grammar frameworks such as CCG pose problems for which the phrase-based methods we've been discussing are not particularly well-suited. To quickly review, CCG consists of three major parts: a set of categories, a lexicon that associates words with categories, and a set of rules that govern how categories combine in context. Categories can be either atomic elements, such as S and NP, or functions such as $(S\backslash NP)/NP$ which specifies the transitive verb category. Rules specify how functions, their arguments, and other functions combine. For example, the following rule templates, **forward** and **backward function application**, specify the way that functions apply to their arguments.

$$\begin{aligned} X/Y\ Y &\Rightarrow X \\ Y\ X\backslash Y &\Rightarrow X \end{aligned}$$

The first rule applies a function to its argument on the right, while the second looks to the left for its argument. The result of applying either of these rules is the

category specified as the value of the function being applied. For the purposes of this discussion, we'll rely on these two rules along with the **forward** and **backward composition** rules and **type-raising**, as described in Chapter 10.

12.7.1 Ambiguity in CCG

As is always the case in parsing, managing ambiguity is the key to successful CCG parsing. The difficulties with CCG parsing arise from the ambiguity caused by the large number of complex lexical categories combined with the very general nature of the grammatical rules. To see some of the ways that ambiguity arises in a categorial framework, consider the following example.

(12.34) United diverted the flight to Reno.

Our grasp of the role of *the flight* in this example depends on whether the prepositional phrase *to Reno* is taken as a modifier of *the flight*, as a modifier of the entire verb phrase, or as a potential second argument to the verb *divert*. In a context-free grammar approach, this ambiguity would manifest itself as a choice among the following rules in the grammar.

$$\begin{aligned}
Nominal &\rightarrow Nominal\ PP \\
VP &\rightarrow VP\ PP \\
VP &\rightarrow Verb\ NP\ PP
\end{aligned}$$

In a phrase-structure approach we would simply assign the word *to* to the category *P* allowing it to combine with *Reno* to form a prepositional phrase. The subsequent choice of grammar rules would then dictate the ultimate derivation. In the categorial approach, we can associate *to* with distinct categories to reflect the ways in which it might interact with other elements in a sentence. The fairly abstract combinatoric rules would then sort out which derivations are possible. Therefore, the source of ambiguity arises not from the grammar but rather from the lexicon.

Let's see how this works by considering several possible derivations for this example. To capture the case where the prepositional phrase *to Reno* modifies *the flight*, we assign the preposition *to* the category $(NP\backslash NP)/NP$, which gives rise to the following derivation.

$$\begin{array}{cccccc}
\underline{\textit{United}} & \underline{\textit{diverted}} & \underline{\textit{the}} & \underline{\textit{flight}} & \underline{\textit{to}} & \underline{\textit{Reno}} \\
\mathrm{NP} & (\mathrm{S}\backslash \mathrm{NP})/\mathrm{NP} & \mathrm{NP}/\mathrm{N} & \mathrm{N} & (\mathrm{NP}\backslash \mathrm{NP})/\mathrm{NP} & \mathrm{NP} \\
 & & \multicolumn{2}{c}{\overline{\qquad \mathrm{NP} \qquad}>} & \multicolumn{2}{c}{\overline{\qquad \mathrm{NP}\backslash \mathrm{NP} \qquad}>} \\
 & & \multicolumn{4}{c}{\overline{\qquad\qquad \mathrm{NP} \qquad\qquad}<} \\
 & \multicolumn{5}{c}{\overline{\qquad\qquad \mathrm{S}\backslash \mathrm{NP} \qquad\qquad}>} \\
\multicolumn{6}{c}{\overline{\qquad\qquad\qquad \mathrm{S} \qquad\qquad\qquad}<}
\end{array}$$

Here, the category assigned to *to* expects to find two arguments: one to the right as with a traditional preposition, and one to the left that corresponds to the *NP* to be modified.

Alternatively, we could assign *to* to the category $(S\backslash S)/NP$, which permits the following derivation where to Reno modifies the preceding verb phrase.

United	*diverted*	*the*	*flight*	*to*	*Reno*
NP	(S\NP)/NP	NP/N	N	(S\S)/NP	NP
		NP (>)		S\S (>)	
	S\NP (>)				
	S\NP (<B)				
S (<)					

A third possibility is to view *divert* as a ditransitive verb by assigning it to the category $((S\backslash NP)/PP)/NP$, while treating *to Reno* as a simple prepositional phrase.

United	*diverted*	*the*	*flight*	*to*	*Reno*
NP	((S\NP)/PP)/NP	NP/N	N	PP/NP	NP
		NP (>)		PP (>)	
	(S\NP)/PP (>)				
	S\NP (>)				
S (<)					

While CCG parsers are still subject to ambiguity arising from the choice of grammar rules, including the kind of spurious ambiguity discussed in Chapter 10, it should be clear that the choice of lexical categories is the primary problem to be addressed in CCG parsing.

12.7.2 CCG Parsing Frameworks

Since the rules in combinatory grammars are either binary or unary, a bottom-up, tabular approach based on the CKY algorithm should be directly applicable to CCG parsing. Recall from Fig. 12.3 that PCKY employs a table that records the location, category and probability of all valid constituents discovered in the input. Given an appropriate probability model for CCG derivations, the same kind of approach can work for CCG parsing.

Unfortunately, the large number of lexical categories available for each word, combined with the promiscuity of CCG's combinatoric rules, leads to an explosion in the number of (mostly useless) constituents added to the parsing table. The key to managing this explosion of zombie constituents is to accurately assess and exploit the most likely lexical categories possible for each word — a process called supertagging.

The following sections describe two approaches to CCG parsing that make use of supertags. Section 12.7.4, presents an approach that structures the parsing process as a heuristic search through the use of the A* algorithm. The following section then briefly describes a more traditional maximum entropy approach that manages the search space complexity through the use of **adaptive supertagging** — a process that iteratively considers more and more tags until a parse is found.

12.7.3 Supertagging

Chapter 8 introduced the task of part-of-speech tagging, the process of assigning the
supertagging correct lexical category to each word in a sentence. **Supertagging** is the corresponding task for highly lexicalized grammar frameworks, where the assigned tags often dictate much of the derivation for a sentence.

CCG supertaggers rely on treebanks such as CCGbank to provide both the overall set of lexical categories as well as the allowable category assignments for each word in the lexicon. CCGbank includes over 1000 lexical categories, however, in practice, most supertaggers limit their tagsets to those tags that occur at least 10 times in the training corpus. This results in an overall total of around 425 lexical categories available for use in the lexicon. Note that even this smaller number is large in contrast to the 45 POS types used by the Penn Treebank tagset.

As with traditional part-of-speech tagging, the standard approach to building a CCG supertagger is to use supervised machine learning to build a sequence classifier using labeled training data. A common approach is to use the maximum entropy Markov model (MEMM), as described in Chapter 8, to find the most likely sequence of tags given a sentence. The features in such a model consist of the current word w_i, its surrounding words within l words w_{i-l}^{i+l}, as well as the k previously assigned supertags t_{i-k}^{i-1}. This type of model is summarized in the following equation from Chapter 8. Training by maximizing log-likelihood of the training corpus and decoding via the Viterbi algorithm are the same as described in Chapter 8.

$$
\begin{aligned}
\hat{T} &= \underset{T}{\operatorname{argmax}}\, P(T|W) \\
&= \underset{T}{\operatorname{argmax}} \prod_i P(t_i|w_{i-l}^{i+l}, t_{i-k}^{i-1}) \\
&= \underset{T}{\operatorname{argmax}} \prod_i \frac{\exp\left(\sum_i w_i f_i(t_i, w_{i-l}^{i+l}, t_{i-k}^{i-1})\right)}{\sum_{t' \in \text{tagset}} \exp\left(\sum_i w_i f_i(t', w_{i-l}^{i+l}, t_{i-k}^{i-1})\right)}
\end{aligned}
\qquad (12.35)
$$

Word and tag-based features with k and l both set to 2 provides reasonable results given sufficient training data. Additional features such as POS tags and short character suffixes are also commonly used to improve performance.

Unfortunately, even with additional features the large number of possible supertags combined with high per-word ambiguity leads to error rates that are too high for practical use in a parser. More specifically, the single best tag sequence $\hat{T}$ will typically contain too many incorrect tags for effective parsing to take place. To overcome this, we can instead return a probability distribution over the possible supertags for each word in the input. The following table illustrates an example distribution for a simple example sentence. In this table, each column represents the probability of each supertag for a given word *in the context of the input sentence*. The "..." represent all the remaining supertags possible for each word.

United	serves	Denver
N/N: 0.4	$(S\backslash NP)/NP$: 0.8	NP: 0.9
NP: 0.3	N: 0.1	N/N: 0.05
S/S: 0.1	...	...
$S\backslash S$: .05		
...		

In a MEMM framework, the probability of the optimal tag sequence defined in Eq. 12.35 is efficiently computed with a suitably modified version of the Viterbi algorithm. However, since Viterbi only finds the single best tag sequence it doesn't

provide exactly what we need here; we need to know the probability of each possible word/tag pair. The probability of any given tag for a word is the sum of the probabilities of all the supertag sequences that contain that tag at that location. A table representing these values can be computed efficiently by using a version of the forward-backward algorithm used for HMMs.

The same result can also be achieved through the use of deep learning approaches based on recurrent neural networks (RNNs). Recent efforts have demonstrated considerable success with RNNs as alternatives to HMM-based methods. These approaches differ from traditional classifier-based methods in the following ways:

- The use of vector-based word representations (embeddings) rather than word-based feature functions.
- Input representations that span the entire sentence, as opposed to size-limited sliding windows.
- Avoiding the use of high-level features, such as part of speech tags, since errors in tag assignment can propagate to errors in supertags.

As with the forward-backward algorithm, RNN-based methods can provide a probability distribution over the lexical categories for each word in the input.

12.7.4 CCG Parsing using the A* Algorithm

The A* algorithm is a heuristic search method that employs an agenda to find an optimal solution. Search states representing partial solutions are added to an agenda based on a cost function, with the least-cost option being selected for further exploration at each iteration. When a state representing a complete solution is first selected from the agenda, it is guaranteed to be optimal and the search terminates.

The A* cost function, $f(n)$, is used to efficiently guide the search to a solution. The f-cost has two components: $g(n)$, the exact cost of the partial solution represented by the state n, and $h(n)$ a heuristic approximation of the cost of a solution that makes use of n. When $h(n)$ satisfies the criteria of not overestimating the actual cost, A* will find an optimal solution. Not surprisingly, the closer the heuristic can get to the actual cost, the more effective A* is at finding a solution without having to explore a significant portion of the solution space.

When applied to parsing, search states correspond to edges representing completed constituents. As with the PCKY algorithm, edges specify a constituent's start and end positions, its grammatical category, and its f-cost. Here, the g component represents the current cost of an edge and the h component represents an estimate of the cost to complete a derivation that makes use of that edge. The use of A* for phrase structure parsing originated with (Klein and Manning, 2003a), while the CCG approach presented here is based on (Lewis and Steedman, 2014).

Using information from a supertagger, an agenda and a parse table are initialized with states representing all the possible lexical categories for each word in the input, along with their f-costs. The main loop removes the lowest cost edge from the agenda and tests to see if it is a complete derivation. If it reflects a complete derivation it is selected as the best solution and the loop terminates. Otherwise, new states based on the applicable CCG rules are generated, assigned costs, and entered into the agenda to await further processing. The loop continues until a complete derivation is discovered, or the agenda is exhausted, indicating a failed parse. The algorithm is given in Fig. 12.11.

```
function CCG-ASTAR-PARSE(words) returns table or failure

  supertags ← SUPERTAGGER(words)
  for i ← from 1 to LENGTH(words) do
      for all {A | (words[i], A, score) ∈ supertags}
          edge ← MAKEEDGE(i − 1, i, A, score)
          table ← INSERTEDGE(table, edge)
          agenda ← INSERTEDGE(agenda, edge)
  loop do
     if EMPTY?(agenda) return failure
     current ← POP(agenda)
     if COMPLETEDPARSE?(current) return table
     table ← INSERTEDGE(chart, edge)
     for each rule in APPLICABLERULES(edge) do
        successor ← APPLY(rule, edge)
        if successor not ∈ in agenda or chart
            agenda ← INSERTEDGE(agenda, successor)
        else if successor ∈ agenda with higher cost
            agenda ← REPLACEEDGE(agenda, successor)
```

Figure 12.11 A*-based CCG parsing.

Heuristic Functions

Before we can define a heuristic function for our A* search, we need to decide how to assess the quality of CCG derivations. For the generic PCFG model, we defined the probability of a tree as the product of the probability of the rules that made up the tree. Given CCG's lexical nature, we'll make the simplifying assumption that the probability of a CCG derivation is just the product of the probability of the supertags assigned to the words in the derivation, ignoring the rules used in the the derivation. More formally, given a sentence S and derivation D that contains suptertag sequence T, we have:

$$P(D,S) = P(T,S) \tag{12.36}$$

$$= \prod_{i=1}^{n} P(t_i|s_i) \tag{12.37}$$

To better fit with the traditional A* approach, we'd prefer to have states scored by a cost function where lower is better (i.e., we're trying to minimize the cost of a derivation). To achieve this, we'll use negative log probabilities to score derivations; this results in the following equation, which we'll use to score completed CCG derivations.

$$P(D,S) = P(T,S) \tag{12.38}$$

$$= \sum_{i=1}^{n} -\log P(t_i|s_i) \tag{12.39}$$

Given this model, we can define our f-cost as follows. The f-cost of an edge is the sum of two components: $g(n)$, the cost of the span represented by the edge, and

$h(n)$, the estimate of the cost to complete a derivation containing that edge (these are often referred to as the **inside** and **outside costs**). We'll define $g(n)$ for an edge using Equation 12.39. That is, it is just the sum of the costs of the supertags that comprise the span.

For $h(n)$, we need a score that approximates but *never overestimates* the actual cost of the final derivation. A simple heuristic that meets this requirement assumes that each of the words in the outside span will be assigned its *most probable supertag*. If these are the tags used in the final derivation, then its score will equal the heuristic. If any other tags are used in the final derivation the f-cost will be higher since the new tags must have higher costs, thus guaranteeing that we will not overestimate.

Putting this all together, we arrive at the following definition of a suitable f-cost for an edge.

$$\begin{aligned} f(w_{i,j}, t_{i,j}) &= g(w_{i,j}) + h(w_{i,j}) \\ &= \sum_{k=i}^{j} -\log P(t_k|w_k) + \\ &\quad \sum_{k=1}^{i-1} \max_{t \in tags}(-\log P(t|w_k)) + \sum_{k=j+1}^{N} \max_{t \in tags}(-\log P(t|w_k)) \end{aligned} \tag{12.40}$$

As an example, consider an edge representing the word *serves* with the supertag *N* in the following example.

(12.41) United serves Denver.

The g-cost for this edge is just the negative log probability of the tag, or X. The outside h-cost consists of the most optimistic supertag assignments for *United* and *Denver*. The resulting f-cost for this edge is therefore x+y+z = 1.494.

An Example

Fig. 12.12 shows the initial agenda and the progress of a complete parse for this example. After initializing the agenda and the parse table with information from the supertagger, it selects the best edge from the agenda — the entry for *United* with the tag N/N and f-cost 0.591. This edge does not constitute a complete parse and is therefore used to generate new states by applying all the relevant grammar rules. In this case, applying forward application to *United: N/N* and *serves: N* results in the creation of the edge *United serves: N[0,2], 1.795* to the agenda.

Skipping ahead, at the the third iteration an edge representing the complete derivation *United serves Denver, S[0,3], .716* is added to the agenda. However, the algorithm does not terminate at this point since the cost of this edge (.716) does not place it at the top of the agenda. Instead, the edge representing *Denver* with the category *NP* is popped. This leads to the addition of another edge to the agenda (type-raising *Denver*). Only after this edge is popped and dealt with does the earlier state representing a complete derivation rise to the top of the agenda where it is popped, goal tested, and returned as a solution.

The effectiveness of the A* approach is reflected in the coloring of the states in Fig. 12.12 as well as the final parsing table. The edges shown in blue (including all the initial lexical category assignments not explicitly shown) reflect states in the search space that never made it to the top of the agenda and, therefore, never

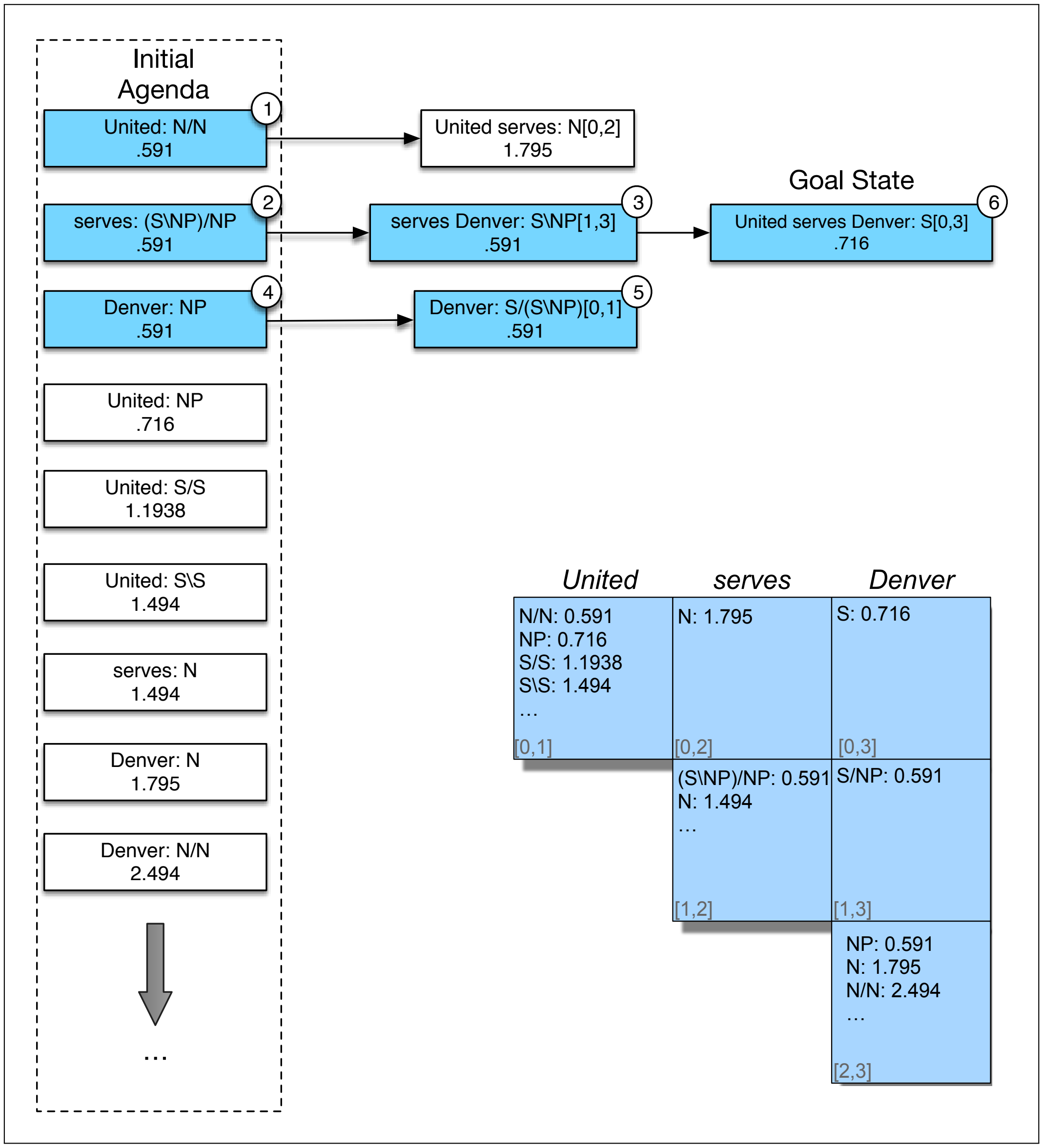


Figure 12.12 Example of an A* search for the example "United serves Denver". The circled numbers on the white boxes indicate the order in which the states are popped from the agenda. The costs in each state are based on f-costs using negative log_{10} probabilities.

contributed any edges to the final table. This is in contrast to the PCKY approach where the parser systematically fills the parse table with all possible constituents for all possible spans in the input, filling the table with myriad constituents that do not contribute to the final analysis.

12.8 Evaluating Parsers

The standard techniques for evaluating parsers and grammars are called the PARSEVAL measures; they were proposed by Black et al. (1991) and were based on the same ideas from signal-detection theory that we saw in earlier chapters. The intuition of the PARSEVAL metric is to measure how much the **constituents** in the hypothesis parse tree look like the constituents in a hand-labeled, gold-reference parse. PARSEVAL thus assumes we have a human-labeled "gold standard" parse tree for each sentence in the test set; we generally draw these gold-standard parses from a treebank like the Penn Treebank.

Given these gold-standard reference parses for a test set, a given constituent in a hypothesis parse C_h of a sentence s is labeled "correct" if there is a constituent in the reference parse C_r with the same starting point, ending point, and non-terminal symbol.

We can then measure the precision and recall just as we did for chunking in the previous chapter.

labeled recall: $= \frac{\text{\# of correct constituents in hypothesis parse of } s}{\text{\# of correct constituents in reference parse of } s}$

labeled precision: $= \frac{\text{\# of correct constituents in hypothesis parse of } s}{\text{\# of total constituents in hypothesis parse of } s}$

F-measure

As with other uses of precision and recall, instead of reporting them separately, we often report a single number, the ***F*-measure** (van Rijsbergen, 1975): The F-measure is defined as

$$F_\beta = \frac{(\beta^2+1)PR}{\beta^2 P + R}$$

The β parameter differentially weights the importance of recall and precision, based perhaps on the needs of an application. Values of $\beta > 1$ favor recall and values of $\beta < 1$ favor precision. When $\beta = 1$, precision and recall are equally balanced; this is sometimes called $F_{\beta=1}$ or just F_1:

$$F_1 = \frac{2PR}{P+R} \tag{12.42}$$

The F-measure derives from a weighted harmonic mean of precision and recall. Remember that the harmonic mean of a set of numbers is the reciprocal of the arithmetic mean of the reciprocals:

$$\text{HarmonicMean}(a_1, a_2, a_3, a_4, ..., a_n) = \frac{n}{\frac{1}{a_1} + \frac{1}{a_2} + \frac{1}{a_3} + ... + \frac{1}{a_n}} \tag{12.43}$$

and hence the F-measure is

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} \quad \text{or} \quad \left(\text{with } \beta^2 = \frac{1-\alpha}{\alpha}\right) \quad F = \frac{(\beta^2+1)PR}{\beta^2 P + R} \tag{12.44}$$

We additionally use a new metric, crossing brackets, for each sentence s:

cross-brackets: the number of constituents for which the reference parse has a bracketing such as ((A B) C) but the hypothesis parse has a bracketing such as (A (B C)).

As of the time of this writing, the performance of modern parsers that are trained and tested on the *Wall Street Journal* treebank was somewhat higher than 90% recall, 90% precision, and about 1% cross-bracketed constituents per sentence.

For comparing parsers that use different grammars, the PARSEVAL metric includes a canonicalization algorithm for removing information likely to be grammar-specific (auxiliaries, pre-infinitival "to", etc.) and for computing a simplified score (Black et al., 1991). The canonical implementation of the PARSEVAL metrics is called **evalb** (Sekine and Collins, 1997).

evalb

Nonetheless, phrasal constituents are not always an appropriate unit for parser evaluation. In lexically-oriented grammars, such as CCG and LFG, the ultimate goal is to extract the appropriate predicate-argument relations or grammatical dependencies, rather than a specific derivation. Such relations are also more directly relevant to further semantic processing. For these purposes, we can use alternative evaluation metrics based on measuring the precision and recall of labeled dependencies, where the labels indicate the grammatical relations (Lin 1995, Carroll et al. 1998, Collins et al. 1999).

Finally, you might wonder why we don't evaluate parsers by measuring how many *sentences* are parsed correctly instead of measuring *component* accuracy in the form of constituents or dependencies. The reason we use components is that it gives us a more fine-grained metric. This is especially true for long sentences, where most parsers don't get a perfect parse. If we just measured sentence accuracy, we wouldn't be able to distinguish between a parse that got most of the parts wrong and one that just got one part wrong.

12.9 Human Parsing

Human sentence processing

Are the kinds of probabilistic parsing models we have been discussing also used by humans when they are parsing? The answer to this question lies in a field called **human sentence processing**. Recent studies suggest that there are at least two ways in which humans apply probabilistic parsing algorithms, although there is still disagreement on the details.

Reading time

One family of studies has shown that when humans read, the predictability of a word seems to influence the **reading time**; more predictable words are read more quickly. One way of defining predictability is from simple bigram measures. For example, Scott and Shillcock (2003) used an eye-tracker to monitor the gaze of participants reading sentences. They constructed the sentences so that some would have a verb-noun pair with a high bigram probability (such as (12.45a)) and others a verb-noun pair with a low bigram probability (such as (12.45b)).

(12.45) a) **HIGH PROB:** One way to **avoid confusion** is to make the changes during vacation
b) **LOW PROB:** One way to **avoid discovery** is to make the changes during vacation

They found that the higher the bigram predictability of a word, the shorter the time that participants looked at the word (the **initial-fixation duration**).

While this result provides evidence only for N-gram probabilities, more recent experiments have suggested that the probability of an upcoming word given the syntactic parse of the preceding sentence prefix also predicts word reading time (Hale 2001, Levy 2008).

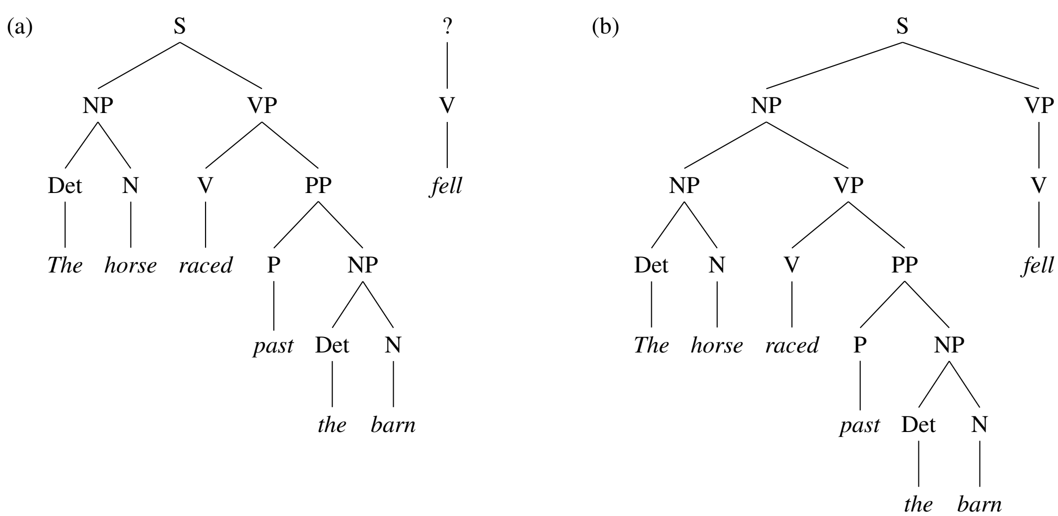
garden-path

The second family of studies has examined how humans disambiguate sentences that have multiple possible parses, suggesting that humans prefer whichever parse is more probable. These studies often rely on a specific class of temporarily ambiguous sentences called **garden-path** sentences. These sentences, first described by Bever (1970), are sentences that are cleverly constructed to have three properties that combine to make them very difficult for people to parse:

1. They are **temporarily ambiguous**: The sentence is unambiguous, but its initial portion is ambiguous.
2. One of the two or more parses in the initial portion is somehow preferable to the human parsing mechanism.
3. But the dispreferred parse is the correct one for the sentence.

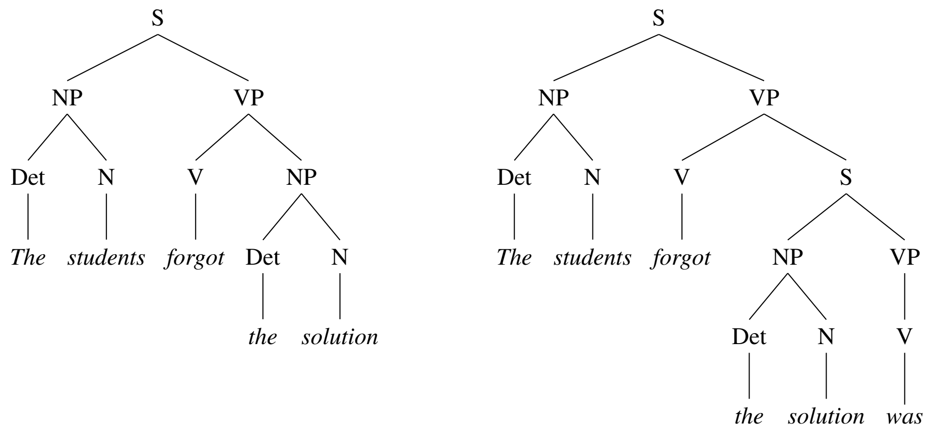
The result of these three properties is that people are "led down the garden path" toward the incorrect parse and then are confused when they realize it's the wrong one. Sometimes this confusion is quite conscious, as in Bever's example (12.46); in fact, this sentence is so hard to parse that readers often need to be shown the correct structure. In the correct structure, *raced* is part of a reduced relative clause modifying *The horse*, and means "The horse [which was raced past the barn] fell"; this structure is also present in the sentence "Students taught by the Berlitz method do worse when they get to France".

(12.46) The horse raced past the barn fell.



Other times, the confusion caused by a garden-path sentence is so subtle that it can only be measured by a slight increase in reading time. Thus, in (12.47) readers often misparse *the solution* as the direct object of *forgot* rather than as the subject of an embedded sentence. This misparse is subtle, and is only noticeable because experimental participants take longer to read the word *was* than in control sentences. This "mini garden path" effect at the word *was* suggests that subjects had chosen the direct object parse and had to reanalyze or rearrange their parse now that they realize they are in a sentential complement.

(12.47) The student forgot the solution was in the back of the book.



While many factors seem to play a role in these preferences for a particular (incorrect) parse, at least one factor seems to be syntactic probabilities, especially lexicalized (subcategorization) probabilities. For example, the probability of the verb *forgot* taking a direct object ($VP \rightarrow V\ NP$) is higher than the probability of it taking a sentential complement ($VP \rightarrow V\ S$); this difference causes readers to expect a direct object after *forget* and be surprised (longer reading times) when they encounter a sentential complement. By contrast, a verb which prefers a sentential complement (like *hope*) didn't cause extra reading time at *was*. The garden path in (12.46) is at least partially caused by the low probability of the reduced relative clause construction.

12.10 Summary

This chapter has sketched the basics of **probabilistic** parsing, concentrating on **probabilistic context-free grammars** and **probabilistic lexicalized context-free grammars**.

- Probabilistic grammars assign a probability to a sentence or string of words while attempting to capture more sophisticated syntactic information than the *N*-gram grammars of Chapter 3.
- A **probabilistic context-free grammar** (**PCFG**) is a context-free grammar in which every rule is annotated with the probability of that rule being chosen. Each PCFG rule is treated as if it were **conditionally independent**; thus, the probability of a sentence is computed by **multiplying** the probabilities of each rule in the parse of the sentence.
- The probabilistic CKY (**Cocke-Kasami-Younger**) algorithm is a probabilistic version of the CKY parsing algorithm. There are also probabilistic versions of other parsers like the Earley algorithm.
- PCFG probabilities can be learned by counting in a **parsed corpus** or by parsing a corpus. The **inside-outside** algorithm is a way of dealing with the fact that the sentences being parsed are ambiguous.
- Raw PCFGs suffer from poor independence assumptions among rules and lack of sensitivity to lexical dependencies.
- One way to deal with this problem is to split and merge non-terminals (automatically or by hand).

- **Probabilistic lexicalized CFG**s are another solution to this problem in which the basic PCFG model is augmented with a **lexical head** for each rule. The probability of a rule can then be conditioned on the lexical head or nearby heads.
- Parsers for lexicalized PCFGs (like the Charniak and Collins parsers) are based on extensions to probabilistic CKY parsing.
- Parsers are evaluated with three metrics: **labeled recall**, **labeled precision**, and **cross-brackets**.
- Evidence from **garden-path sentences** and other on-line sentence-processing experiments suggest that the human parser uses some kinds of probabilistic information about grammar.

Bibliographical and Historical Notes

Many of the formal properties of probabilistic context-free grammars were first worked out by Booth (1969) and Salomaa (1969). Baker (1979) proposed the inside-outside algorithm for unsupervised training of PCFG probabilities, and used a CKY-style parsing algorithm to compute inside probabilities. Jelinek and Lafferty (1991) extended the CKY algorithm to compute probabilities for prefixes. Stolcke (1995) drew on both of these algorithms in adapting the Earley algorithm to use with PCFGs.

A number of researchers starting in the early 1990s worked on adding lexical dependencies to PCFGs and on making PCFG rule probabilities more sensitive to surrounding syntactic structure. For example, Schabes et al. (1988) and Schabes (1990) presented early work on the use of heads. Many papers on the use of lexical dependencies were first presented at the DARPA Speech and Natural Language Workshop in June 1990. A paper by Hindle and Rooth (1990) applied lexical dependencies to the problem of attaching prepositional phrases; in the question session to a later paper, Ken Church suggested applying this method to full parsing (Marcus, 1990). Early work on such probabilistic CFG parsing augmented with probabilistic dependency information includes Magerman and Marcus (1991), Black et al. (1992), Bod (1993), and Jelinek et al. (1994), in addition to Collins (1996), Charniak (1997), and Collins (1999) discussed above. Other recent PCFG parsing models include Klein and Manning (2003a) and Petrov et al. (2006).

This early lexical probabilistic work led initially to work focused on solving specific parsing problems like preposition-phrase attachment by using methods including transformation-based learning (TBL) (Brill and Resnik, 1994), maximum entropy (Ratnaparkhi et al., 1994), memory-based Learning (Zavrel and Daelemans, 1997), log-linear models (Franz, 1997), decision trees that used semantic distance between heads (computed from WordNet) (Stetina and Nagao, 1997), and boosting (Abney et al., 1999).

Another direction extended the lexical probabilistic parsing work to build probabilistic formulations of grammars other than PCFGs, such as probabilistic TAG grammar (Resnik 1992, Schabes 1992), based on the TAG grammars discussed in Chapter 10, probabilistic LR parsing (Briscoe and Carroll, 1993), and probabilistic link grammar (Lafferty et al., 1992). An approach to probabilistic parsing called **supertagging** extends the part-of-speech tagging metaphor to parsing by using very complex tags that are, in fact, fragments of lexicalized parse trees (Bangalore and

supertagging

Joshi 1999, Joshi and Srinivas 1994), based on the lexicalized TAG grammars of Schabes et al. (1988). For example, the noun *purchase* would have a different tag as the first noun in a noun compound (where it might be on the left of a small tree dominated by Nominal) than as the second noun (where it might be on the right).

Goodman (1997), Abney (1997), and Johnson et al. (1999) gave early discussions of probabilistic treatments of feature-based grammars. Other recent work on building statistical models of feature-based grammar formalisms like HPSG and LFG includes (Riezler et al. 2002, Kaplan et al. 2004), and Toutanova et al. (2005).

We mentioned earlier that discriminative approaches to parsing fall into the two broad categories of dynamic programming methods and discriminative reranking methods. Recall that discriminative reranking approaches require *N*-best parses. Parsers based on A* search can easily be modified to generate *N*-best lists just by continuing the search past the first-best parse (Roark, 2001). Dynamic programming algorithms like the ones described in this chapter can be modified by the elimination of the dynamic programming with heavy pruning (Collins 2000, Collins and Koo 2005, Bikel 2004), or through new algorithms (Jiménez and Marzal 2000,Charniak and Johnson 2005,Huang and Chiang 2005), some adapted from speech recognition algorithms such as those of Schwartz and Chow (1990) (see Section **??**).

In dynamic programming methods, instead of outputting and then reranking an *N*-best list, the parses are represented compactly in a chart, and log-linear and other methods are applied for decoding directly from the chart. Such modern methods include (Johnson 2001, Clark and Curran 2004), and Taskar et al. (2004). Other reranking developments include changing the optimization criterion (Titov and Henderson, 2006).

Collins' (1999) dissertation includes a very readable survey of the field and an introduction to his parser. Manning and Schütze (1999) extensively cover probabilistic parsing.

The field of grammar induction is closely related to statistical parsing, and a parser is often used as part of a grammar induction algorithm. One of the earliest statistical works in grammar induction was Horning (1969), who showed that PCFGs could be induced without negative evidence. Early modern probabilistic grammar work showed that simply using EM was insufficient (Lari and Young 1990, Carroll and Charniak 1992). Recent probabilistic work, such as Yuret (1998), Clark (2001), Klein and Manning (2002), and Klein and Manning (2004), are summarized in Klein (2005) and Adriaans and van Zaanen (2004). Work since that summary includes Smith and Eisner (2005), Haghighi and Klein (2006), and Smith and Eisner (2007).

Exercises

12.1 Implement the CKY algorithm.

12.2 Modify the algorithm for conversion to CNF from Chapter 11 to correctly handle rule probabilities. Make sure that the resulting CNF assigns the same total probability to each parse tree.

12.3 Recall that Exercise 13.3 asked you to update the CKY algorithm to handle unit productions directly rather than converting them to CNF. Extend this change to probabilistic CKY.

12.4 Fill out the rest of the probabilistic CKY chart in Fig. 12.4.

12.5 Sketch how the CKY algorithm would have to be augmented to handle lexicalized probabilities.

12.6 Implement your lexicalized extension of the CKY algorithm.

12.7 Implement the PARSEVAL metrics described in Section 12.8. Next, either use a treebank or create your own hand-checked parsed test set. Now use your CFG (or other) parser and grammar, parse the test set and compute labeled recall, labeled precision, and cross-brackets.

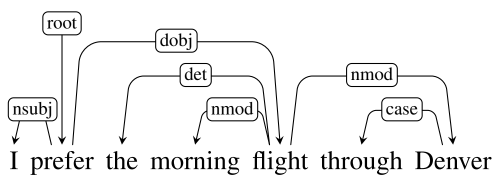
CHAPTER

13 Dependency Parsing

dependency grammars

The focus of the three previous chapters has been on context-free grammars and their use in automatically generating constituent-based representations. Here we present another family of grammar formalisms called **dependency grammars** that are quite important in contemporary speech and language processing systems. In these formalisms, phrasal constituents and phrase-structure rules do not play a direct role. Instead, the syntactic structure of a sentence is described solely in terms of the words (or lemmas) in a sentence and an associated set of directed binary grammatical relations that hold among the words.

The following diagram illustrates a dependency-style analysis using the standard graphical method favored in the dependency-parsing community.



(13.1)

typed dependency

Relations among the words are illustrated above the sentence with directed, labeled arcs from heads to dependents. We call this a **typed dependency structure** because the labels are drawn from a fixed inventory of grammatical relations. It also includes a *root* node that explicitly marks the root of the tree, the head of the entire structure.

Figure 13.1 shows the same dependency analysis as a tree alongside its corresponding phrase-structure analysis of the kind given in Chapter 10. Note the absence of nodes corresponding to phrasal constituents or lexical categories in the dependency parse; the internal structure of the dependency parse consists solely of directed relations between lexical items in the sentence. These relationships directly encode important information that is often buried in the more complex phrase-structure parses. For example, the arguments to the verb *prefer* are directly linked to it in the dependency structure, while their connection to the main verb is more distant in the phrase-structure tree. Similarly, *morning* and *Denver*, modifiers of *flight*, are linked to it directly in the dependency structure.

free word order

A major advantage of dependency grammars is their ability to deal with languages that are morphologically rich and have a relatively **free word order**. For example, word order in Czech can be much more flexible than in English; a grammatical *object* might occur before or after a *location adverbial*. A phrase-structure grammar would need a separate rule for each possible place in the parse tree where such an adverbial phrase could occur. A dependency-based approach would just have one link type representing this particular adverbial relation. Thus, a dependency grammar approach abstracts away from word-order information, representing only the information that is necessary for the parse.

An additional practical motivation for a dependency-based approach is that the head-dependent relations provide an approximation to the semantic relationship be-

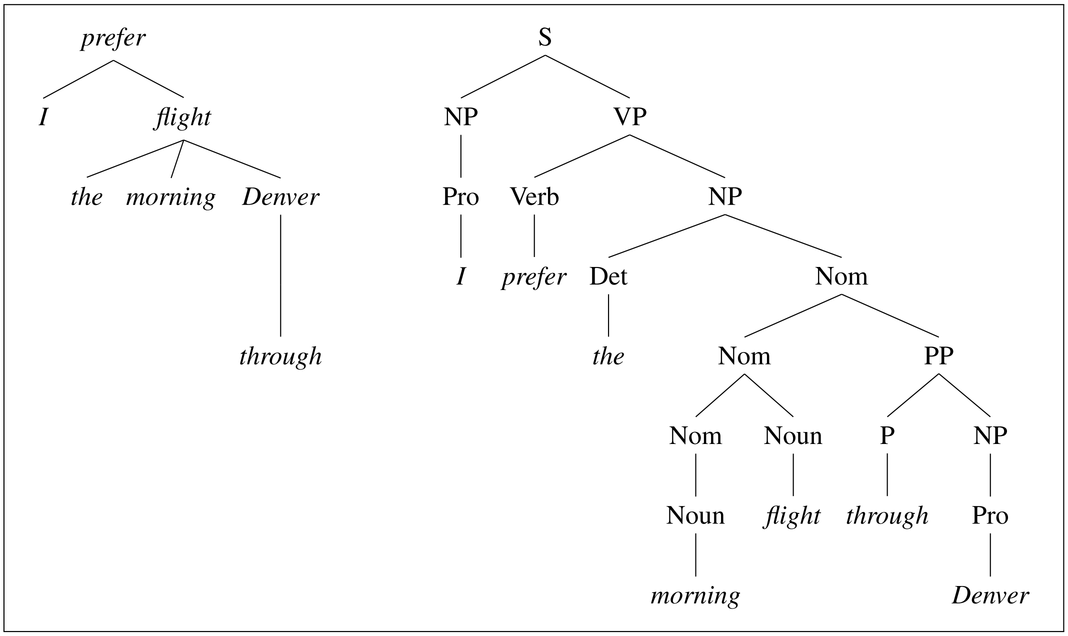


Figure 13.1 A dependency-style parse alongside the corresponding constituent-based analysis for *I prefer the morning flight through Denver.*

tween predicates and their arguments that makes them directly useful for many applications such as coreference resolution, question answering and information extraction. Constituent-based approaches to parsing provide similar information, but it often has to be distilled from the trees via techniques such as the head finding rules discussed in Chapter 10.

In the following sections, we'll discuss in more detail the inventory of relations used in dependency parsing, as well as the formal basis for these dependency structures. We'll then move on to discuss the dominant families of algorithms that are used to automatically produce these structures. Finally, we'll discuss how to evaluate dependency parsers and point to some of the ways they are used in language processing applications.

13.1 Dependency Relations

grammatical relation

head

dependent

The traditional linguistic notion of **grammatical relation** provides the basis for the binary relations that comprise these dependency structures. The arguments to these relations consist of a **head** and a **dependent**. We've already discussed the notion of heads in Chapter 10 and Chapter 12 in the context of constituent structures. There, the head word of a constituent was the central organizing word of a larger constituent (e.g, the primary noun in a noun phrase, or verb in a verb phrase). The remaining words in the constituent are either direct, or indirect, dependents of their head. In dependency-based approaches, the head-dependent relationship is made explicit by directly linking heads to the words that are immediately dependent on them, bypassing the need for constituent structures.

grammatical function

In addition to specifying the head-dependent pairs, dependency grammars allow us to further classify the kinds of grammatical relations, or **grammatical function**,

Clausal Argument Relations	**Description**
NSUBJ	Nominal subject
DOBJ	Direct object
IOBJ	Indirect object
CCOMP	Clausal complement
XCOMP	Open clausal complement
Nominal Modifier Relations	**Description**
NMOD	Nominal modifier
AMOD	Adjectival modifier
NUMMOD	Numeric modifier
APPOS	Appositional modifier
DET	Determiner
CASE	Prepositions, postpositions and other case markers
Other Notable Relations	**Description**
CONJ	Conjunct
CC	Coordinating conjunction

Figure 13.2 Selected dependency relations from the Universal Dependency set. (de Marneffe et al., 2014)

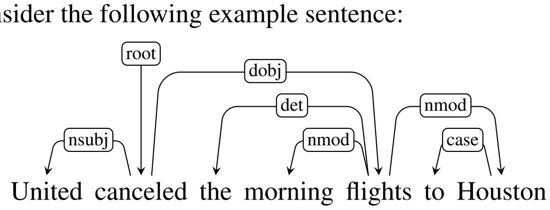
in terms of the role that the dependent plays with respect to its head. Familiar notions such as *subject*, *direct object* and *indirect object* are among the kind of relations we have in mind. In English these notions strongly correlate with, but by no means determine, both position in a sentence and constituent type and are therefore somewhat redundant with the kind of information found in phrase-structure trees. However, in more flexible languages the information encoded directly in these grammatical relations is critical since phrase-based constituent syntax provides little help.

Not surprisingly, linguists have developed taxonomies of relations that go well beyond the familiar notions of subject and object. While there is considerable variation from theory to theory, there is enough commonality that efforts to develop a computationally useful standard are now possible. The **Universal Dependencies** project (Nivre et al., 2016b) provides an inventory of dependency relations that are linguistically motivated, computationally useful, and cross-linguistically applicable. Fig. 13.2 shows a subset of the relations from this effort. Fig. 13.3 provides some example sentences illustrating selected relations.

Universal Dependencies

The motivation for all of the relations in the Universal Dependency scheme is beyond the scope of this chapter, but the core set of frequently used relations can be broken into two sets: clausal relations that describe syntactic roles with respect to a predicate (often a verb), and modifier relations that categorize the ways that words that can modify their heads.

Consider the following example sentence:



(13.2)

The clausal relations NSUBJ and DOBJ identify the subject and direct object of the predicate *cancel*, while the NMOD, DET, and CASE relations denote modifiers of the nouns *flights* and *Houston*.

Relation	Examples with *head* and **dependent**
NSUBJ	**United** *canceled* the flight.
DOBJ	United *diverted* the **flight** to Reno.
	We *booked* her the first **flight** to Miami.
IOBJ	We *booked* **her** the flight to Miami.
NMOD	We took the **morning** *flight*.
AMOD	Book the **cheapest** *flight*.
NUMMOD	Before the storm JetBlue canceled **1000** *flights*.
APPOS	*United*, a **unit** of UAL, matched the fares.
DET	**The** *flight* was canceled.
	Which *flight* was delayed?
CONJ	We *flew* to Denver and **drove** to Steamboat.
CC	We flew to Denver **and** *drove* to Steamboat.
CASE	Book the flight **through** *Houston*.

Figure 13.3 Examples of core Universal Dependency relations.

13.2 Dependency Formalisms

In their most general form, the dependency structures we're discussing are simply directed graphs. That is, structures $G = (V, A)$ consisting of a set of vertices V, and a set of ordered pairs of vertices A, which we'll refer to as arcs.

For the most part we will assume that the set of vertices, V, corresponds exactly to the set of words in a given sentence. However, they might also correspond to punctuation, or when dealing with morphologically complex languages the set of vertices might consist of stems and affixes. The set of arcs, A, captures the head-dependent and grammatical function relationships between the elements in V.

Further constraints on these dependency structures are specific to the underlying grammatical theory or formalism. Among the more frequent restrictions are that the structures must be connected, have a designated root node, and be acyclic or planar. Of most relevance to the parsing approaches discussed in this chapter is the common, computationally-motivated, restriction to rooted trees. That is, a **dependency tree** is a directed graph that satisfies the following constraints:

dependency tree

1. There is a single designated root node that has no incoming arcs.
2. With the exception of the root node, each vertex has exactly one incoming arc.
3. There is a unique path from the root node to each vertex in V.

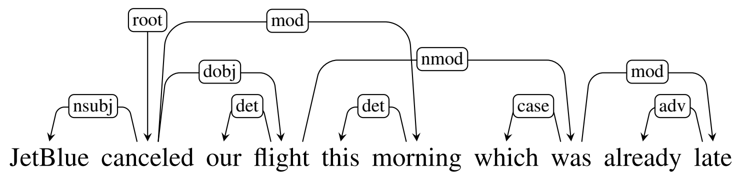
Taken together, these constraints ensure that each word has a single head, that the dependency structure is connected, and that there is a single root node from which one can follow a unique directed path to each of the words in the sentence.

13.2.1 Projectivity

The notion of projectivity imposes an additional constraint that is derived from the order of the words in the input, and is closely related to the context-free nature of human languages discussed in Chapter 10. An arc from a head to a dependent is said to be projective if there is a path from the head to every word that lies between the head and the dependent in the sentence. A dependency tree is then said to be projective if all the arcs that make it up are projective. All the dependency trees we've seen thus far have been projective. There are, however, many perfectly valid

constructions which lead to non-projective trees, particularly in languages with a relatively flexible word order.

Consider the following example.



(13.3)

In this example, the arc from *flight* to its modifier *was* is non-projective since there is no path from *flight* to the intervening words *this* and *morning*. As we can see from this diagram, projectivity (and non-projectivity) can be detected in the way we've been drawing our trees. A dependency tree is projective if it can be drawn with no crossing edges. Here there is no way to link *flight* to its dependent *was* without crossing the arc that links *morning* to its head.

Our concern with projectivity arises from two related issues. First, the most widely used English dependency treebanks were automatically derived from phrase-structure treebanks through the use of head-finding rules (Chapter 10). The trees generated in such a fashion are guaranteed to be projective since they're generated from context-free grammars.

Second, there are computational limitations to the most widely used families of parsing algorithms. The transition-based approaches discussed in Section 13.4 can only produce projective trees, hence any sentences with non-projective structures will necessarily contain some errors. This limitation is one of the motivations for the more flexible graph-based parsing approach described in Section 13.5.

13.3 Dependency Treebanks

As with constituent-based methods, treebanks play a critical role in the development and evaluation of dependency parsers. Dependency treebanks have been created using similar approaches to those discussed in Chapter 10 — having human annotators directly generate dependency structures for a given corpus, or using automatic parsers to provide an initial parse and then having annotators hand correct those parsers. We can also use a deterministic process to translate existing constituent-based treebanks into dependency trees through the use of head rules.

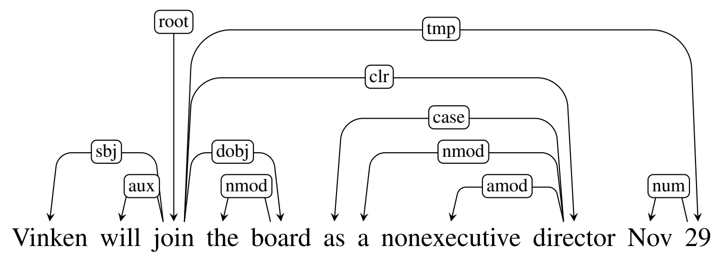
For the most part, directly annotated dependency treebanks have been created for morphologically rich languages such as Czech, Hindi and Finnish that lend themselves to dependency grammar approaches, with the Prague Dependency Treebank (Bejček et al., 2013) for Czech being the most well-known effort. The major English dependency treebanks have largely been extracted from existing resources such as the Wall Street Journal sections of the Penn Treebank(Marcus et al., 1993). The more recent OntoNotes project (Hovy et al. 2006,Weischedel et al. 2011) extends this approach going beyond traditional news text to include conversational telephone speech, weblogs, usenet newsgroups, broadcast, and talk shows in English, Chinese and Arabic.

The translation process from constituent to dependency structures has two subtasks: identifying all the head-dependent relations in the structure and identifying the correct dependency relations for these relations. The first task relies heavily on

the use of head rules discussed in Chapter 10 first developed for use in lexicalized probabilistic parsers (Magerman 1994,Collins 1999,Collins 2003b). Here's a simple and effective algorithm from Xia and Palmer (2001).

1. Mark the head child of each node in a phrase structure, using the appropriate head rules.
2. In the dependency structure, make the head of each non-head child depend on the head of the head-child.

When a phrase-structure parse contains additional information in the form of grammatical relations and function tags, as in the case of the Penn Treebank, these tags can be used to label the edges in the resulting tree. When applied to the parse tree in Fig. 13.4, this algorithm would produce the dependency structure in Fig. 13.4.



(13.4)

The primary shortcoming of these extraction methods is that they are limited by the information present in the original constituent trees. Among the most important issues are the failure to integrate morphological information with the phrase-structure trees, the inability to easily represent non-projective structures, and the lack of internal structure to most noun-phrases, as reflected in the generally flat rules used in most treebank grammars. For these reasons, outside of English, most dependency treebanks are developed directly using human annotators.

13.4 Transition-Based Dependency Parsing

shift-reduce parsing

Our first approach to dependency parsing is motivated by a stack-based approach called **shift-reduce parsing** originally developed for analyzing programming languages (Aho and Ullman, 1972). This classic approach is simple and elegant, employing a context-free grammar, a stack, and a list of tokens to be parsed. Input tokens are successively shifted onto the stack and the top two elements of the stack are matched against the right-hand side of the rules in the grammar; when a match is found the matched elements are replaced on the stack (reduced) by the non-terminal from the left-hand side of the rule being matched. In adapting this approach for dependency parsing, we forgo the explicit use of a grammar and alter the reduce operation so that instead of adding a non-terminal to a parse tree, it introduces a dependency relation between a word and its head. More specifically, the reduce action is replaced with two possible actions: assert a head-dependent relation between the word at the top of the stack and the word below it, or vice versa. Figure 13.5 illustrates the basic operation of such a parser.

configuration

A key element in transition-based parsing is the notion of a **configuration** which consists of a stack, an input buffer of words, or tokens, and a set of relations representing a dependency tree. Given this framework, the parsing process consists of a sequence of transitions through the space of possible configurations. The goal of

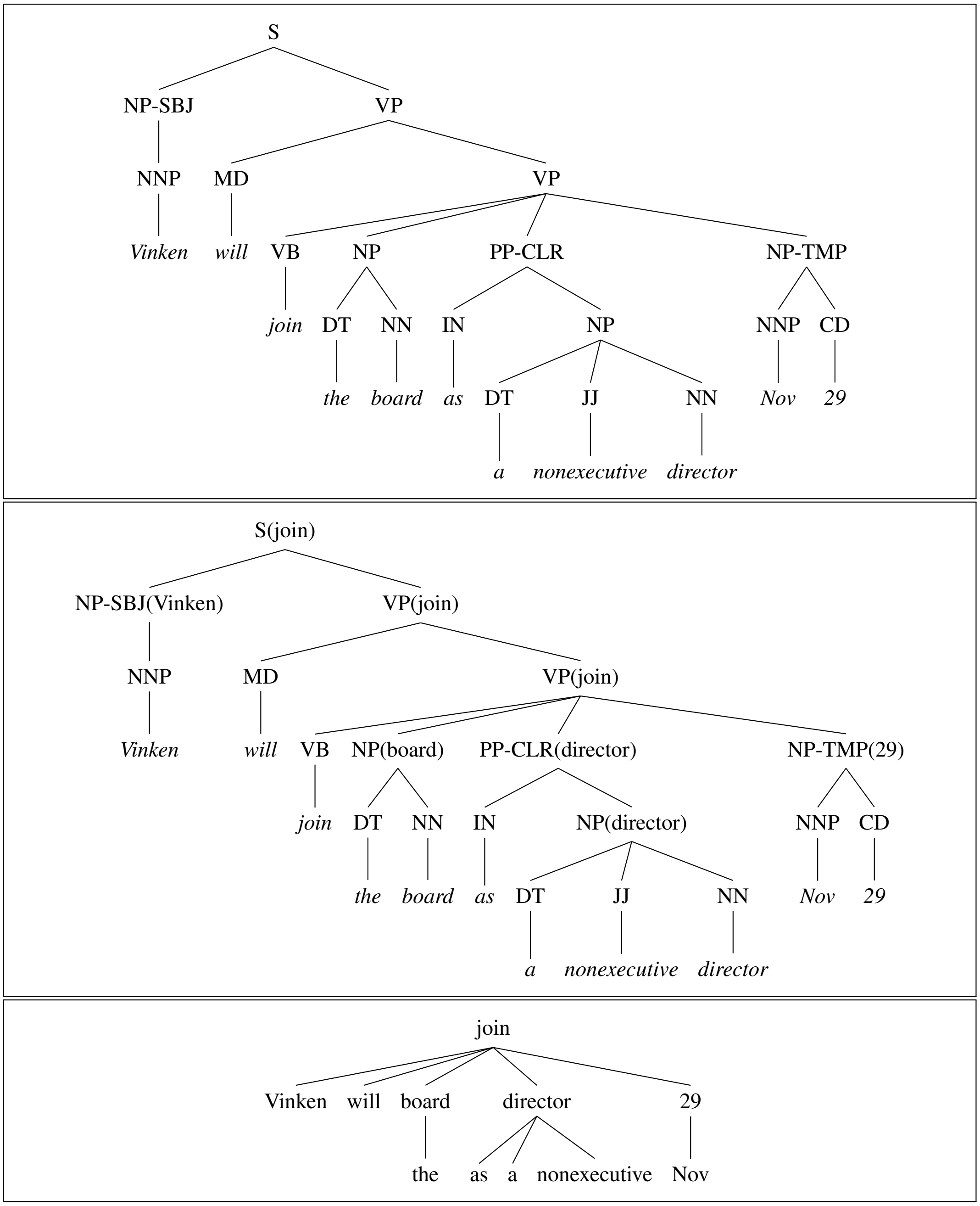


Figure 13.4 A phrase-structure tree from the *Wall Street Journal* component of the Penn Treebank 3.

this process is to find a final configuration where all the words have been accounted for and an appropriate dependency tree has been synthesized.

To implement such a search, we'll define a set of transition operators, which when applied to a configuration produce new configurations. Given this setup, we can view the operation of a parser as a search through a space of configurations for a sequence of transitions that leads from a start state to a desired goal state. At the start of this process we create an initial configuration in which the stack contains the

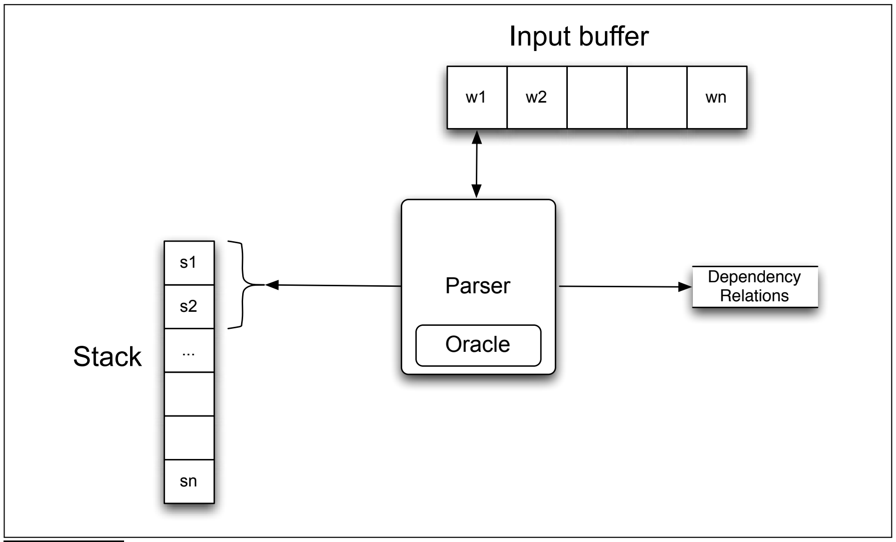


Figure 13.5 Basic transition-based parser. The parser examines the top two elements of the stack and selects an action based on consulting an oracle that examines the current configuration.

ROOT node, the word list is initialized with the set of the words or lemmatized tokens in the sentence, and an empty set of relations is created to represent the parse. In the final goal state, the stack and the word list should be empty, and the set of relations will represent the final parse.

In the standard approach to transition-based parsing, the operators used to produce new configurations are surprisingly simple and correspond to the intuitive actions one might take in creating a dependency tree by examining the words in a single pass over the input from left to right (Covington, 2001):

- Assign the current word as the head of some previously seen word,
- Assign some previously seen word as the head of the current word,
- Or postpone doing anything with the current word, adding it to a store for later processing.

To make these actions more precise, we'll create three transition operators that will operate on the top two elements of the stack:

- LEFTARC: Assert a head-dependent relation between the word at the top of stack and the word directly beneath it; remove the lower word from the stack.
- RIGHTARC: Assert a head-dependent relation between the second word on the stack and the word at the top; remove the word at the top of the stack;
- SHIFT: Remove the word from the front of the input buffer and push it onto the stack.

arc standard

This particular set of operators implements what is known as the **arc standard** approach to transition-based parsing (Covington 2001,Nivre 2003). There are two notable characteristics to this approach: the transition operators only assert relations between elements at the top of the stack, and once an element has been assigned its head it is removed from the stack and is not available for further processing. As we'll see, there are alternative transition systems which demonstrate different parsing behaviors, but the arc standard approach is quite effective and is simple to implement.

To assure that these operators are used properly we'll need to add some preconditions to their use. First, since, by definition, the ROOT node cannot have any incoming arcs, we'll add the restriction that the LEFTARC operator cannot be applied when ROOT is the second element of the stack. Second, both reduce operators require two elements to be on the stack to be applied. Given these transition operators and preconditions, the specification of a transition-based parser is quite simple. Fig. 13.6 gives the basic algorithm.

```
function DEPENDENCYPARSE(words) returns dependency tree

  state ← {[root], [words], [] }  ; initial configuration
  while state not final
    t ← ORACLE(state)        ; choose a transition operator to apply
    state ← APPLY(t, state)  ; apply it, creating a new state
  return state
```

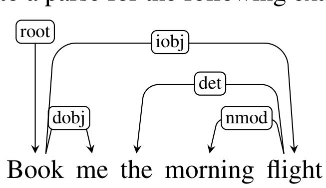
Figure 13.6 A generic transition-based dependency parser

At each step, the parser consults an oracle (we'll come back to this shortly) that provides the correct transition operator to use given the current configuration. It then applies that operator to the current configuration, producing a new configuration. The process ends when all the words in the sentence have been consumed and the ROOT node is the only element remaining on the stack.

The efficiency of transition-based parsers should be apparent from the algorithm. The complexity is linear in the length of the sentence since it is based on a single left to right pass through the words in the sentence. More specifically, each word must first be shifted onto the stack and then later reduced.

Note that unlike the dynamic programming and search-based approaches discussed in Chapters 12 and 13, this approach is a straightforward greedy algorithm — the oracle provides a single choice at each step and the parser proceeds with that choice, no other options are explored, no backtracking is employed, and a single parse is returned in the end.

Figure 13.7 illustrates the operation of the parser with the sequence of transitions leading to a parse for the following example.



(13.5)

Let's consider the state of the configuration at Step 2, after the word *me* has been pushed onto the stack.

Stack	Word List	Relations
[root, book, me]	[the, morning, flight]	

The correct operator to apply here is RIGHTARC which assigns *book* as the head of *me* and pops *me* from the stack resulting in the following configuration.

Stack	Word List	Relations
[root, book]	[the, morning, flight]	(book $\rightarrow$ me)

Step	Stack	Word List	Action	Relation Added
0	[root]	[book, me, the, morning, flight]	SHIFT	
1	[root, book]	[me, the, morning, flight]	SHIFT	
2	[root, book, me]	[the, morning, flight]	RIGHTARC	(book → me)
3	[root, book]	[the, morning, flight]	SHIFT	
4	[root, book, the]	[morning, flight]	SHIFT	
5	[root, book, the, morning]	[flight]	SHIFT	
6	[root, book, the, morning, flight]	[]	LEFTARC	(morning ← flight)
7	[root, book, the, flight]	[]	LEFTARC	(the ← flight)
8	[root, book, flight]	[]	RIGHTARC	(book → flight)
9	[root, book]	[]	RIGHTARC	(root → book)
10	[root]	[]	Done	

Figure 13.7 Trace of a transition-based parse.

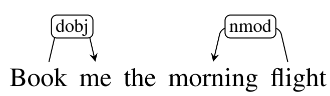
After several subsequent applications of the SHIFT and LEFTARC operators, the configuration in Step 6 looks like the following:

Stack	Word List	Relations
[root, book, the, morning, flight]	[]	(book → me)

Here, all the remaining words have been passed onto the stack and all that is left to do is to apply the appropriate reduce operators. In the current configuration, we employ the LEFTARC operator resulting in the following state.

Stack	Word List	Relations
[root, book, the, flight]	[]	(book → me)
		(morning ← flight)

At this point, the parse for this sentence consists of the following structure.

(13.6)


There are several important things to note when examining sequences such as the one in Figure 13.7. First, the sequence given is not the only one that might lead to a reasonable parse. In general, there may be more than one path that leads to the same result, and due to ambiguity, there may be other transition sequences that lead to different equally valid parses.

Second, we are assuming that the oracle always provides the correct operator at each point in the parse — an assumption that is unlikely to be true in practice. As a result, given the greedy nature of this algorithm, incorrect choices will lead to incorrect parses since the parser has no opportunity to go back and pursue alternative choices. Section 13.4.2 will introduce several techniques that allow transition-based approaches to explore the search space more fully.

Finally, for simplicity, we have illustrated this example without the labels on the dependency relations. To produce labeled trees, we can parameterize the LEFTARC and RIGHTARC operators with dependency labels, as in LEFTARC(NSUBJ) or RIGHTARC(DOBJ). This is equivalent to expanding the set of transition operators from our original set of three to a set that includes LEFTARC and RIGHTARC operators for each relation in the set of dependency relations being used, plus an additional one for the SHIFT operator. This, of course, makes the job of the oracle more difficult since it now has a much larger set of operators from which to choose.

13.4.1 Creating an Oracle

State-of-the-art transition-based systems use supervised machine learning methods to train classifiers that play the role of the oracle. Given appropriate training data, these methods learn a function that maps from configurations to transition operators.

As with all supervised machine learning methods, we will need access to appropriate training data and we will need to extract features useful for characterizing the decisions to be made. The source for this training data will be representative treebanks containing dependency trees. The features will consist of many of the same features we encountered in Chapter 8 for part-of-speech tagging, as well as those used in Chapter 12 for statistical parsing models.

Generating Training Data

Let's revisit the oracle from the algorithm in Fig. 13.6 to fully understand the learning problem. The oracle takes as input a configuration and returns as output a transition operator. Therefore, to train a classifier, we will need configurations paired with transition operators (i.e., LEFTARC, RIGHTARC, or SHIFT). Unfortunately, treebanks pair entire sentences with their corresponding trees, and therefore they don't directly provide what we need.

To generate the required training data, we will employ the oracle-based parsing algorithm in a clever way. We will supply our oracle with the training sentences to be parsed *along with* their corresponding reference parses from the treebank. To produce training instances, we will then *simulate* the operation of the parser by running the algorithm and relying on a new **training oracle** to give us correct transition operators for each successive configuration.

training oracle

To see how this works, let's first review the operation of our parser. It begins with a default initial configuration where the stack contains the ROOT, the input list is just the list of words, and the set of relations is empty. The LEFTARC and RIGHTARC operators each add relations between the words at the top of the stack to the set of relations being accumulated for a given sentence. Since we have a gold-standard reference parse for each training sentence, we know which dependency relations are valid for a given sentence. Therefore, we can use the reference parse to guide the selection of operators as the parser steps through a sequence of configurations.

To be more precise, given a reference parse and a configuration, the training oracle proceeds as follows:

- Choose LEFTARC if it produces a correct head-dependent relation given the reference parse and the current configuration,
- Otherwise, choose RIGHTARC if (1) it produces a correct head-dependent relation given the reference parse and (2) all of the dependents of the word at the top of the stack have already been assigned,
- Otherwise, choose SHIFT.

The restriction on selecting the RIGHTARC operator is needed to ensure that a word is not popped from the stack, and thus lost to further processing, before all its dependents have been assigned to it.

More formally, during training the oracle has access to the following information:

- A current configuration with a stack S and a set of dependency relations R_c
- A reference parse consisting of a set of vertices V and a set of dependency relations R_p

Step	Stack	Word List	Predicted Action
0	[root]	[book, the, flight, through, houston]	SHIFT
1	[root, book]	[the, flight, through, houston]	SHIFT
2	[root, book, the]	[flight, through, houston]	SHIFT
3	[root, book, the, flight]	[through, houston]	LEFTARC
4	[root, book, flight]	[through, houston]	SHIFT
5	[root, book, flight, through]	[houston]	SHIFT
6	[root, book, flight, through, houston]	[]	LEFTARC
7	[root, book, flight, houston]	[]	RIGHTARC
8	[root, book, flight]	[]	RIGHTARC
9	[root, book]	[]	RIGHTARC
10	[root]	[]	Done

Figure 13.8 Generating training items consisting of configuration/predicted action pairs by simulating a parse with a given reference parse.

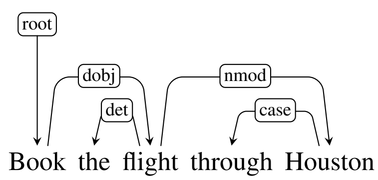
Given this information, the oracle chooses transitions as follows:

LEFTARC(r): **if** $(S_1\ r\ S_2) \in R_p$

RIGHTARC(r): **if** $(S_2\ r\ S_1) \in R_p$ **and** $\forall r', w\ s.t.(S_1\ r'\ w) \in R_p$ **then** $(S_1\ r'\ w) \in R_c$

SHIFT: **otherwise**

Let's walk through some the steps of this process with the following example as shown in Fig. 13.8.



(13.7)

At Step 1, LEFTARC is not applicable in the initial configuration since it asserts a relation, (root ← book), not in the reference answer; RIGHTARC does assert a relation contained in the final answer (root → book), however *book* has not been attached to any of its dependents yet, so we have to defer, leaving SHIFT as the only possible action. The same conditions hold in the next two steps. In step 3, LEFTARC is selected to link *the* to its head.

Now consider the situation in Step 4.

Stack	Word buffer	Relations
[root, book, flight]	[through, Houston]	(the ← flight)

Here, we might be tempted to add a dependency relation between *book* and *flight*, which is present in the reference parse. But doing so now would prevent the later attachment of *Houston* since *flight* would have been removed from the stack. Fortunately, the precondition on choosing RIGHTARC prevents this choice and we're again left with SHIFT as the only viable option. The remaining choices complete the set of operators needed for this example.

To recap, we derive appropriate training instances consisting of configuration-transition pairs from a treebank by simulating the operation of a parser in the context of a reference dependency tree. We can deterministically record correct parser actions at each step as we progress through each training example, thereby creating the training set we require.

Features

Having generated appropriate training instances (configuration-transition pairs), we need to extract useful features from the configurations so what we can train classifiers. The features that are used to train transition-based systems vary by language, genre, and the kind of classifier being employed. For example, morphosyntactic features such as case marking on subjects or direct objects may be more or less important depending on the language being processed. That said, the basic features that we have already seen with part-of-speech tagging and partial parsing have proven to be useful in training dependency parsers across a wide range of languages. Word forms, lemmas and parts of speech are all powerful features, as are the head, and dependency relation to the head.

In the transition-based parsing framework, such features need to be extracted from the configurations that make up the training data. Recall that configurations consist of three elements: the stack, the buffer and the current set of relations. In principle, any property of any or all of these elements can be represented as features in the usual way for training. However, to avoid sparsity and encourage generalization, it is best to focus the learning algorithm on the most useful aspects of decision making at each point in the parsing process. The focus of feature extraction for transition-based parsing is, therefore, on the top levels of the stack, the words near the front of the buffer, and the dependency relations already associated with any of those elements.

feature template

By combining simple features, such as word forms or parts of speech, with specific locations in a configuration, we can employ the notion of a **feature template** that we've already encountered with sentiment analysis and part-of-speech tagging. Feature templates allow us to automatically generate large numbers of specific features from a training set. As an example, consider the following feature templates that are based on single positions in a configuration.

$$\begin{aligned} &\langle s_1.w, op\rangle, \langle s_2.w, op\rangle \langle s_1.t, op\rangle, \langle s_2.t, op\rangle \\ &\langle b_1.w, op\rangle, \langle b_1.t, op\rangle \langle s_1.wt, op\rangle \end{aligned} \tag{13.8}$$

In these examples, individual features are denoted as *location.property*, where s denotes the stack, b the word buffer, and r the set of relations. Individual properties of locations include w for word forms, l for lemmas, and t for part-of-speech. For example, the feature corresponding to the word form at the top of the stack would be denoted as $s_1.w$, and the part of speech tag at the front of the buffer $b_1.t$. We can also combine individual features via concatenation into more specific features that may prove useful. For example, the feature designated by $s_1.wt$ represents the word form concatenated with the part of speech of the word at the top of the stack. Finally, op stands for the transition operator for the training example in question (i.e., the label for the training instance).

Let's consider the simple set of single-element feature templates given above in the context of the following intermediate configuration derived from a training oracle for Example 13.2.

Stack	Word buffer	Relations
[root, canceled, flights]	[to Houston]	(canceled → United)
		(flights → morning)
		(flights → the)

The correct transition here is SHIFT (you should convince yourself of this before

proceeding). The application of our set of feature templates to this configuration would result in the following set of instantiated features.

$$
\begin{aligned}
&\langle s_1.w = \textit{flights}, op = \textit{shift}\rangle \\
&\langle s_2.w = \textit{canceled}, op = \textit{shift}\rangle \\
&\langle s_1.t = \textit{NNS}, op = \textit{shift}\rangle \\
&\langle s_2.t = \textit{VBD}, op = \textit{shift}\rangle \\
&\langle b_1.w = \textit{to}, op = \textit{shift}\rangle \\
&\langle b_1.t = \textit{TO}, op = \textit{shift}\rangle \\
&\langle s_1.wt = \textit{flightsNNS}, op = \textit{shift}\rangle
\end{aligned}
\tag{13.9}
$$

Given that the left and right arc transitions operate on the top two elements of the stack, features that *combine* properties from these positions are even more useful. For example, a feature like $s_1.t \circ s_2.t$ concatenates the part of speech tag of the word at the top of the stack with the tag of the word beneath it.

$$
\langle s_1.t \circ s_2.t = \textit{NNSVBD}, op = \textit{shift}\rangle \tag{13.10}
$$

Not surprisingly, if two properties are useful then three or more should be even better. Figure 13.9 gives a baseline set of feature templates that have been employed in various state-of-the-art systems (Zhang and Clark 2008,Huang and Sagae 2010,Zhang and Nivre 2011).

Note that some of these features make use of *dynamic* features — features such as head words and dependency relations that have been predicted at earlier steps in the parsing process, as opposed to features that are derived from static properties of the input.

Source	Feature templates		
One word	$s_1.w$	$s_1.t$	$s_1.wt$
	$s_2.w$	$s_2.t$	$s_2.wt$
	$b_1.w$	$b_1.w$	$b_0.wt$
Two word	$s_1.w \circ s_2.w$	$s_1.t \circ s_2.t$	$s_1.t \circ b_1.w$
	$s_1.t \circ s_2.wt$	$s_1.w \circ s_2.w \circ s_2.t$	$s_1.w \circ s_1.t \circ s_2.t$
	$s_1.w \circ s_1.t \circ s_2.t$	$s_1.w \circ s_1.t$	

Figure 13.9 Standard feature templates for training transition-based dependency parsers. In the template specifications s_n refers to a location on the stack, b_n refers to a location in the word buffer, w refers to the wordform of the input, and t refers to the part of speech of the input.

Learning

Over the years, the dominant approaches to training transition-based dependency parsers have been multinomial logistic regression and support vector machines, both of which can make effective use of large numbers of sparse features of the kind described in the last section. More recently, neural network, or deep learning, approaches of the kind described in Chapter 8 have been applied successfully to transition-based parsing (Chen and Manning, 2014). These approaches eliminate the need for complex, hand-crafted features and have been particularly effective at overcoming the data sparsity issues normally associated with training transition-based parsers.

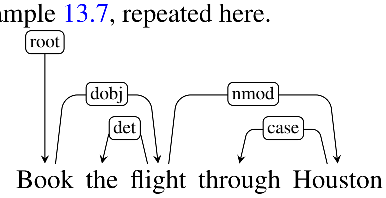
13.4.2 Advanced Methods in Transition-Based Parsing

The basic transition-based approach can be elaborated in a number of ways of ways to improve performance by addressing some of the most obvious flaws in the approach.

Alternative Transition Systems

arc eager

The arc-standard transition system described above is only one of many possible systems. A frequently used alternative is the **arc eager** transition system. The arc eager approach gets its name from its ability to assert rightward relations much sooner than in the arc standard approach. To see this, let's revisit the arc standard trace of Example 13.7, repeated here.



Consider the dependency relation between *book* and *flight* in this analysis. As is shown in Fig. 13.8, an arc-standard approach would assert this relation at Step 8, despite the fact that *book* and *flight* first come together on the stack much earlier at Step 4. The reason this relation can't be captured at this point is due to the presence of the post-nominal modifier *through Houston*. In an arc-standard approach, dependents are removed from the stack as soon as they are assigned their heads. If *flight* had been assigned *book* as its head in Step 4, it would no longer be available to serve as the head of *Houston*.

While this delay doesn't cause any issues in this example, in general the longer a word has to wait to get assigned its head the more opportunities there are for something to go awry. The arc-eager system addresses this issue by allowing words to be attached to their heads as early as possible, before all the subsequent words dependent on them have been seen. This is accomplished through minor changes to the LEFTARC and RIGHTARC operators and the addition of a new REDUCE operator.

- LEFTARC: Assert a head-dependent relation between the word at the front of the input buffer and the word at the top of the stack; pop the stack.
- RIGHTARC: Assert a head-dependent relation between the word on the top of the stack and the word at front of the input buffer; shift the word at the front of the input buffer to the stack.
- SHIFT: Remove the word from the front of the input buffer and push it onto the stack.
- REDUCE: Pop the stack.

The LEFTARC and RIGHTARC operators are applied to the top of the stack and the front of the input buffer, instead of the top two elements of the stack as in the arc-standard approach. The RIGHTARC operator now moves the dependent to the stack from the buffer rather than removing it, thus making it available to serve as the head of following words. The new REDUCE operator removes the top element from the stack. Together these changes permit a word to be eagerly assigned its head and still allow it to serve as the head for later dependents. The trace shown in Fig. 13.10 illustrates the new decision sequence for this example.

In addition to demonstrating the arc-eager transition system, this example demonstrates the power and flexibility of the overall transition-based approach. We were

Step	Stack	Word List	Action	Relation Added
0	[root]	[book, the, flight, through, houston]	RIGHTARC	(root → book)
1	[root, book]	[the, flight, through, houston]	SHIFT	
2	[root, book, the]	[flight, through, houston]	LEFTARC	(the ← flight)
3	[root, book]	[flight, through, houston]	RIGHTARC	(book → flight)
4	[root, book, flight]	[through, houston]	SHIFT	
5	[root, book, flight, through]	[houston]	LEFTARC	(through ← houston)
6	[root, book, flight]	[houston]	RIGHTARC	(flight → houston)
7	[root, book, flight, houston]	[]	REDUCE	
8	[root, book, flight]	[]	REDUCE	
9	[root, book]	[]	REDUCE	
10	[root]	[]	Done	

Figure 13.10 A processing trace of *Book the flight through Houston* using the arc-eager transition operators.

able to swap in a new transition system without having to make any changes to the underlying parsing algorithm. This flexibility has led to the development of a diverse set of transition systems that address different aspects of syntax and semantics including: assigning part of speech tags (Choi and Palmer, 2011a), allowing the generation of non-projective dependency structures (Nivre, 2009), assigning semantic roles (Choi and Palmer, 2011b), and parsing texts containing multiple languages (Bhat et al., 2017).

Beam Search

The computational efficiency of the transition-based approach discussed earlier derives from the fact that it makes a single pass through the sentence, greedily making decisions without considering alternatives. Of course, this is also the source of its greatest weakness – once a decision has been made it can not be undone, even in the face of overwhelming evidence arriving later in a sentence. Another approach is to systematically explore alternative decision sequences, selecting the best among those alternatives. The key problem for such a search is to manage the large number of potential sequences. **Beam search** accomplishes this by combining a breadth-first search strategy with a heuristic filter that prunes the search frontier to stay within a fixed-size **beam width**.

Beam search

beam width

In applying beam search to transition-based parsing, we'll elaborate on the algorithm given in Fig. 13.6. Instead of choosing the single best transition operator at each iteration, we'll apply all applicable operators to each state on an agenda and then score the resulting configurations. We then add each of these new configurations to the frontier, subject to the constraint that there has to be room within the beam. As long as the size of the agenda is within the specified beam width, we can add new configurations to the agenda. Once the agenda reaches the limit, we only add new configurations that are better than the worst configuration on the agenda (removing the worst element so that we stay within the limit). Finally, to insure that we retrieve the best possible state on the agenda, the while loop continues as long as there are non-final states on the agenda.

The beam search approach requires a more elaborate notion of scoring than we used with the greedy algorithm. There, we assumed that a classifier trained using supervised machine learning would serve as an oracle, selecting the best transition operator based on features extracted from the current configuration. Regardless of the specific learning approach, this choice can be viewed as assigning a score to all

the possible transitions and picking the best one.

$$\hat{T}(c) = argmaxScore(t,c)$$

With a beam search we are now searching through the space of decision sequences, so it makes sense to base the score for a configuration on its entire history. More specifically, we can define the score for a new configuration as the score of its predecessor plus the score of the operator used to produce it.

$$\begin{aligned} ConfigScore(c_0) &= 0.0 \\ ConfigScore(c_i) &= ConfigScore(c_{i-1}) + Score(t_i, c_{i-1}) \end{aligned}$$

This score is used both in filtering the agenda and in selecting the final answer. The new beam search version of transition-based parsing is given in Fig. 13.11.

```
function DEPENDENCYBEAMPARSE(words, width) returns dependency tree

  state ← {[root], [words], [], 0.0}     ;initial configuration
  agenda ← ⟨state⟩;       initial agenda

  while agenda contains non-final states
    newagenda ← ⟨⟩
    for each state ∈ agenda do
      for all {t | t ∈ VALIDOPERATORS(state)} do
        child ← APPLY(t, state)
        newagenda ← ADDTOBEAM(child, newagenda, width)
    agenda ← newagenda
  return BESTOF(agenda)

function ADDTOBEAM(state, agenda, width) returns updated agenda

  if LENGTH(agenda) < width then
     agenda ← INSERT(state, agenda)
  else if SCORE(state) > SCORE(WORSTOF(agenda))
     agenda ← REMOVE(WORSTOF(agenda))
     agenda ← INSERT(state, agenda)
  return agenda
```

Figure 13.11 Beam search applied to transition-based dependency parsing.

13.5 Graph-Based Dependency Parsing

Graph-based approaches to dependency parsing search through the space of possible trees for a given sentence for a tree (or trees) that maximize some score. These methods encode the search space as directed graphs and employ methods drawn from graph theory to search the space for optimal solutions. More formally, given a sentence S we're looking for the best dependency tree in $\mathscr{G}_s$, the space of all possible trees for that sentence, that maximizes some score.

$$\hat{T}(S) = \underset{t \in \mathscr{G}_S}{\operatorname{argmax}}\, score(t,S)$$

As with the probabilistic approaches to context-free parsing discussed in Chapter 12, the overall score for a tree can be viewed as a function of the scores of the parts of the tree. The focus of this section is on **edge-factored** approaches where the score for a tree is based on the scores of the edges that comprise the tree.

edge-factored

$$score(t,S) = \sum_{e \in t} score(e)$$

There are several motivations for the use of graph-based methods. First, unlike transition-based approaches, these methods are capable of producing non-projective trees. Although projectivity is not a significant issue for English, it is definitely a problem for many of the world's languages. A second motivation concerns parsing accuracy, particularly with respect to longer dependencies. Empirically, transition-based methods have high accuracy on shorter dependency relations but accuracy declines significantly as the distance between the head and dependent increases (McDonald and Nivre, 2011). Graph-based methods avoid this difficulty by scoring entire trees, rather than relying on greedy local decisions.

The following section examines a widely-studied approach based on the use of a **maximum spanning tree** (MST) algorithm for weighted, directed graphs. We then discuss features that are typically used to score trees, as well as the methods used to train the scoring models.

maximum spanning tree

13.5.1 Parsing

The approach described here uses an efficient greedy algorithm to search for optimal spanning trees in directed graphs. Given an input sentence, it begins by constructing a fully-connected, weighted, directed graph where the vertices are the input words and the directed edges represent *all possible* head-dependent assignments. An additional ROOT node is included with outgoing edges directed at all of the other vertices. The weights in the graph reflect the score for each possible head-dependent relation as provided by a model generated from training data. Given these weights, a maximum spanning tree of this graph emanating from the ROOT represents the preferred dependency parse for the sentence. A directed graph for the example *Book that flight* is shown in Fig. 13.12, with the maximum spanning tree corresponding to the desired parse shown in blue. For ease of exposition, we'll focus here on unlabeled dependency parsing. Graph-based approaches to labeled parsing are discussed in Section 13.5.3.

Before describing the algorithm it's useful to consider two intuitions about directed graphs and their spanning trees. The first intuition begins with the fact that every vertex in a spanning tree has exactly one incoming edge. It follows from this that every *connected component* of a spanning tree will also have one incoming edge. The second intuition is that the absolute values of the edge scores are not critical to determining its maximum spanning tree. Instead, it is the relative weights of the edges entering each vertex that matters. If we were to subtract a constant amount from each edge entering a given vertex it would have no impact on the choice of the maximum spanning tree since every possible spanning tree would decrease by exactly the same amount.

The first step of the algorithm itself is quite straightforward. For each vertex in the graph, an incoming edge (representing a possible head assignment) with the highest score is chosen. If the resulting set of edges produces a spanning tree then we're done. More formally, given the original fully-connected graph $G = (V,E)$, a subgraph $T = (V,F)$ is a spanning tree if it has no cycles and each vertex (other than

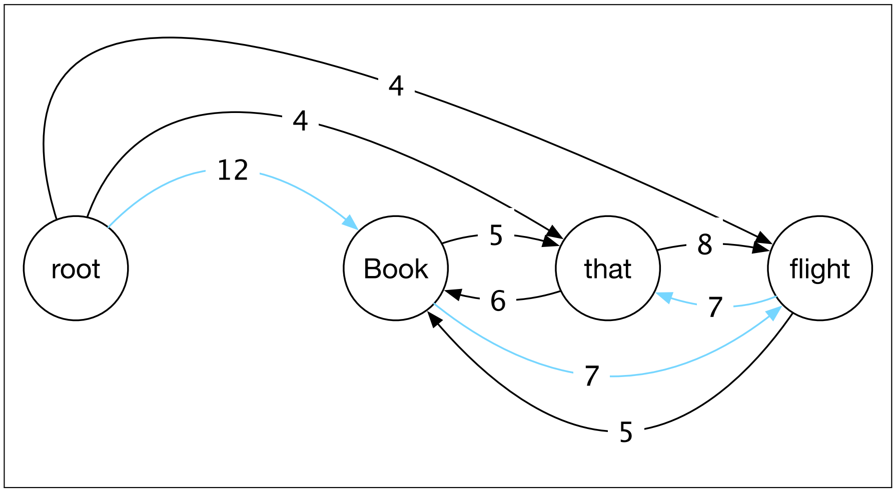


Figure 13.12 Initial rooted, directed graph for *Book that flight*.

the root) has exactly one edge entering it. If the greedy selection process produces such a tree then it is the best possible one.

Unfortunately, this approach doesn't always lead to a tree since the set of edges selected may contain cycles. Fortunately, in yet another case of multiple discovery, there is a straightforward way to eliminate cycles generated during the greedy selection phase. Chu and Liu (1965) and Edmonds (1967) independently developed an approach that begins with greedy selection and follows with an elegant recursive cleanup phase that eliminates cycles.

The cleanup phase begins by adjusting all the weights in the graph by subtracting the score of the maximum edge entering each vertex from the score of all the edges entering that vertex. This is where the intuitions mentioned earlier come into play. We have scaled the values of the edges so that the weight of the edges in the cycle have no bearing on the weight of *any* of the possible spanning trees. Subtracting the value of the edge with maximum weight from each edge entering a vertex results in a weight of zero for all of the edges selected during the greedy selection phase, *including all of the edges involved in the cycle*.

Having adjusted the weights, the algorithm creates a new graph by selecting a cycle and collapsing it into a single new node. Edges that enter or leave the cycle are altered so that they now enter or leave the newly collapsed node. Edges that do not touch the cycle are included and edges within the cycle are dropped.

Now, if we knew the maximum spanning tree of this new graph, we would have what we need to eliminate the cycle. The edge of the maximum spanning tree directed towards the vertex representing the collapsed cycle tells us which edge to delete to eliminate the cycle. How do we find the maximum spanning tree of this new graph? We recursively apply the algorithm to the new graph. This will either result in a spanning tree or a graph with a cycle. The recursions can continue as long as cycles are encountered. When each recursion completes we expand the collapsed vertex, restoring all the vertices and edges from the cycle *with the exception of the single edge to be deleted*.

Putting all this together, the maximum spanning tree algorithm consists of greedy edge selection, re-scoring of edge costs and a recursive cleanup phase when needed. The full algorithm is shown in Fig. 13.13.

function MAXSPANNINGTREE(*G=(V,E)*, *root*, *score*) **returns** *spanning tree*

$F \leftarrow []$
$T' \leftarrow []$
$score' \leftarrow []$
for each $v \in V$ **do**
  $bestInEdge \leftarrow \text{argmax}_{e=(u,v)\in E}\ score[e]$
  $F \leftarrow F \cup bestInEdge$
  for each e=$(u,v) \in E$ **do**
    $score'[e] \leftarrow score[e] - score[bestInEdge]$

  if T=(V,F) is a spanning tree **then return** it
  else
    $C \leftarrow$ a cycle in F
    $G' \leftarrow$ CONTRACT(G, C)
    $T' \leftarrow$ MAXSPANNINGTREE(G', *root*, *score'*)
    T $\leftarrow$ EXPAND(T', C)
    return T

function CONTRACT(G, C) **returns** *contracted graph*

function EXPAND(T, C) **returns** *expanded graph*

Figure 13.13 The Chu-Liu Edmonds algorithm for finding a maximum spanning tree in a weighted directed graph.

Fig. 13.14 steps through the algorithm with our *Book that flight* example. The first row of the figure illustrates greedy edge selection with the edges chosen shown in blue (corresponding to the set F in the algorithm). This results in a cycle between *that* and *flight*. The scaled weights using the maximum value entering each node are shown in the graph to the right.

Collapsing the cycle between *that* and *flight* to a single node (labelled *tf*) and recursing with the newly scaled costs is shown in the second row. The greedy selection step in this recursion yields a spanning tree that links *root* to *book*, as well as an edge that links *book* to the contracted node. Expanding the contracted node, we can see that this edge corresponds to the edge from *book* to *flight* in the original graph. This in turn tells us which edge to drop to eliminate the cycle

On arbitrary directed graphs, this version of the CLE algorithm runs in $O(mn)$ time, where m is the number of edges and n is the number of nodes. Since this particular application of the algorithm begins by constructing a fully connected graph $m = n^2$ yielding a running time of $O(n^3)$. Gabow et al. (1986) present a more efficient implementation with a running time of $O(m + nlogn)$.

13.5.2 Features and Training

Given a sentence, S, and a candidate tree, T, edge-factored parsing models reduce the score for the tree to a sum of the scores of the edges that comprise the tree.

$$score(S,T) = \sum_{e \in T} score(S,e)$$

Each edge score can, in turn, be reduced to a weighted sum of features extracted

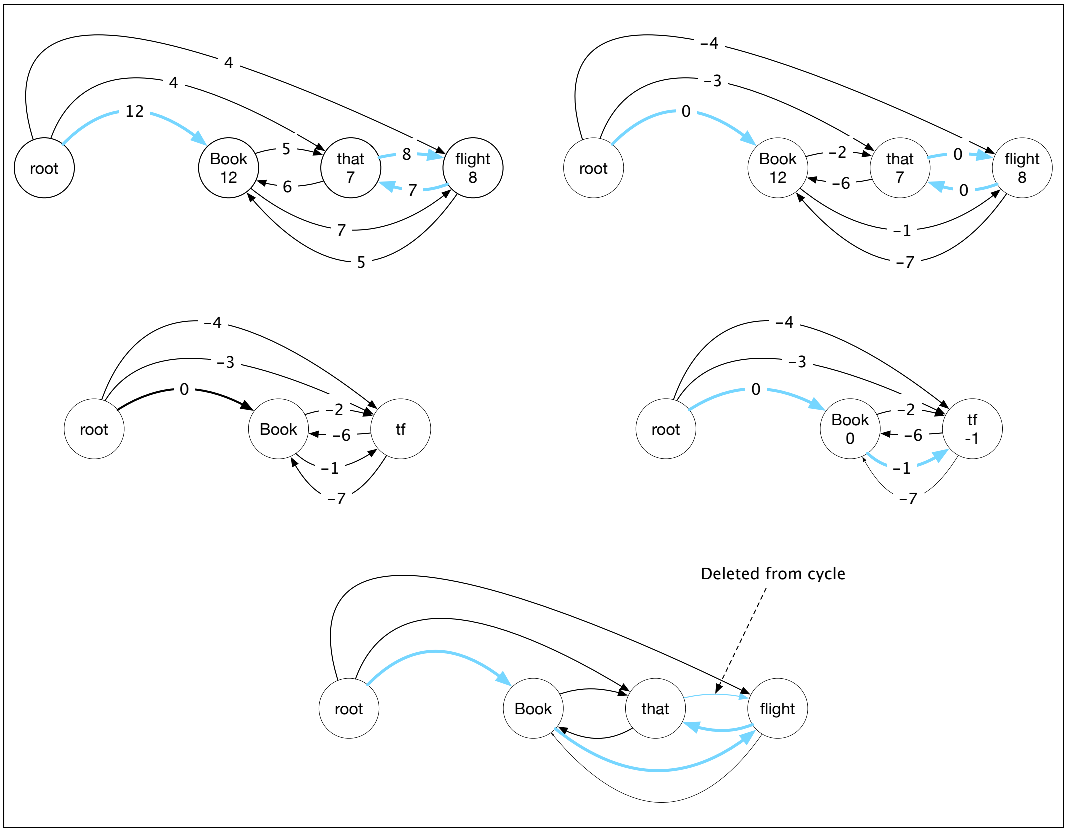


Figure 13.14 Chu-Liu-Edmonds graph-based example for *Book that flight*

from it.

$$score(S,e) \;=\; \sum_{i=1}^{N} w_i f_i(S,e)$$

Or more succinctly.

$$score(S,e) \;=\; w \cdot f$$

Given this formulation, we are faced with two problems in training our parser: identifying relevant features and finding the weights used to score those features.

The features used to train edge-factored models mirror those used in training transition-based parsers (as shown in Fig. 13.9). This is hardly surprising since in both cases we're trying to capture information about the relationship between heads and their dependents in the context of a single relation. To summarize this earlier discussion, commonly used features include:

- Wordforms, lemmas, and parts of speech of the headword and its dependent.
- Corresponding features derived from the contexts before, after and between the words.
- Word embeddings.
- The dependency relation itself.
- The direction of the relation (to the right or left).

- The distance from the head to the dependent.

As with transition-based approaches, pre-selected combinations of these features are often used as well.

Given a set of features, our next problem is to learn a set of weights corresponding to each. Unlike many of the learning problems discussed in earlier chapters, here we are not training a model to associate training items with class labels, or parser actions. Instead, we seek to train a model that assigns higher scores to correct trees than to incorrect ones. An effective framework for problems like this is to use **inference-based learning** combined with the perceptron learning rule. In this framework, we parse a sentence (i.e, perform inference) from the training set using some initially random set of initial weights. If the resulting parse matches the corresponding tree in the training data, we do nothing to the weights. Otherwise, we find those features in the incorrect parse that are *not* present in the reference parse and we lower their weights by a small amount based on the learning rate. We do this incrementally for each sentence in our training data until the weights converge.

inference-based learning

More recently, recurrent neural network (RNN) models have demonstrated state-of-the-art performance in shared tasks on multilingual parsing (Zeman et al. 2017,Dozat et al. 2017). These neural approaches rely solely on lexical information in the form of word embeddings, eschewing the use of hand-crafted features such as those described earlier.

13.5.3 Advanced Issues in Graph-Based Parsing

13.6 Evaluation

As with phrase structure-based parsing, the evaluation of dependency parsers proceeds by measuring how well they work on a test-set. An obvious metric would be exact match (EM) — how many sentences are parsed correctly. This metric is quite pessimistic, with most sentences being marked wrong. Such measures are not fine-grained enough to guide the development process. Our metrics need to be sensitive enough to tell if actual improvements are being made.

For these reasons, the most common method for evaluating dependency parsers are labeled and unlabeled attachment accuracy. Labeled attachment refers to the proper assignment of a word to its head along with the correct dependency relation. Unlabeled attachment simply looks at the correctness of the assigned head, ignoring the dependency relation. Given a system output and a corresponding reference parse, accuracy is simply the percentage of words in an input that are assigned the correct head with the correct relation. This metrics are usually referred to as the labeled attachment score (LAS) and unlabeled attachment score (UAS). Finally, we can make use of a label accuracy score (LS), the percentage of tokens with correct labels, ignoring where the relations are coming from.

As an example, consider the reference parse and system parse for the following example shown in Fig. 13.15.

(13.11) Book me the flight through Houston.

The system correctly finds 4 of the 6 dependency relations present in the reference parse and therefore receives an LAS of 2/3. However, one of the 2 incorrect relations found by the system holds between *book* and *flight*, which are in a head-dependent relation in the reference parse; therefore the system therefore achieves an

UAS of 5/6.

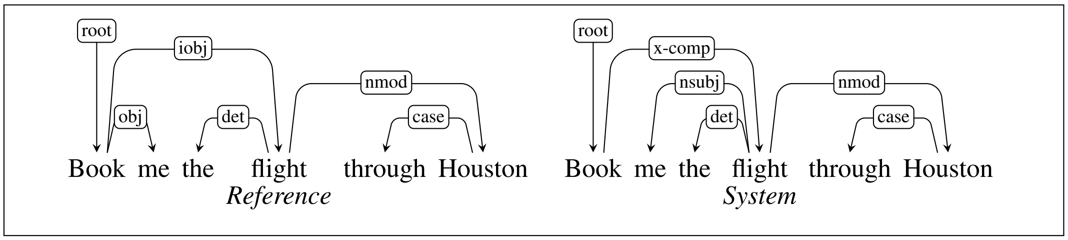


Figure 13.15 Reference and system parses for *Book me the flight through Houston*, resulting in an LAS of 4/6 and an UAS of 5/6.

Beyond attachment scores, we may also be interested in how well a system is performing on a particular kind of dependency relation, for example NSUBJ, across a development corpus. Here we can make use of the notions of precision and recall introduced in Chapter 8, measuring the percentage of relations labeled NSUBJ by the system that were correct (precision), and the percentage of the NSUBJ relations present in the development set that were in fact discovered by the system (recall). We can employ a confusion matrix to keep track of how often each dependency type was confused for another.

13.7 Summary

This chapter has introduced the concept of dependency grammars and dependency parsing. Here's a summary of the main points that we covered:

- In dependency-based approaches to syntax, the structure of a sentence is described in terms of a set of binary relations that hold between the words in a sentence. Larger notions of constituency are not directly encoded in dependency analyses.
- The relations in a dependency structure capture the head-dependent relationship among the words in a sentence.
- Dependency-based analyses provides information directly useful in further language processing tasks including information extraction, semantic parsing and question answering
- Transition-based parsing systems employ a greedy stack-based algorithm to create dependency structures.
- Graph-based methods for creating dependency structures are based on the use of maximum spanning tree methods from graph theory.
- Both transition-based and graph-based approaches are developed using supervised machine learning techniques.
- Treebanks provide the data needed to train these systems. Dependency treebanks can be created directly by human annotators or via automatic transformation from phrase-structure treebanks.
- Evaluation of dependency parsers is based on labeled and unlabeled accuracy scores as measured against withheld development and test corpora.

Bibliographical and Historical Notes

The dependency-based approach to grammar is much older than the relatively recent phrase-structure or constituency grammars that have been the primary focus of both theoretical and computational linguistics for years. It has its roots in the ancient Greek and Indian linguistic traditions. Contemporary theories of dependency grammar all draw heavily on the work of Tesnière (1959). The most influential dependency grammar frameworks include Meaning-Text Theory (MTT) (Mel'čuk, 1988), Word Grammar (Hudson, 1984), Functional Generative Description (FDG) (Sgall et al., 1986). These frameworks differ along a number of dimensions including the degree and manner in which they deal with morphological, syntactic, semantic and pragmatic factors, their use of multiple layers of representation, and the set of relations used to categorize dependency relations.

Automatic parsing using dependency grammars was first introduced into computational linguistics by early work on machine translation at the RAND Corporation led by David Hays. This work on dependency parsing closely paralleled work on constituent parsing and made explicit use of grammars to guide the parsing process. After this early period, computational work on dependency parsing remained intermittent over the following decades. Notable implementations of dependency parsers for English during this period include Link Grammar (Sleator and Temperley, 1993), Constraint Grammar (Karlsson et al., 1995), and MINIPAR (Lin, 2003).

Dependency parsing saw a major resurgence in the late 1990's with the appearance of large dependency-based treebanks and the associated advent of data driven approaches described in this chapter. Eisner (1996) developed an efficient dynamic programming approach to dependency parsing based on bilexical grammars derived from the Penn Treebank. Covington (2001) introduced the deterministic word by word approach underlying current transition-based approaches. Yamada and Matsumoto (2003) and Kudo and Matsumoto (2002) introduced both the shift-reduce paradigm and the use of supervised machine learning in the form of support vector machines to dependency parsing.

Nivre (2003) defined the modern, deterministic, transition-based approach to dependency parsing. Subsequent work by Nivre and his colleagues formalized and analyzed the performance of numerous transition systems, training methods, and methods for dealing with non-projective language Nivre and Scholz 2004,Nivre 2006,Nivre and Nilsson 2005,Nivre et al. 2007,Nivre 2007.

The graph-based maximum spanning tree approach to dependency parsing was introduced by McDonald et al. 2005,McDonald et al. 2005.

The earliest source of data for training and evaluating dependency English parsers came from the WSJ Penn Treebank (Marcus et al., 1993) described in Chapter 10. The use of head-finding rules developed for use with probabilistic parsing facilitated the automatic extraction of dependency parses from phrase-based ones (Xia and Palmer, 2001).

The long-running Prague Dependency Treebank project (Hajič, 1998) is the most significant effort to directly annotate a corpus with multiple layers of morphological, syntactic and semantic information. The current PDT 3.0 now contains over 1.5 M tokens (Bejček et al., 2013).

Universal Dependencies (UD) (Nivre et al., 2016b) is a project directed at creating a consistent framework for dependency treebank annotation across languages with the goal of advancing parser development across the worlds languages. Un-

der the auspices of this effort, treebanks for over 30 languages have been annotated and made available in a single consistent format. The UD annotation scheme evolved out of several distinct efforts including Stanford dependencies de Marneffe et al. 2006,de Marneffe and Manning 2008,de Marneffe et al. 2014, Google's universal part-of-speech tags (Petrov et al., 2012) al., 2012), and the Interset interlingua for morphosyntactic tagsets (Zeman, 2008). Driven in part by the UD framework, dependency treebanks of a significant size and quality are now available in over 30 languages (Nivre et al., 2016b).

The Conference on Natural Language Learning (CoNLL) has conducted an influential series of shared tasks related to dependency parsing over the years (Buchholz and Marsi 2006,Nilsson et al. 2007,Surdeanu et al. 2008a,Hajič et al. 2009). More recent evaluations have focused on parser robustness with respect to morphologically rich languages (Seddah et al., 2013), and non-canonical language forms such as social media, texts, and spoken language (Petrov and McDonald, 2012). Choi et al. (2015) presents a detailed performance analysis of 10 state-of-the-art dependency parsers across an impressive range of metrics, as well as DEPENDABLE, a robust parser evaluation tool.

Exercises

CHAPTER

14 The Representation of Sentence Meaning

Placeholder

CHAPTER

15 Computational Semantics

Placeholder

CHAPTER

16 Semantic Parsing

Placeholder

CHAPTER 17 Information Extraction

I am the very model of a modern Major-General,
I've information vegetable, animal, and mineral,
I know the kings of England, and I quote the fights historical
From Marathon to Waterloo, in order categorical...
Gilbert and Sullivan, *Pirates of Penzance*

Imagine that you are an analyst with an investment firm that tracks airline stocks. You're given the task of determining the relationship (if any) between airline announcements of fare increases and the behavior of their stocks the next day. Historical data about stock prices is easy to come by, but what about the airline announcements? You will need to know at least the name of the airline, the nature of the proposed fare hike, the dates of the announcement, and possibly the response of other airlines. Fortunately, these can be all found in news articles like this one:

> Citing high fuel prices, United Airlines said Friday it has increased fares by $6 per round trip on flights to some cities also served by lower-cost carriers. American Airlines, a unit of AMR Corp., immediately matched the move, spokesman Tim Wagner said. United, a unit of UAL Corp., said the increase took effect Thursday and applies to most routes where it competes against discount carriers, such as Chicago to Dallas and Denver to San Francisco.

information extraction

This chapter presents techniques for extracting limited kinds of semantic content from text. This process of **information extraction** (IE), turns the unstructured information embedded in texts into structured data, for example for populating a relational database to enable further processing.

named entity recognition

We begin with the first step in most IE tasks, finding the proper names or **named entities** in a text. The task of **named entity recognition** (NER) is to find each **mention** of a named entity in the text and label its type. What constitutes a named entity type is task specific; people, places, and organizations are common, but gene or protein names (Cohen and Demner-Fushman, 2014) or financial asset classes might be relevant for some tasks. Once all the named entities in a text have been extracted, they can be linked together in sets corresponding to real-world entities, inferring, for example, that mentions of *United Airlines* and *United* refer to the same company. This is the joint task of **coreference resolution** and **entity linking** which we defer til Chapter 21.

relation extraction

Next, we turn to the task of **relation extraction**: finding and classifying semantic relations among the text entities. These are often binary relations like child-of, employment, part-whole, and geospatial relations. Relation extraction has close links to populating a relational database.

event extraction

Finally, we discuss three tasks related to *events*. **Event extraction** is finding events in which these entities participate, like, in our sample text, the fare increases

by *United* and *American* and the reporting events *said* and *cite*. **Event coreference** (Chapter 21) is needed to figure out which event mentions in a text refer to the same event; in our running example the two instances of *increase* and the phrase *the move* all refer to the same event.

temporal expression

temporal normalization

To figure out *when* the events in a text happened we extract **temporal expressions** like days of the week (*Friday* and *Thursday*), relative expressions like *two days from now* or *next year* and times such as *3:30 P.M.*. These expressions must be **normalized** onto specific calendar dates or times of day to situate events in time. In our sample task, this will allow us to link *Friday* to the time of United's announcement, and *Thursday* to the previous day's fare increase, and produce a timeline in which United's announcement follows the fare increase and American's announcement follows both of those events.

template filling

Finally, many texts describe recurring stereotypical events or situations. The task of **template filling** is to find such situations in documents and fill in the template slots. These slot-fillers may consist of text segments extracted directly from the text, or concepts like times, amounts, or ontology entities that have been inferred from text elements through additional processing.

Our airline text is an example of this kind of stereotypical situation since airlines often raise fares and then wait to see if competitors follow along. In this situation, we can identify *United* as a lead airline that initially raised its fares, $6 as the amount, *Thursday* as the increase date, and *American* as an airline that followed along, leading to a filled template like the following.

FARE-RAISE ATTEMPT:	LEAD AIRLINE:	UNITED AIRLINES
	AMOUNT:	$6
	EFFECTIVE DATE:	2006-10-26
	FOLLOWER:	AMERICAN AIRLINES

17.1 Named Entity Recognition

named entity

temporal expressions

The first step in information extraction is to detect the **entities** in the text. A **named entity** is, roughly speaking, anything that can be referred to with a proper name: a person, a location, an organization. The term is commonly extended to include things that aren't entities per se, including dates, times, and other kinds of **temporal expressions**, and even numerical expressions like prices. Here's the sample text introduced earlier with the named entities marked:

> Citing high fuel prices, [$_{ORG}$ **United Airlines**] said [$_{TIME}$ **Friday**] it has increased fares by [$_{MONEY}$ **$6**] per round trip on flights to some cities also served by lower-cost carriers. [$_{ORG}$ **American Airlines**], a unit of [$_{ORG}$ **AMR Corp.**], immediately matched the move, spokesman [$_{PER}$ **Tim Wagner**] said. [$_{ORG}$ **United**], a unit of [$_{ORG}$ **UAL Corp.**], said the increase took effect [$_{TIME}$ **Thursday**] and applies to most routes where it competes against discount carriers, such as [$_{LOC}$ **Chicago**] to [$_{LOC}$ **Dallas**] and [$_{LOC}$ **Denver**] to [$_{LOC}$ **San Francisco**].

The text contains 13 mentions of named entities including 5 organizations, 4 locations, 2 times, 1 person, and 1 mention of money.

In addition to their use in extracting events and the relationship between participants, named entities are useful for many other language processing tasks. In

sentiment analysis we might want to know a consumer's sentiment toward a particular entity. Entities are a useful first stage in question answering, or for linking text to information in structured knowledge sources like Wikipedia.

Figure 17.1 shows typical generic named entity types. Many applications will also need to use specific entity types like proteins, genes, commercial products, or works of art.

Type	Tag	Sample Categories	Example sentences
People	PER	people, characters	**Turing** is a giant of computer science.
Organization	ORG	companies, sports teams	The **IPCC** warned about the cyclone.
Location	LOC	regions, mountains, seas	The **Mt. Sanitas** loop is in **Sunshine Canyon**.
Geo-Political Entity	GPE	countries, states, provinces	**Palo Alto** is raising the fees for parking.
Facility	FAC	bridges, buildings, airports	Consider the **Golden Gate Bridge**.
Vehicles	VEH	planes, trains, automobiles	It was a classic **Ford Falcon**.

Figure 17.1 A list of generic named entity types with the kinds of entities they refer to.

Named entity recognition means finding spans of text that constitute proper names and then classifying the type of the entity. Recognition is difficult partly because of the ambiguity of segmentation; we need to decide what's an entity and what isn't, and where the boundaries are. Another difficulty is caused by type ambiguity. The mention `JFK` can refer to a person, the airport in New York, or any number of schools, bridges, and streets around the United States. Some examples of this kind of cross-type confusion are given in Figures 17.2 and 17.3.

Name	Possible Categories
Washington	Person, Location, Political Entity, Organization, Vehicle
Downing St.	Location, Organization
IRA	Person, Organization, Monetary Instrument
Louis Vuitton	Person, Organization, Commercial Product

Figure 17.2 Common categorical ambiguities associated with various proper names.

[$_{\text{PER}}$ Washington] was born into slavery on the farm of James Burroughs.
[$_{\text{ORG}}$ Washington] went up 2 games to 1 in the four-game series.
Blair arrived in [$_{\text{LOC}}$ Washington] for what may well be his last state visit.
In June, [$_{\text{GPE}}$ Washington] passed a primary seatbelt law.
The [$_{\text{VEH}}$ Washington] had proved to be a leaky ship, every passage I made...

Figure 17.3 Examples of type ambiguities in the use of the name *Washington*.

17.1.1 NER as Sequence Labeling

The standard algorithm for named entity recognition is as a word-by-word sequence labeling task, in which the assigned tags capture both the boundary and the type. A sequence classifier like an MEMM/CRF or a bi-LSTM is trained to label the tokens in a text with tags that indicate the presence of particular kinds of named entities. Consider the following simplified excerpt from our running example.

> [$_{\text{ORG}}$ **American Airlines**], a unit of [$_{\text{ORG}}$ **AMR Corp.**], immediately matched the move, spokesman [$_{\text{PER}}$ **Tim Wagner**] said.

IOB Figure 17.4 shows the same excerpt represented with **IOB** tagging. In IOB tagging we introduce a tag for the beginning (B) and inside (I) of each entity type, and one for tokens outside (O) any entity. The number of tags is thus $2n+1$ tags, where n is the number of entity types. IOB tagging can represent exactly the same information as the bracketed notation.

Words	IOB Label	IO Label
American	B-ORG	I-ORG
Airlines	I-ORG	I-ORG
,	O	O
a	O	O
unit	O	O
of	O	O
AMR	B-ORG	I-ORG
Corp.	I-ORG	I-ORG
,	O	O
immediately	O	O
matched	O	O
the	O	O
move	O	O
,	O	O
spokesman	O	O
Tim	B-PER	I-PER
Wagner	I-PER	I-PER
said	O	O
.	O	O

Figure 17.4 Named entity tagging as a sequence model, showing IOB and IO encodings.

We've also shown IO tagging, which loses some information by eliminating the B tag. Without the B tag IO tagging is unable to distinguish between two entities of the same type that are right next to each other. Since this situation doesn't arise very often (usually there is at least some punctuation or other deliminator), IO tagging may be sufficient, and has the advantage of using only $n+1$ tags.

In the following three sections we introduce the three standard families of algorithms for NER tagging: feature based (MEMM/CRF), neural (bi-LSTM), and rule-based.

17.1.2 A feature-based algorithm for NER

identity of w_i, identity of neighboring words
embeddings for w_i, embeddings for neighboring words
part of speech of w_i, part of speech of neighboring words
base-phrase syntactic chunk label of w_i and neighboring words
presence of w_i in a **gazetteer**
w_i contains a particular prefix (from all prefixes of length ≤ 4)
w_i contains a particular suffix (from all suffixes of length ≤ 4)
w_i is all upper case
word shape of w_i, word shape of neighboring words
short word shape of w_i, short word shape of neighboring words
presence of hyphen

Figure 17.5 Typical features for a feature-based NER system.

The first approach is to extract features and train an MEMM or CRF sequence model of the type we saw for part-of-speech tagging in Chapter 8. Figure 17.5 lists standard features used in such feature-based systems. We've seen many of these features before in the context of part-of-speech tagging, particularly for tagging unknown words. This is not surprising, as many unknown words are in fact named entities. Word shape features are thus particularly important in the context of NER. Recall that **word shape** features are used to represent the abstract letter pattern of the word by mapping lower-case letters to 'x', upper-case to 'X', numbers to 'd', and retaining punctuation. Thus for example I.M.F would map to X.X.X. and DC10-30 would map to XXdd-dd. A second class of shorter word shape features is also used. In these features consecutive character types are removed, so DC10-30 would be mapped to Xd-d but I.M.F would still map to X.X.X. This feature by itself accounts for a considerable part of the success of feature-based NER systems for English news text. Shape features are also particularly important in recognizing names of proteins and genes in biological texts.

word shape

For example the named entity token *L'Occitane* would generate the following non-zero valued feature values:

prefix(w_i) = `L`	suffix(w_i) = `tane`
prefix(w_i) = `L'`	suffix(w_i) = `ane`
prefix(w_i) = `L'O`	suffix(w_i) = `ne`
prefix(w_i) = `L'Oc`	suffix(w_i) = `e`
word-shape(w_i) = `X'Xxxxxxxx`	short-word-shape(w_i) = `X'Xx`

gazetteer

A **gazetteer** is a list of place names, often providing millions of entries for locations with detailed geographical and political information.[1] A related resource is **name-lists**; the United States Census Bureau also provides extensive lists of first names and surnames derived from its decadal census in the U.S.[2] Similar lists of corporations, commercial products, and all manner of things biological and mineral are also available from a variety of sources. Gazetteer and name features are typically implemented as a binary feature for each name list. Unfortunately, such lists can be difficult to create and maintain, and their usefulness varies considerably. While gazetteers can be quite effective, lists of persons and organizations are not always helpful (Mikheev et al., 1999).

Feature effectiveness depends on the application, genre, media, and language. For example, shape features, critical for English newswire texts, are of little use with automatic speech recognition transcripts, or other non-edited or informally-edited sources, or for languages like Chinese that don't use orthographic case. The features in Fig. 17.5 should therefore be thought of as only a starting point.

Figure 17.6 illustrates the result of adding part-of-speech tags, syntactic base-phrase chunk tags, and some shape information to our earlier example.

Given such a training set, a sequence classifier like an MEMM can be trained to label new sentences. Figure 17.7 illustrates the operation of such a sequence labeler at the point where the token *Corp.* is next to be labeled. If we assume a context window that includes the two preceding and following words, then the features available to the classifier are those shown in the boxed area.

[1] www.geonames.org
[2] www.census.gov

Word	POS	Chunk	Short shape	Label
American	NNP	B-NP	Xx	B-ORG
Airlines	NNPS	I-NP	Xx	I-ORG
,	,	O	,	O
a	DT	B-NP	x	O
unit	NN	I-NP	x	O
of	IN	B-PP	x	O
AMR	NNP	B-NP	X	B-ORG
Corp.	NNP	I-NP	Xx.	I-ORG
,	,	O	,	O
immediately	RB	B-ADVP	x	O
matched	VBD	B-VP	x	O
the	DT	B-NP	x	O
move	NN	I-NP	x	O
,	,	O	,	O
spokesman	NN	B-NP	x	O
Tim	NNP	I-NP	Xx	B-PER
Wagner	NNP	I-NP	Xx	I-PER
said	VBD	B-VP	x	O
.	,	O	.	O

Figure 17.6 Word-by-word feature encoding for NER.

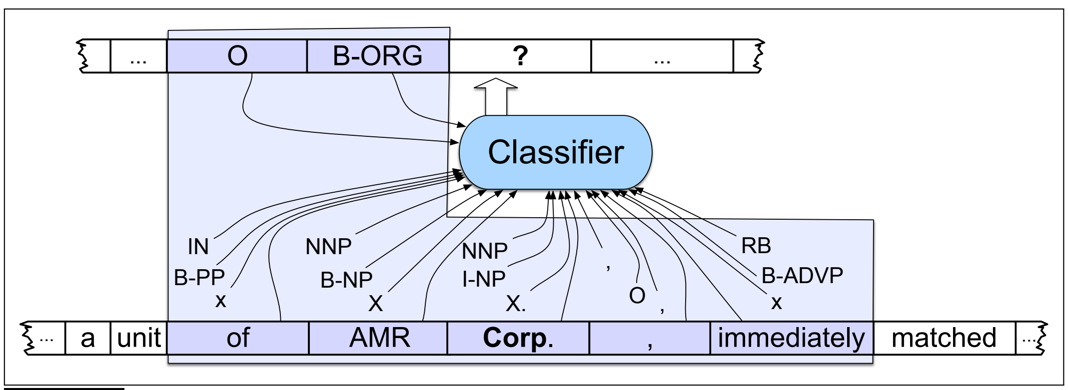


Figure 17.7 Named entity recognition as sequence labeling. The features available to the classifier during training and classification are those in the boxed area.

17.1.3 A neural algorithm for NER

The standard neural algorithm for NER is based on the bi-LSTM introduced in Chapter 9. Recall that in that model, word and character embeddings are computed for input word w_i. These are passed through a left-to-right LSTM and a right-to-left LSTM, whose outputs are concatenated (or otherwise combined) to produce a single output layer at position i. In the simplest method, this layer can then be directly passed onto a softmax that creates a probability distribution over all NER tags, and the most likely tag is chosen as t_i.

For named entity tagging this greedy approach to decoding is insufficient, since it doesn't allow us to impose the strong constraints neighboring tokens have on each other (e.g., the tag I-PER must follow another I-PER or B-PER). Instead a CRF layer is normally used on top of the bi-LSTM output, and the Viterbi decoding algorithm is used to decode. Fig. 17.8 shows a sketch of the algorithm

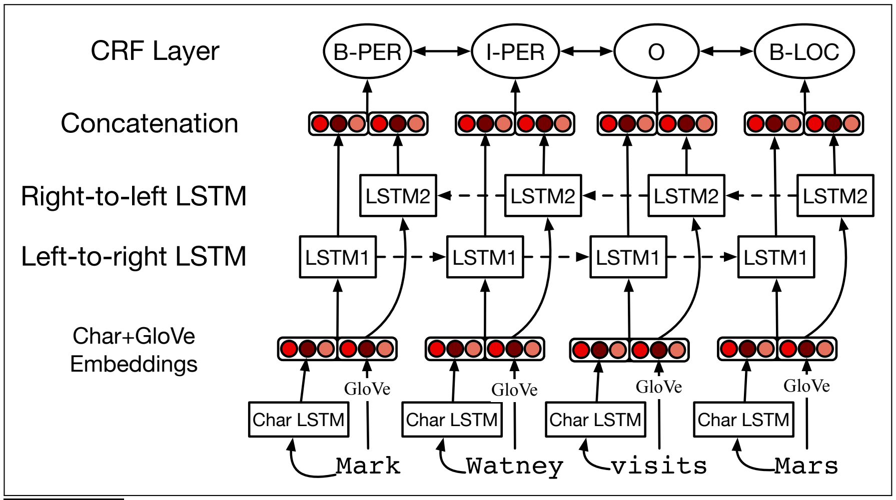


Figure 17.8 Putting it all together: character embeddings and words together a bi-LSTM sequence model. After (Lample et al., 2016)

17.1.4 Rule-based NER

While machine learned (neural or MEMM/CRF) sequence models are the norm in academic research, commercial approaches to NER are often based on pragmatic combinations of lists and rules, with some smaller amount of supervised machine learning (Chiticariu et al., 2013). For example IBM System T is a text understanding architecture in which a user specifies complex declarative constraints for tagging tasks in a formal query language that includes regular expressions, dictionaries, semantic constraints, NLP operators, and table structures, all of which the system compiles into an efficient extractor (Chiticariu et al., 2018)

One common approach is to make repeated rule-based passes over a text, allowing the results of one pass to influence the next. The stages typically first involve the use of rules that have extremely high precision but low recall. Subsequent stages employ more error-prone statistical methods that take the output of the first pass into account.

1. First, use high-precision rules to tag unambiguous entity mentions.
2. Then, search for substring matches of the previously detected names.
3. Consult application-specific name lists to identify likely name entity mentions from the given domain.
4. Finally, apply probabilistic sequence labeling techniques that make use of the tags from previous stages as additional features.

The intuition behind this staged approach is twofold. First, some of the entity mentions in a text will be more clearly indicative of a given entity's class than others. Second, once an unambiguous entity mention is introduced into a text, it is likely that subsequent shortened versions will refer to the same entity (and thus the same type of entity).

17.1.5 Evaluation of Named Entity Recognition

The familiar metrics of **recall**, **precision**, and F_1 **measure** are used to evaluate NER systems. Remember that recall is the ratio of the number of correctly labeled responses to the total that should have been labeled; precision is the ratio of the num-

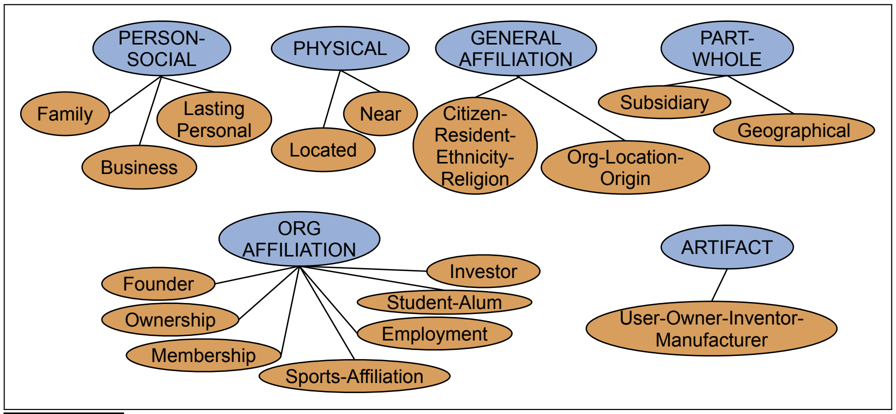


Figure 17.9 The 17 relations used in the ACE relation extraction task.

ber of correctly labeled responses to the total labeled; and *F*-measure is the harmonic mean of the two. For named entities, the *entity* rather than the word is the unit of response. Thus in the example in Fig. 17.6, the two entities *Tim Wagner* and *AMR Corp.* and the non-entity *said* would each count as a single response.

The fact that named entity tagging has a segmentation component which is not present in tasks like text categorization or part-of-speech tagging causes some problems with evaluation. For example, a system that labeled *American* but not *American Airlines* as an organization would cause two errors, a false positive for O and a false negative for I-ORG. In addition, using entities as the unit of response but words as the unit of training means that there is a mismatch between the training and test conditions.

17.2 Relation Extraction

Next on our list of tasks is to discern the relationships that exist among the detected entities. Let's return to our sample airline text:

> Citing high fuel prices, [$_{\text{ORG}}$ **United Airlines**] said [$_{\text{TIME}}$ **Friday**] it has increased fares by [$_{\text{MONEY}}$ **\$6**] per round trip on flights to some cities also served by lower-cost carriers. [$_{\text{ORG}}$ **American Airlines**], a unit of [$_{\text{ORG}}$ **AMR Corp.**], immediately matched the move, spokesman [$_{\text{PER}}$ **Tim Wagner**] said. [$_{\text{ORG}}$ **United**], a unit of [$_{\text{ORG}}$ **UAL Corp.**], said the increase took effect [$_{\text{TIME}}$ **Thursday**] and applies to most routes where it competes against discount carriers, such as [$_{\text{LOC}}$ **Chicago**] to [$_{\text{LOC}}$ **Dallas**] and [$_{\text{LOC}}$ **Denver**] to [$_{\text{LOC}}$ **San Francisco**].

The text tells us, for example, that *Tim Wagner* is a spokesman for *American Airlines*, that *United* is a unit of *UAL Corp.*, and that *American* is a unit of *AMR*. These binary relations are instances of more generic relations such as **part-of** or **employs** that are fairly frequent in news-style texts. Figure 17.9 lists the 17 relations used in the ACE relation extraction evaluations and Fig. 17.10 shows some sample relations. We might also extract more domain-specific relation such as the notion of an airline route. For example from this text we can conclude that United has routes to Chicago, Dallas, Denver, and San Francisco.

Relations	Types	Examples
Physical-Located	PER-GPE	**He** was in **Tennessee**
Part-Whole-Subsidiary	ORG-ORG	**XYZ**, the parent company of **ABC**
Person-Social-Family	PER-PER	**Yoko**'s husband **John**
Org-AFF-Founder	PER-ORG	**Steve Jobs**, co-founder of **Apple**...

Figure 17.10 Semantic relations with examples and the named entity types they involve.

Domain	$\mathscr{D} = \{a,b,c,d,e,f,g,h,i\}$
United, UAL, American Airlines, AMR	a,b,c,d
Tim Wagner	e
Chicago, Dallas, Denver, and San Francisco	f,g,h,i
Classes	
United, UAL, American, and AMR are organizations	$Org = \{a,b,c,d\}$
Tim Wagner is a person	$Pers = \{e\}$
Chicago, Dallas, Denver, and San Francisco are places	$Loc = \{f,g,h,i\}$
Relations	
United is a unit of UAL	$PartOf = \{\langle a,b\rangle, \langle c,d\rangle\}$
American is a unit of AMR	
Tim Wagner works for American Airlines	$OrgAff = \{\langle c,e\rangle\}$
United serves Chicago, Dallas, Denver, and San Francisco	$Serves = \{\langle a,f\rangle, \langle a,g\rangle, \langle a,h\rangle, \langle a,i\rangle\}$

Figure 17.11 A model-based view of the relations and entities in our sample text.

These relations correspond nicely to the model-theoretic notions we introduced in Chapter 14 to ground the meanings of the logical forms. That is, a relation consists of a set of ordered tuples over elements of a domain. In most standard information-extraction applications, the domain elements correspond to the named entities that occur in the text, to the underlying entities that result from co-reference resolution, or to entities selected from a domain ontology. Figure 17.11 shows a model-based view of the set of entities and relations that can be extracted from our running example. Notice how this model-theoretic view subsumes the NER task as well; named entity recognition corresponds to the identification of a class of unary relations.

Sets of relations have been defined for many other domains as well. For example UMLS, the Unified Medical Language System from the US National Library of Medicine has a network that defines 134 broad subject categories, entity types, and 54 relations between the entities, such as the following:

Entity	Relation	Entity
Injury	disrupts	Physiological Function
Bodily Location	location-of	Biologic Function
Anatomical Structure	part-of	Organism
Pharmacologic Substance	causes	Pathological Function
Pharmacologic Substance	treats	Pathologic Function

Given a medical sentence like this one:

(17.1) Doppler echocardiography can be used to diagnose left anterior descending artery stenosis in patients with type 2 diabetes

We could thus extract the UMLS relation:

Echocardiography, Doppler Diagnoses *Acquired stenosis*

infoboxes Wikipedia also offers a large supply of relations, drawn from **infoboxes**, structured tables associated with certain Wikipedia articles. For example, the Wikipedia

infobox for **Stanford** includes structured facts like `state = "California"` or `president = "Mark Tessier-Lavigne"`. These facts can be turned into relations like *president-of* or *located-in*. or into relations in a metalanguage called **RDF** (Resource Description Framework). An **RDF triple** is a tuple of entity-relation-entity, called a subject-predicate-object expression. Here's a sample RDF triple:

RDF

RDF triple

subject	**predicate**	**object**
Golden Gate Park	location	San Francisco

For example the crowdsourced DBpedia (Bizer et al., 2009) is an ontology derived from Wikipedia containing over 2 billion RDF triples. Another dataset from Wikipedia infoboxes, **Freebase** (Bollacker et al., 2008), has relations like

Freebase

people/person/nationality
location/location/contains

WordNet or other ontologies offer useful ontological relations that express hierarchical relations between words or concepts. For example WordNet has the **is-a** or **hypernym** relation between classes,

is-a

hypernym

Giraffe is-a ruminant is-a ungulate is-a mammal is-a vertebrate ...

WordNet also has *Instance-of* relation between individuals and classes, so that for example *San Francisco* is in the *Instance-of* relation with *city*. Extracting these relations is an important step in extending or building ontologies.

There are five main classes of algorithms for relation extraction: **hand-written patterns**, **supervised machine learning**, **semi-supervised** (via **bootstrapping** and via **distant supervision**), and **unsupervised**. We'll introduce each of these in the next sections.

17.2.1 Using Patterns to Extract Relations

The earliest and still common algorithm for relation extraction is lexico-syntactic patterns, first developed by Hearst (1992a). Consider the following sentence:

> Agar is a substance prepared from a mixture of red algae, such as Gelidium, for laboratory or industrial use.

Hearst points out that most human readers will not know what *Gelidium* is, but that they can readily infer that it is a kind of (a **hyponym** of) *red algae*, whatever that is. She suggests that the following **lexico-syntactic pattern**

$$NP_0 \textit{ such as } NP_1\{,NP_2\ldots,(and|or)NP_i\}, i \geq 1 \tag{17.2}$$

implies the following semantics

$$\forall NP_i, i \geq 1, \text{hyponym}(NP_i, NP_0) \tag{17.3}$$

allowing us to infer

$$\text{hyponym}(\text{Gelidium}, \text{red algae}) \tag{17.4}$$

Figure 17.12 shows five patterns Hearst (1992a, 1998) suggested for inferring the hyponym relation; we've shown NP_H as the parent/hyponym. Modern versions of the pattern-based approach extend it by adding named entity constraints. For example if our goal is to answer questions about "Who holds what office in which organization?", we can use patterns like the following:

NP {, NP}* {,} (and|or) other NP_H	temples, treasuries, and other important civic buildings
NP_H such as {NP,}* {(or|and)} NP	red algae such as Gelidium
such NP_H as {NP,}* {(or|and)} NP	such authors as Herrick, Goldsmith, and Shakespeare
NP_H {,} including {NP,}* {(or|and)} NP	common-law countries, including Canada and England
NP_H {,} especially {NP}* {(or|and)} NP	European countries, especially France, England, and Spain

Figure 17.12 Hand-built lexico-syntactic patterns for finding hypernyms, using {} to mark optionality (Hearst 1992a, Hearst 1998).

PER, POSITION of ORG:
George Marshall, Secretary of State of the United States

PER (named|appointed|chose|etc.) PER Prep? POSITION
Truman appointed Marshall Secretary of State

PER [be]? (named|appointed|etc.) Prep? ORG POSITION
George Marshall was named US Secretary of State

Hand-built patterns have the advantage of high-precision and they can be tailored to specific domains. On the other hand, they are often low-recall, and it's a lot of work to create them for all possible patterns.

17.2.2 Relation Extraction via Supervised Learning

Supervised machine learning approaches to relation extraction follow a scheme that should be familiar by now. A fixed set of relations and entities is chosen, a training corpus is hand-annotated with the relations and entities, and the annotated texts are then used to train classifiers to annotate an unseen test set.

The most straightforward approach has three steps, illustrated in Fig. 17.13. Step one is to find pairs of named entities (usually in the same sentence). In step two, a filtering classifier is trained to make a binary decision as to whether a given pair of named entities are related (by any relation). Positive examples are extracted directly from all relations in the annotated corpus, and negative examples are generated from within-sentence entity pairs that are not annotated with a relation. In step 3, a classifier is trained to assign a label to the relations that were found by step 2. The use of the filtering classifier can speed up the final classification and also allows the use of distinct feature-sets appropriate for each task. For each of the two classifiers, we can use any of the standard classification techniques (logistic regression, neural network, SVM, etc.)

```
function FINDRELATIONS(words) returns relations

    relations ← nil
    entities ← FINDENTITIES(words)
    forall entity pairs ⟨e1, e2⟩ in entities do
        if RELATED?(e1, e2)
            relations ← relations+CLASSIFYRELATION(e1, e2)
```

Figure 17.13 Finding and classifying the relations among entities in a text.

For the feature-based classifiers like logistic regression or random forests the most important step is to identify useful features. Let's consider features for clas-

sifying the relationship between *American Airlines* (Mention 1, or M1) and *Tim Wagner* (Mention 2, M2) from this sentence:

(17.5) **American Airlines**, a unit of AMR, immediately matched the move, spokesman **Tim Wagner** said

Useful word features include

- The headwords of M1 and M2 and their concatenation
  Airlines Wagner Airlines-Wagner
- Bag-of-words and bigrams in M1 and M2
  American, Airlines, Tim, Wagner, American Airlines, Tim Wagner
- Words or bigrams in particular positions
  M2: -1 spokesman
  M2: +1 said
- Bag of words or bigrams between M1 and M2:
  a, AMR, of, immediately, matched, move, spokesman, the, unit
- Stemmed versions of the same

Embeddings can be used to represent words in any of these features. Useful named entity features include

- Named-entity types and their concatenation
  (M1: ORG, M2: PER, M1M2: ORG-PER)
- Entity Level of M1 and M2 (from the set NAME, NOMINAL, PRONOUN)
  M1: NAME [it or he would be PRONOUN]
  M2: NAME [the company would be NOMINAL]
- Number of entities between the arguments (in this case 1, for AMR)

The **syntactic structure** of a sentence can also signal relationships among its entities. Syntax is often featured by using strings representing **syntactic paths**: the (dependency or constituency) path traversed through the tree in getting from one entity to the other.

- Base syntactic chunk sequence from M1 to M2
  NP NP PP VP NP NP
- Constituent paths between M1 and M2
  $NP \uparrow NP \uparrow S \uparrow S \downarrow NP$
- Dependency-tree paths
  $Airlines \leftarrow_{subj} matched \leftarrow_{comp} said \rightarrow_{subj} Wagner$

Figure 17.14 summarizes many of the features we have discussed that could be used for classifying the relationship between *American Airlines* and *Tim Wagner* from our example text.

Neural models for relation extraction similarly treat the task as supervised classification. One option is to use a similar architecture as we saw for named entity tagging: a bi-LSTM model with word embeddings as inputs and a single softmax classification of the sentence output as a 1-of-N relation label. Because relations often hold between entities that are far part in a sentence (or across sentences), it may be possible to get higher performance from algorithms like convolutional nets (dos Santos et al., 2015) or chain or tree LSTMS (Miwa and Bansal 2016, Peng et al. 2017).

In general, if the test set is similar enough to the training set, and if there is enough hand-labeled data, supervised relation extraction systems can get high accuracies. But labeling a large training set is extremely expensive and supervised

M1 headword	*airlines* (as a word token or an embedding)
M2 headword	*Wagner*
Word(s) before M1	NONE
Word(s) after M2	*said*
Bag of words between	{*a, unit, of, AMR, Inc., immediately, matched, the, move, spokesman* }
M1 type	ORG
M2 type	PERS
Concatenated types	ORG-PERS
Constituent path	$NP \uparrow NP \uparrow S \uparrow S \downarrow NP$
Base phrase path	$NP \rightarrow NP \rightarrow PP \rightarrow NP \rightarrow VP \rightarrow NP \rightarrow NP$
Typed-dependency path	$Airlines \leftarrow_{subj} matched \leftarrow_{comp} said \rightarrow_{subj} Wagner$

Figure 17.14 Sample of features extracted during classification of the <American Airlines, Tim Wagner> tuple; M1 is the first mention, M2 the second.

models are brittle: they don't generalize well to different text genres. For this reason, much research in relation extraction has focused on the semi-supervised and unsupervised approaches we turn to next.

17.2.3 Semisupervised Relation Extraction via Bootstrapping

seed patterns
seed tuples
bootstrapping

Supervised machine learning assumes that we have lots of labeled data. Unfortunately, this is expensive. But suppose we just have a few high-precision **seed patterns**, like those in Section 17.2.1, or perhaps a few **seed tuples**. That's enough to bootstrap a classifier! **Bootstrapping** proceeds by taking the entities in the seed pair, and then finding sentences (on the web, or whatever dataset we are using) that contain both entities. From all such sentences, we extract and generalize the context around the entities to learn new patterns. Fig. 17.15 sketches a basic algorithm.

```
function BOOTSTRAP(Relation R) returns new relation tuples

  tuples ← Gather a set of seed tuples that have relation R
  iterate
     sentences ← find sentences that contain entities in tuples
     patterns ← generalize the context between and around entities in sentences
     newpairs ← use patterns to grep for more tuples
     newpairs ← newpairs with high confidence
     tuples ← tuples + newpairs
  return tuples
```

Figure 17.15 Bootstrapping from seed entity pairs to learn relations.

Suppose, for example, that we need to create a list of airline/hub pairs, and we know only that Ryanair has a hub at Charleroi. We can use this seed fact to discover new patterns by finding other mentions of this relation in our corpus. We search for the terms *Ryanair*, *Charleroi* and *hub* in some proximity. Perhaps we find the following set of sentences:

(17.6) Budget airline Ryanair, which uses Charleroi as a hub, scrapped all weekend flights out of the airport.

(17.7) All flights in and out of Ryanair's Belgian hub at Charleroi airport were grounded on Friday...

(17.8) A spokesman at Charleroi, a main hub for Ryanair, estimated that 8000 passengers had already been affected.

From these results, we can use the context of words between the entity mentions, the words before mention one, the word after mention two, and the named entity types of the two mentions, and perhaps other features, to extract general patterns such as the following:

```
/ [ORG], which uses [LOC] as a hub /
/ [ORG]'s hub at [LOC] /
/ [LOC] a main hub for [ORG] /
```

These new patterns can then be used to search for additional tuples.

confidence values
semantic drift

Bootstrapping systems also assign **confidence values** to new tuples to avoid **semantic drift**. In semantic drift, an erroneous pattern leads to the introduction of erroneous tuples, which, in turn, lead to the creation of problematic patterns and the meaning of the extracted relations 'drifts'. Consider the following example:

(17.9) Sydney has a ferry hub at Circular Quay.

If accepted as a positive example, this expression could lead to the incorrect introduction of the tuple $\langle Sydney, CircularQuay \rangle$. Patterns based on this tuple could propagate further errors into the database.

Confidence values for patterns are based on balancing two factors: the pattern's performance with respect to the current set of tuples and the pattern's productivity in terms of the number of matches it produces in the document collection. More formally, given a document collection $\mathscr{D}$, a current set of tuples T, and a proposed pattern p, we need to track two factors:

- $hits$: the set of tuples in T that p matches while looking in $\mathscr{D}$
- $finds$: The total set of tuples that p finds in $\mathscr{D}$

The following equation balances these considerations (Riloff and Jones, 1999).

$$Conf_{RlogF}(p) = \frac{hits_p}{finds_p} \times log(finds_p) \tag{17.10}$$

This metric is generally normalized to produce a probability.

noisy-or

We can assess the confidence in a proposed new tuple by combining the evidence supporting it from all the patterns P' that match that tuple in $\mathscr{D}$ (Agichtein and Gravano, 2000). One way to combine such evidence is the **noisy-or** technique. Assume that a given tuple is supported by a subset of the patterns in P, each with its own confidence assessed as above. In the noisy-or model, we make two basic assumptions. First, that for a proposed tuple to be false, *all* of its supporting patterns must have been in error, and second, that the sources of their individual failures are all independent. If we loosely treat our confidence measures as probabilities, then the probability of any individual pattern p failing is $1 - Conf(p)$; the probability of all of the supporting patterns for a tuple being wrong is the product of their individual failure probabilities, leaving us with the following equation for our confidence in a new tuple.

$$Conf(t) = 1 - \prod_{p \in P'} (1 - Conf(p)) \tag{17.11}$$

Setting conservative confidence thresholds for the acceptance of new patterns and tuples during the bootstrapping process helps prevent the system from drifting away from the targeted relation.

17.2.4 Distant Supervision for Relation Extraction

distant supervision

Although text that has been hand-labeled with relation labels is extremely expensive to produce, there are ways to find indirect sources of training data. The **distant supervision** method of Mintz et al. (2009) combines the advantages of bootstrapping with supervised learning. Instead of just a handful of seeds, distant supervision uses a large database to acquire a huge number of seed examples, creates lots of noisy pattern features from all these examples and then combines them in a supervised classifier.

For example suppose we are trying to learn the *place-of-birth* relationship between people and their birth cities. In the seed-based approach, we might have only 5 examples to start with. But Wikipedia-based databases like DBPedia or Freebase have tens of thousands of examples of many relations; including over 100,000 examples of *place-of-birth*, (<Edwin Hubble, Marshfield>, <Albert Einstein, Ulm>, etc.,). The next step is to run named entity taggers on large amounts of text—Mintz et al. (2009) used 800,000 articles from Wikipedia—and extract all sentences that have two named entities that match the tuple, like the following:

...Hubble was born in Marshfield...
...Einstein, born (1879), Ulm...
...Hubble's birthplace in Marshfield...

Training instances can now be extracted from this data, one training instance for each identical tuple <relation, entity1, entity2>. Thus there will be one training instance for each of:

```
<born-in, Edwin Hubble, Marshfield>
<born-in, Albert Einstein, Ulm>
<born-year, Albert Einstein, 1879>
```

and so on.

We can then apply feature-based or neural classification. For feature-based classification, standard supervised relation extraction features like the named entity labels of the two mentions, the words and dependency paths in between the mentions, and neighboring words. Each tuple will have features collected from many training instances; the feature vector for a single training instance like (<born-in,Albert Einstein, Ulm> will have lexical and syntactic features from many different sentences that mention Einstein and Ulm.

Because distant supervision has very large training sets, it is also able to use very rich features that are conjunctions of these individual features. So we will extract thousands of patterns that conjoin the entity types with the intervening words or dependency paths like these:

PER was born in LOC
PER, born (XXXX), LOC
PER's birthplace in LOC

To return to our running example, for this sentence:

(17.12) **American Airlines**, a unit of AMR, immediately matched the move, spokesman **Tim Wagner** said

we would learn rich conjunction features like this one:

M1 = ORG & M2 = PER & nextword="said"& path= $NP \uparrow NP \uparrow S \uparrow S \downarrow NP$

The result is a supervised classifier that has a huge rich set of features to use in detecting relations. Since not every test sentence will have one of the training

relations, the classifier will also need to be able to label an example as *no-relation*. This label is trained by randomly selecting entity pairs that do not appear in any Freebase relation, extracting features for them, and building a feature vector for each such tuple. The final algorithm is sketched in Fig. 17.16.

```
function DISTANT SUPERVISION(Database D, Text T) returns relation classifier C

  foreach relation R
    foreach tuple (e1,e2) of entities with relation R in D
      sentences←Sentences in T that contain e1 and e2
      f←Frequent features in sentences
      observations←observations + new training tuple (e1, e2, f, R)
  C←Train supervised classifier on observations
  return C
```

Figure 17.16 The distant supervision algorithm for relation extraction. A neural classifier might not need to use the feature set f.

Distant supervision shares advantages with each of the methods we've examined. Like supervised classification, distant supervision uses a classifier with lots of features, and supervised by detailed hand-created knowledge. Like pattern-based classifiers, it can make use of high-precision evidence for the relation between entities. Indeed, distance supervision systems learn patterns just like the hand-built patterns of early relation extractors. For example the *is-a* or *hypernym* extraction system of Snow et al. (2005) used hypernym/hyponym NP pairs from WordNet as distant supervision, and then learned new patterns from large amounts of text. Their system induced exactly the original 5 template patterns of Hearst (1992a), but also 70,000 additional patterns including these four:

NP_H like NP	*Many hormones like leptin...*
NP_H called NP	*...using a markup language called XHTML*
NP is a NP_H	Ruby is a programming language...
NP, a NP_H	*IBM, a company with a long...*

This ability to use a large number of features simultaneously means that, unlike the iterative expansion of patterns in seed-based systems, there's no semantic drift. Like unsupervised classification, it doesn't use a labeled training corpus of texts, so it isn't sensitive to genre issues in the training corpus, and relies on very large amounts of unlabeled data. Distant supervision also has the advantage that it can create training tuples to be used with neural classifiers, where features are not required.

But distant supervision can only help in extracting relations for which a large enough database already exists. To extract new relations without datasets, or relations for new domains, purely unsupervised methods must be used.

17.2.5 Unsupervised Relation Extraction

The goal of unsupervised relation extraction is to extract relations from the web when we have no labeled training data, and not even any list of relations. This task is often called **open information extraction** or **Open IE**. In Open IE, the relations are simply strings of words (usually beginning with a verb).

open information extraction

For example, the **ReVerb** system (Fader et al., 2011) extracts a relation from a sentence s in 4 steps:

1. Run a part-of-speech tagger and entity chunker over s
2. For each verb in s, find the longest sequence of words w that start with a verb and satisfy syntactic and lexical constraints, merging adjacent matches.
3. For each phrase w, find the nearest noun phrase x to the left which is not a relative pronoun, wh-word or existential "there". Find the nearest noun phrase y to the right.
4. Assign confidence c to the relation $r = (x, w, y)$ using a confidence classifier and return it.

A relation is only accepted if it meets syntactic and lexical constraints. The syntactic constraints ensure that it is a verb-initial sequence that might also include nouns (relations that begin with light verbs like *make*, *have*, or *do* often express the core of the relation with a noun, like *have a hub in*):

V | VP | VW*P
V = verb particle? adv?
W = (noun | adj | adv | pron | det)
P = (prep | particle | inf. marker)

The lexical constraints are based on a dictionary D that is used to prune very rare, long relation strings. The intuition is to eliminate candidate relations that don't occur with sufficient number of distinct argument types and so are likely to be bad examples. The system first runs the above relation extraction algorithm offline on 500 million web sentences and extracts a list of all the relations that occur after normalizing them (removing inflection, auxiliary verbs, adjectives, and adverbs). Each relation r is added to the dictionary if it occurs with at least 20 different arguments. Fader et al. (2011) used a dictionary of 1.7 million normalized relations.

Finally, a confidence value is computed for each relation using a logistic regression classifier. The classifier is trained by taking 1000 random web sentences, running the extractor, and hand labelling each extracted relation as correct or incorrect. A confidence classifier is then trained on this hand-labeled data, using features of the relation and the surrounding words. Fig. 17.17 shows some sample features used in the classification.

(x,r,y) covers all words in s
the last preposition in r is *for*
the last preposition in r is *on*
len(s) $\leq$ 10
there is a coordinating conjunction to the left of r in s
r matches a lone V in the syntactic constraints
there is preposition to the left of x in s
there is an NP to the right of y in s

Figure 17.17 Features for the classifier that assigns confidence to relations extracted by the Open Information Extraction system REVERB (Fader et al., 2011).

For example the following sentence:

(17.13) United has a hub in Chicago, which is the headquarters of United Continental Holdings.

has the relation phrases *has a hub in* and *is the headquarters of* (it also has *has* and *is*, but longer phrases are preferred). Step 3 finds *United* to the left and *Chicago* to the right of *has a hub in*, and skips over *which* to find Chicago to the left of *is the headquarters of*. The final output is:

```
r1: <United, has a hub in, Chicago>
r2: <Chicago, is the headquarters of, United Continental Holdings>
```

The great advantage of unsupervised relation extraction is its ability to handle a huge number of relations without having to specify them in advance. The disadvantage is the need to map these large sets of strings into some canonical form for adding to databases or other knowledge sources. Current methods focus heavily on relations expressed with verbs, and so will miss many relations that are expressed nominally.

17.2.6 Evaluation of Relation Extraction

Supervised relation extraction systems are evaluated by using test sets with human-annotated, gold-standard relations and computing precision, recall, and F-measure. Labeled precision and recall require the system to classify the relation correctly, whereas unlabeled methods simply measure a system's ability to detect entities that are related.

Semi-supervised and **unsupervised** methods are much more difficult to evaluate, since they extract totally new relations from the web or a large text. Because these methods use very large amounts of text, it is generally not possible to run them solely on a small labeled test set, and as a result it's not possible to pre-annotate a gold set of correct instances of relations.

For these methods it's possible to approximate (only) precision by drawing a random sample of relations from the output, and having a human check the accuracy of each of these relations. Usually this approach focuses on the **tuples** to be extracted from a body of text rather than on the relation **mentions**; systems need not detect every mention of a relation to be scored correctly. Instead, the evaluation is based on the set of tuples occupying the database when the system is finished. That is, we want to know if the system can discover that Ryanair has a hub at Charleroi; we don't really care how many times it discovers it. The estimated precision $\hat{P}$ is then

$$\hat{P} = \frac{\text{\# of correctly extracted relation tuples in the sample}}{\text{total \# of extracted relation tuples in the sample.}} \tag{17.14}$$

Another approach that gives us a little bit of information about recall is to compute precision at different levels of recall. Assuming that our system is able to rank the relations it produces (by probability, or confidence) we can separately compute precision for the top 1000 new relations, the top 10,000 new relations, the top 100,000, and so on. In each case we take a random sample of that set. This will show us how the precision curve behaves as we extract more and more tuples. But there is no way to directly evaluate recall.

17.3 Extracting Times

Times and dates are a particularly important kind of named entity that play a role in question answering, in calendar and personal assistant applications. In order to reason about times and dates, after we extract these **temporal expressions** they must be **normalized**—converted to a standard format so we can reason about them. In this section we consider both the extraction and normalization of temporal expressions.

17.3.1 Temporal Expression Extraction

absolute
relative
duration

Temporal expressions are those that refer to absolute points in time, relative times, durations, and sets of these. **Absolute** temporal expressions are those that can be mapped directly to calendar dates, times of day, or both. **Relative** temporal expressions map to particular times through some other reference point (as in *a week from last Tuesday*). Finally, **durations** denote spans of time at varying levels of granularity (seconds, minutes, days, weeks, centuries, etc.). Figure 17.18 lists some sample temporal expressions in each of these categories.

Absolute	Relative	Durations
April 24, 1916	yesterday	four hours
The summer of '77	next semester	three weeks
10:15 AM	two weeks from yesterday	six days
The 3rd quarter of 2006	last quarter	the last three quarters

Figure 17.18 Examples of absolute, relational and durational temporal expressions.

lexical triggers

Temporal expressions are grammatical constructions that have temporal **lexical triggers** as their heads. Lexical triggers might be nouns, proper nouns, adjectives, and adverbs; full temporal expressions consist of their phrasal projections: noun phrases, adjective phrases, and adverbial phrases. Figure 17.19 provides examples.

Category	Examples
Noun	*morning*, *noon*, *night*, *winter*, *dusk*, *dawn*
Proper Noun	*January, Monday, Ides, Easter, Rosh Hashana, Ramadan, Tet*
Adjective	*recent, past, annual, former*
Adverb	*hourly, daily, monthly, yearly*

Figure 17.19 Examples of temporal lexical triggers.

Let's look at the TimeML annotation scheme, in which temporal expressions are annotated with an XML tag, TIMEX3, and various attributes to that tag (Pustejovsky et al. 2005, Ferro et al. 2005). The following example illustrates the basic use of this scheme (we defer discussion of the attributes until Section 17.3.2).

> A fare increase initiated <TIMEX3>last week</TIMEX3> by UAL Corp's United Airlines was matched by competitors over <TIMEX3>the weekend</TIMEX3>, marking the second successful fare increase in <TIMEX3>two weeks</TIMEX3>.

The temporal expression recognition task consists of finding the start and end of all of the text spans that correspond to such temporal expressions. **Rule-based approaches** to temporal expression recognition use cascades of automata to recognize patterns at increasing levels of complexity. Tokens are first part-of-speech tagged, and then larger and larger chunks are recognized from the results from previous stages, based on patterns containing trigger words (e.g., *February*) or classes (e.g., *MONTH*). Figure 17.20 gives a fragment from a rule-based system.

Sequence-labeling approaches follow the same IOB scheme used for named-entity tags, marking words that are either inside, outside or at the beginning of a TIMEX3-delimited temporal expression with the I, O, and B tags as follows:

A	*fare*	*increase*	*initiated*	*last*	*week*	*by*	*UAL*	*Corp's...*
O	O	O	O	B	I	O	O	O

```
# yesterday/today/tomorrow
$string =˜ s/((($OT+the$CT+\s+)?$OT+day$CT+\s+$OT+(before|after)$CT+\s+)?$OT+$TERelDayExpr$CT+
(\s+$OT+(morning|afternoon|evening|night)$CT+)?)/<TIMEX$tever TYPE=\"DATE\">$1
<\/TIMEX$tever>/gio;

$string =˜ s/($OT+\w+$CT+\s+)<TIMEX$tever TYPE=\"DATE\"[ˆ>]*>($OT+(Today|Tonight)$CT+)
<\/TIMEX$tever>/$1$4/gso;

# this (morning/afternoon/evening)
$string =˜ s/(($OT+(early|late)$CT+\s+)?$OT+this$CT+\s*$OT+(morning|afternoon|evening)$CT+)/
  <TIMEX$tever TYPE=\"DATE\">$1<\/TIMEX$tever>/gosi;
$string =˜ s/(($OT+(early|late)$CT+\s+)?$OT+last$CT+\s*$OT+night$CT+)/<TIMEX$tever
   TYPE=\"DATE\">$1<\/TIMEX$tever>/gsio;
```

Figure 17.20 Perl fragment from the GUTime temporal tagging system in Tarsqi (Verhagen et al., 2005).

Features are extracted from the token and its context, and a statistical sequence labeler is trained (any sequence model can be used). Figure 17.21 lists standard features used in temporal tagging.

Feature	Explanation
Token	The target token to be labeled
Tokens in window	Bag of tokens in the window around a target
Shape	Character shape features
POS	Parts of speech of target and window words
Chunk tags	Base-phrase chunk tag for target and words in a window
Lexical triggers	Presence in a list of temporal terms

Figure 17.21 Typical features used to train IOB-style temporal expression taggers.

Temporal expression recognizers are evaluated with the usual recall, precision, and *F*-measures. A major difficulty for all of these very lexicalized approaches is avoiding expressions that trigger false positives:

(17.15) *1984* tells the story of Winston Smith...

(17.16) ...U2's classic *Sunday* Bloody *Sunday*

17.3.2 Temporal Normalization

temporal normalization

Temporal normalization is the process of mapping a temporal expression to either a specific point in time or to a duration. Points in time correspond to calendar dates, to times of day, or both. Durations primarily consist of lengths of time but may also include information about start and end points. Normalized times are represented with the VALUE attribute from the ISO 8601 standard for encoding temporal values (ISO8601, 2004). Fig. 17.22 reproduces our earlier example with the value attributes added in.

```
<TIMEX3 id=''t1'' type="DATE" value="2007-07-02" functionInDocument="CREATION_TIME"
    > July 2, 2007 </TIMEX3> A fare increase initiated <TIMEX3 id="t2" type="DATE"
    value="2007-W26" anchorTimeID="t1">last week</TIMEX3> by United Airlines was
    matched by competitors over <TIMEX3 id="t3" type="DURATION" value="P1WE"
    anchorTimeID="t1"> the weekend </TIMEX3>, marking the second successful fare
    increase in <TIMEX3 id="t4" type="DURATION" value="P2W" anchorTimeID="t1"> two
    weeks </TIMEX3>.
```

Figure 17.22 TimeML markup including normalized values for temporal expressions.

The dateline, or document date, for this text was *July 2, 2007*. The ISO representation for this kind of expression is YYYY-MM-DD, or in this case, 2007-07-02.

The encodings for the temporal expressions in our sample text all follow from this date, and are shown here as values for the VALUE attribute.

The first temporal expression in the text proper refers to a particular week of the year. In the ISO standard, weeks are numbered from 01 to 53, with the first week of the year being the one that has the first Thursday of the year. These weeks are represented with the template YYYY-Wnn. The ISO week for our document date is week 27; thus the value for *last week* is represented as "2007-W26".

The next temporal expression is *the weekend*. ISO weeks begin on Monday; thus, weekends occur at the end of a week and are fully contained within a single week. Weekends are treated as durations, so the value of the VALUE attribute has to be a length. Durations are represented according to the pattern Pnx, where n is an integer denoting the length and x represents the unit, as in P3Y for *three years* or P2D for *two days*. In this example, one weekend is captured as P1WE. In this case, there is also sufficient information to anchor this particular weekend as part of a particular week. Such information is encoded in the ANCHORTIMEID attribute. Finally, the phrase *two weeks* also denotes a duration captured as P2W. There is a lot more to the various temporal annotation standards—far too much to cover here. Figure 17.23 describes some of the basic ways that other times and durations are represented. Consult ISO8601 (2004), Ferro et al. (2005), and Pustejovsky et al. (2005) for more details.

Unit	Pattern	Sample Value
Fully specified dates	YYYY-MM-DD	1991-09-28
Weeks	YYYY-Wnn	2007-W27
Weekends	PnWE	P1WE
24-hour clock times	HH:MM:SS	11:13:45
Dates and times	YYYY-MM-DDTHH:MM:SS	1991-09-28T11:00:00
Financial quarters	Qn	1999-Q3

Figure 17.23 Sample ISO patterns for representing various times and durations.

Most current approaches to temporal normalization are rule-based (Chang and Manning 2012, Strötgen and Gertz 2013). Patterns that match temporal expressions are associated with semantic analysis procedures. As in the compositional rule-to-rule approach introduced in Chapter 15, the meaning of a constituent is computed from the meaning of its parts using a method specific to the constituent, although here the semantic composition rules involve temporal arithmetic rather than λ-calculus attachments.

fully qualified

Fully qualified date expressions contain a year, month, and day in some conventional form. The units in the expression must be detected and then placed in the correct place in the corresponding ISO pattern. The following pattern normalizes expressions like *April 24, 1916*.

$$FQTE \rightarrow Month\ Date\ ,\ Year \qquad \{Year.val - Month.val - Date.val\}$$

The non-terminals *Month*, *Date*, and *Year* represent constituents that have already been recognized and assigned semantic values, accessed through the **.val* notation. The value of this *FQE* constituent can, in turn, be accessed as *FQTE.val* during further processing.

Fully qualified temporal expressions are fairly rare in real texts. Most temporal expressions in news articles are incomplete and are only implicitly anchored, often with respect to the dateline of the article, which we refer to as the document's

temporal anchor

temporal anchor. The values of temporal expressions such as *today*, *yesterday*, or *tomorrow* can all be computed with respect to this temporal anchor. The semantic procedure for *today* simply assigns the anchor, and the attachments for *tomorrow* and *yesterday* add a day and subtract a day from the anchor, respectively. Of course, given the cyclic nature of our representations for months, weeks, days, and times of day, our temporal arithmetic procedures must use modulo arithmetic appropriate to the time unit being used.

Unfortunately, even simple expressions such as *the weekend* or *Wednesday* introduce a fair amount of complexity. In our current example, *the weekend* clearly refers to the weekend of the week that immediately precedes the document date. But this won't always be the case, as is illustrated in the following example.

(17.17) Random security checks that began yesterday at Sky Harbor will continue at least through the weekend.

In this case, the expression *the weekend* refers to the weekend of the week that the anchoring date is part of (i.e., the coming weekend). The information that signals this meaning comes from the tense of *continue*, the verb governing *the weekend*.

Relative temporal expressions are handled with temporal arithmetic similar to that used for *today* and *yesterday*. The document date indicates that our example article is ISO week 27, so the expression *last week* normalizes to the current week minus 1. To resolve ambiguous *next* and *last* expressions we consider the distance from the anchoring date to the nearest unit. *Next Friday* can refer either to the immediately next Friday or to the Friday following that, but the closer the document date is to a Friday, the more likely it is that the phrase will skip the nearest one. Such ambiguities are handled by encoding language and domain-specific heuristics into the temporal attachments.

17.4 Extracting Events and their Times

event extraction

The task of **event extraction** is to identify mentions of events in texts. For the purposes of this task, an event mention is any expression denoting an event or state that can be assigned to a particular point, or interval, in time. The following markup of the sample text on page 294 shows all the events in this text.

> [$_{\text{EVENT}}$ Citing] high fuel prices, United Airlines [$_{\text{EVENT}}$ said] Friday it has [$_{\text{EVENT}}$ increased] fares by $6 per round trip on flights to some cities also served by lower-cost carriers. American Airlines, a unit of AMR Corp., immediately [$_{\text{EVENT}}$ matched] [$_{\text{EVENT}}$ the move], spokesman Tim Wagner [$_{\text{EVENT}}$ said]. United, a unit of UAL Corp., [$_{\text{EVENT}}$ said] [$_{\text{EVENT}}$ the increase] took effect Thursday and [$_{\text{EVENT}}$ applies] to most routes where it [$_{\text{EVENT}}$ competes] against discount carriers, such as Chicago to Dallas and Denver to San Francisco.

In English, most event mentions correspond to verbs, and most verbs introduce events. However, as we can see from our example, this is not always the case. Events can be introduced by noun phrases, as in *the move* and *the increase*, and some verbs fail to introduce events, as in the phrasal verb *took effect*, which refers to when the event began rather than to the event itself. Similarly, light verbs such as *make*, *take*, and *have* often fail to denote events; for light verbs the event is often expressed by the nominal direct object (*took a flight*), and these light verbs just provide a syntactic structure for the noun's arguments.

reporting events

Various versions of the event extraction task exist, depending on the goal. For example in the TempEval shared tasks (Verhagen et al. 2009) the goal is to extract events and aspects like their aspectual and temporal properties. Events are to be classified as actions, states, **reporting events** (*say, report, tell, explain*), perception events, and so on. The aspect, tense, and modality of each event also needs to be extracted. Thus for example the various *said* events in the sample text would be annotated as (class=REPORTING, tense=PAST, aspect=PERFECTIVE).

Event extraction is generally modeled via supervised learning, detecting events via sequence models with IOB tagging, and assigning event classes and attributes with multi-class classifiers. Common features include surface information like parts of speech, lexical items, and verb tense information; see Fig. 17.24.

Feature	**Explanation**
Character affixes	Character-level prefixes and suffixes of target word
Nominalization suffix	Character level suffixes for nominalizations (e.g., *-tion*)
Part of speech	Part of speech of the target word
Light verb	Binary feature indicating that the target is governed by a light verb
Subject syntactic category	Syntactic category of the subject of the sentence
Morphological stem	Stemmed version of the target word
Verb root	Root form of the verb basis for a nominalization
WordNet hypernyms	Hypernym set for the target

Figure 17.24 Features commonly used in both rule-based and machine learning approaches to event detection.

17.4.1 Temporal Ordering of Events

With both the events and the temporal expressions in a text having been detected, the next logical task is to use this information to fit the events into a complete timeline. Such a timeline would be useful for applications such as question answering and summarization. This ambitious task is the subject of considerable current research but is beyond the capabilities of current systems.

A somewhat simpler, but still useful, task is to impose a partial ordering on the events and temporal expressions mentioned in a text. Such an ordering can provide many of the same benefits as a true timeline. An example of such a partial ordering is the determination that the fare increase by *American Airlines* came *after* the fare increase by *United* in our sample text. Determining such an ordering can be viewed as a binary relation detection and classification task similar to those described earlier in Section 17.2. The temporal relation between events is classified into one of the standard set of **Allen relations** shown in Fig. 17.25 (Allen, 1984), using feature-based classifiers as in Section 17.2, trained on the TimeBank corpus with features like words/embeddings, parse paths, tense and aspect.

Allen relations

TimeBank

The **TimeBank** corpus consists of text annotated with much of the information we've been discussing throughout this section (Pustejovsky et al., 2003b). TimeBank 1.2 consists of 183 news articles selected from a variety of sources, including the Penn TreeBank and PropBank collections.

Each article in the TimeBank corpus has had the temporal expressions and event mentions in them explicitly annotated in the TimeML annotation (Pustejovsky et al., 2003a). In addition to temporal expressions and events, the TimeML annotation provides temporal links between events and temporal expressions that specify the nature of the relation between them. Consider the following sample sentence and

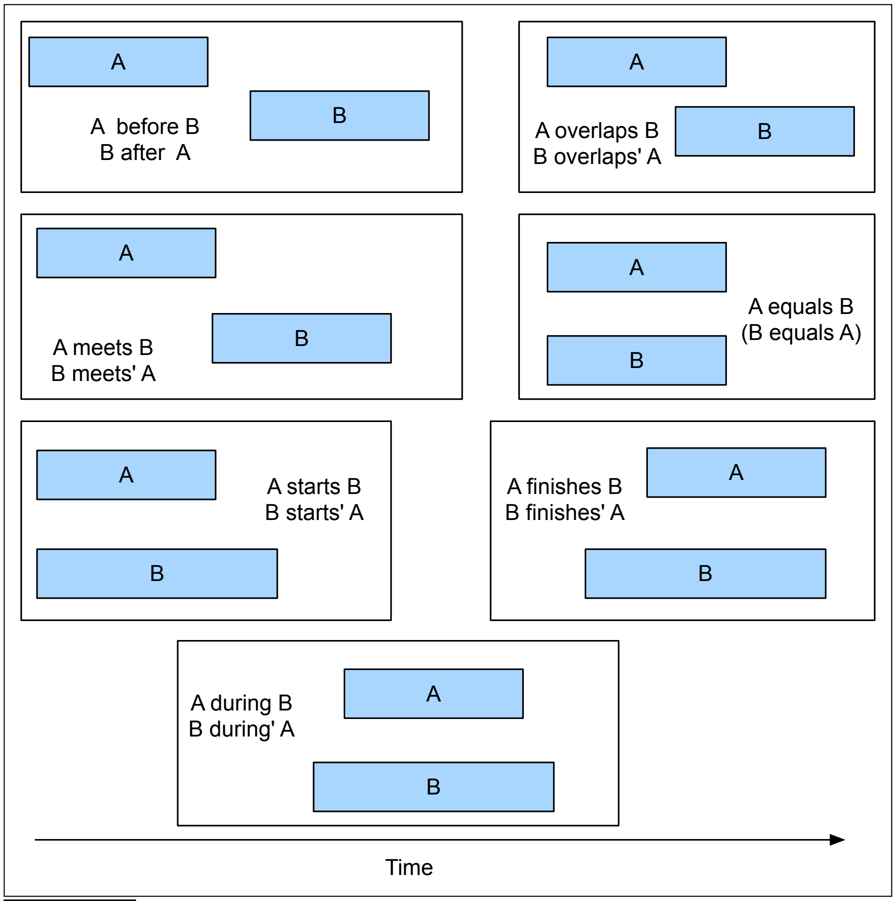


Figure 17.25 The 13 temporal relations from Allen (1984).

```
<TIMEX3 tid="t57" type="DATE" value="1989-10-26"  functionInDocument="CREATION_TIME">
10/26/89  </TIMEX3>

Delta Air Lines earnings <EVENT eid="e1" class="OCCURRENCE"> soared </EVENT> 33% to a
record in  <TIMEX3 tid="t58" type="DATE" value="1989-Q1" anchorTimeID="t57"> the
fiscal first quarter </TIMEX3>, <EVENT eid="e3"  class="OCCURRENCE">bucking</EVENT>
the industry trend toward <EVENT eid="e4" class="OCCURRENCE">declining</EVENT>
profits.
```

Figure 17.26 Example from the TimeBank corpus.

its corresponding markup shown in Fig. 17.26, selected from one of the TimeBank documents.

(17.18) Delta Air Lines earnings soared 33% to a record in the fiscal first quarter, bucking the industry trend toward declining profits.

As annotated, this text includes three events and two temporal expressions. The events are all in the occurrence class and are given unique identifiers for use in further annotations. The temporal expressions include the creation time of the article, which serves as the document time, and a single temporal expression within the text.

In addition to these annotations, TimeBank provides four links that capture the temporal relations between the events and times in the text, using the Allen relations from Fig. 17.25. The following are the within-sentence temporal relations annotated for this example.

- Soaring$_{e1}$ is **included** in the fiscal first quarter$_{t58}$
- Soaring$_{e1}$ is **before** 1989-10-26$_{t57}$
- Soaring$_{e1}$ is **simultaneous** with the bucking$_{e3}$
- Declining$_{e4}$ **includes** soaring$_{e1}$

17.5 Template Filling

Many texts contain reports of events, and possibly sequences of events, that often correspond to fairly common, stereotypical situations in the world. These abstract situations or stories, related to what have been called **scripts** (Schank and Abelson, 1977), consist of prototypical sequences of sub-events, participants, and their roles. The strong expectations provided by these scripts can facilitate the proper classification of entities, the assignment of entities into roles and relations, and most critically, the drawing of inferences that fill in things that have been left unsaid. In their simplest form, such scripts can be represented as **templates** consisting of fixed sets of **slots** that take as values **slot-fillers** belonging to particular classes. The task of **template filling** is to find documents that invoke particular scripts and then fill the slots in the associated templates with fillers extracted from the text. These slot-fillers may consist of text segments extracted directly from the text, or they may consist of concepts that have been inferred from text elements through some additional processing.

scripts

templates

template filling

A filled template from our original airline story might look like the following.

FARE-RAISE ATTEMPT:	LEAD AIRLINE:	UNITED AIRLINES
	AMOUNT:	$6
	EFFECTIVE DATE:	2006-10-26
	FOLLOWER:	AMERICAN AIRLINES

This template has four slots (LEAD AIRLINE, AMOUNT, EFFECTIVE DATE, FOLLOWER). The next section describes a standard sequence-labeling approach to filling slots. Section 17.5.2 then describes an older system based on the use of cascades of finite-state transducers and designed to address a more complex template-filling task that current learning-based systems don't yet address.

17.5.1 Machine Learning Approaches to Template Filling

In the standard paradigm for template filling, we are trying to fill fixed known templates with known slots, and also assumes training documents labeled with examples of each template, and the fillers of each slot marked in the text. The is to create one template for each event in the input documents, with the slots filled with text from the document.

The task is generally modeled by training two separate supervised systems. The first system decides whether the template is present in a particular sentence. This task is called **template recognition** or sometimes, in a perhaps confusing bit of terminology, *event recognition*. Template recognition can be treated as a text classification task, with features extracted from every sequence of words that was labeled in training documents as filling any slot from the template being detected. The usual set of features can be used: tokens, embeddings, word shapes, part-of-speech tags, syntactic chunk tags, and named entity tags.

template recognition

role-filler extraction

The second system has the job of **role-filler extraction**. A separate classifier is trained to detect each role (LEAD-AIRLINE, AMOUNT, and so on). This can be a binary classifier that is run on every noun-phrase in the parsed input sentence, or a sequence model run over sequences of words. Each role classifier is trained on the labeled data in the training set. Again, the usual set of features can be used, but now trained only on an individual noun phrase or the fillers of a single slot.

Multiple non-identical text segments might be labeled with the same slot label. For example in our sample text, the strings *United* or *United Airlines* might be labeled as the LEAD AIRLINE. These are not incompatible choices and the coreference resolution techniques introduced in Chapter 21 can provide a path to a solution.

A variety of annotated collections have been used to evaluate this style of approach to template filling, including sets of job announcements, conference calls for papers, restaurant guides, and biological texts. Recent work focuses on extracting templates in cases where there is no training data or even predefined templates, by inducing templates as sets of linked events (Chambers and Jurafsky, 2011).

17.5.2 Earlier Finite-State Template-Filling Systems

The templates above are relatively simple. But consider the task of producing a template that contained all the information in a text like this one (Grishman and Sundheim, 1995):

> Bridgestone Sports Co. said Friday it has set up a joint venture in Taiwan with a local concern and a Japanese trading house to produce golf clubs to be shipped to Japan. The joint venture, Bridgestone Sports Taiwan Co., capitalized at 20 million new Taiwan dollars, will start production in January 1990 with production of 20,000 iron and "metal wood" clubs a month.

The MUC-5 'joint venture' task (the *Message Understanding Conferences* were a series of U.S. government-organized information-extraction evaluations) was to produce hierarchically linked templates describing joint ventures. Figure 17.27 shows a structure produced by the FASTUS system (Hobbs et al., 1997). Note how the filler of the ACTIVITY slot of the TIE-UP template is itself a template with slots.

Tie-up-1		**Activity-1**:	
RELATIONSHIP	tie-up	COMPANY	Bridgestone Sports Taiwan Co.
ENTITIES	Bridgestone Sports Co.	PRODUCT	iron and "metal wood" clubs
	a local concern	START DATE	DURING: January 1990
	a Japanese trading house		
JOINT VENTURE	Bridgestone Sports Taiwan Co.		
ACTIVITY	Activity-1		
AMOUNT	NT$20000000		

Figure 17.27 The templates produced by FASTUS given the input text on page 301.

Early systems for dealing with these complex templates were based on cascades of transducers based on hand-written rules, as sketched in Fig. 17.28.

The first four stages use hand-written regular expression and grammar rules to do basic tokenization, chunking, and parsing. Stage 5 then recognizes entities and events with a FST-based recognizer and inserts the recognized objects into the appropriate slots in templates. This FST recognizer is based on hand-built regular expressions like the following (NG indicates Noun-Group and VG Verb-Group), which matches the first sentence of the news story above.

No.	Step	Description
1	**Tokens**	Tokenize input stream of characters
2	**Complex Words**	Multiword phrases, numbers, and proper names.
3	**Basic phrases**	Segment sentences into noun and verb groups
4	**Complex phrases**	Identify complex noun groups and verb groups
5	**Semantic Patterns**	Identify entities and events, insert into templates.
6	**Merging**	Merge references to the same entity or event

Figure 17.28 Levels of processing in FASTUS (Hobbs et al., 1997). Each level extracts a specific type of information which is then passed on to the next higher level.

```
NG(Company/ies) VG(Set-up) NG(Joint-Venture) with NG(Company/ies)
VG(Produce) NG(Product)
```

The result of processing these two sentences is the five draft templates (Fig. 17.29) that must then be merged into the single hierarchical structure shown in Fig. 17.27. The merging algorithm, after performing coreference resolution, merges two activities that are likely to be describing the same events.

#	Template/Slot	Value
1	RELATIONSHIP:	TIE-UP
	ENTITIES:	Bridgestone Co., a local concern, a Japanese trading house
2	ACTIVITY:	PRODUCTION
	PRODUCT:	"golf clubs"
3	RELATIONSHIP:	TIE-UP
	JOINT VENTURE:	"Bridgestone Sports Taiwan Co."
	AMOUNT:	NT$20000000
4	ACTIVITY:	PRODUCTION
	COMPANY:	"Bridgestone Sports Taiwan Co."
	STARTDATE:	DURING: January 1990
5	ACTIVITY:	PRODUCTION
	PRODUCT:	"iron and "metal wood" clubs"

Figure 17.29 The five partial templates produced by stage 5 of FASTUS. These templates are merged in stage 6 to produce the final template shown in Fig. 17.27 on page 301.

17.6 Summary

This chapter has explored techniques for extracting limited forms of semantic content from texts.

- **Named entities** can be recognized and classified by featured-based or neural sequence labeling techniques.
- **Relations among entities** can be extracted by pattern-based approaches, supervised learning methods when annotated training data is available, lightly supervised **bootstrapping** methods when small numbers of **seed tuples** or **seed patterns** are available, **distant supervision** when a database of relations is available, and **unsupervised** or **Open IE** methods.
- Reasoning about time can be facilitated by detection and normalization of **temporal expressions** through a combination of statistical learning and rule-

based methods.

- **Events** can be detected and ordered in time using sequence models and classifiers trained on temporally- and event-labeled data like the **TimeBank corpus**.
- **Template-filling** applications can recognize stereotypical situations in texts and assign elements from the text to roles represented as **fixed sets of slots**.

Bibliographical and Historical Notes

The earliest work on information extraction addressed the template-filling task in the context of the Frump system (DeJong, 1982). Later work was stimulated by the U.S. government-sponsored MUC conferences (Sundheim 1991, Sundheim 1992, Sundheim 1993, Sundheim 1995). Early MUC systems like CIRCUS system (Lehnert et al., 1991) and SCISOR (Jacobs and Rau, 1990) were quite influential and inspired later systems like FASTUS (Hobbs et al., 1997). Chinchor et al. (1993) describe the MUC evaluation techniques.

Due to the difficulty of porting systems from one domain to another, attention shifted to machine learning approaches.

Early supervised learning approaches to IE (Cardie 1993, Cardie 1994, Riloff 1993, Soderland et al. 1995, Huffman 1996) focused on automating the knowledge acquisition process, mainly for finite-state rule-based systems. Their success, and the earlier success of HMM-based speech recognition, led to the use of sequence labeling (HMMs: Bikel et al. 1997; MEMMs McCallum et al. 2000; CRFs: Lafferty et al. 2001), and a wide exploration of features (Zhou et al., 2005). Neural approaches to NER mainly follow from the pioneering results of Collobert et al. (2011), who applied a CRF on top of a convolutional net. BiLSTMs with word and character-based embeddings as input followed shortly and became a standard neural algorithm for NER (Huang et al. 2015, Ma and Hovy 2016, Lample et al. 2016).

Neural algorithms for relation extraction often explore architectures that can handle entities far apart in the sentence: recursive networks (Socher et al., 2012), convolutional nets (dos Santos et al., 2015), or chain or tree LSTMS (Miwa and Bansal 2016, Peng et al. 2017).

Progress in this area continues to be stimulated by formal evaluations with shared benchmark datasets, including the Automatic Content Extraction (ACE) evaluations of 2000-2007 on named entity recognition, relation extraction, and temporal expressions[3], the **KBP** (**Knowledge Base Population**) evaluations (Ji et al. 2010, Surdeanu 2013) of relation extraction tasks like **slot filling** (extracting attributes ('slots') like age, birthplace, and spouse for a given entity) and a series of SemEval workshops (Hendrickx et al., 2009).

KBP
slot filling

Semisupervised relation extraction was first proposed by Hearst (1992b), and extended by systems like AutoSlog-TS (?), DIPRE (Brin, 1998), SNOWBALL (Agichtein and Gravano, 2000), and (Jones et al., 1999). The distant supervision algorithm we describe was drawn from Mintz et al. (2009), who coined the term 'distant supervision', but similar ideas occurred in earlier systems like Craven and Kumlien (1999) and Morgan et al. (2004) under the name *weakly labeled data*, as well as in Snow et al. (2005) and Wu and Weld (2007). Among the many extensions are Wu and Weld (2010), Riedel et al. (2010), and Ritter et al. (2013). Open IE systems include

[3] www.nist.gov/speech/tests/ace/

KNOWITALL Etzioni et al. (2005), TextRunner (Banko et al., 2007), and REVERB (Fader et al., 2011). See Riedel et al. (2013) for a universal schema that combines the advantages of distant supervision and Open IE.

HeidelTime (Strötgen and Gertz, 2013) and SUTime (Chang and Manning, 2012) are downloadable temporal extraction and normalization systems. The 2013 TempEval challenge is described in UzZaman et al. (2013); Chambers (2013) and Bethard (2013) give typical approaches.

Exercises

17.1 Develop a set of regular expressions to recognize the character shape features described on page 280.

17.2 The IOB labeling scheme given in this chapter isn't the only possible one. For example, an E tag might be added to mark the end of entities, or the B tag can be reserved only for those situations where an ambiguity exists between adjacent entities. Propose a new set of IOB tags for use with your NER system. Experiment with it and compare its performance with the scheme presented in this chapter.

17.3 Names of works of art (books, movies, video games, etc.) are quite different from the kinds of named entities we've discussed in this chapter. Collect a list of names of works of art from a particular category from a Web-based source (e.g., gutenberg.org, amazon.com, imdb.com, etc.). Analyze your list and give examples of ways that the names in it are likely to be problematic for the techniques described in this chapter.

17.4 Develop an NER system specific to the category of names that you collected in the last exercise. Evaluate your system on a collection of text likely to contain instances of these named entities.

17.5 Acronym expansion, the process of associating a phrase with an acronym, can be accomplished by a simple form of relational analysis. Develop a system based on the relation analysis approaches described in this chapter to populate a database of acronym expansions. If you focus on English **Three Letter Acronyms** (TLAs) you can evaluate your system's performance by comparing it to Wikipedia's TLA page.

17.6 A useful functionality in newer email and calendar applications is the ability to associate temporal expressions connected with events in email (doctor's appointments, meeting planning, party invitations, etc.) with specific calendar entries. Collect a corpus of email containing temporal expressions related to event planning. How do these expressions compare to the kinds of expressions commonly found in news text that we've been discussing in this chapter?

17.7 Acquire the CMU seminar corpus and develop a template-filling system by using any of the techniques mentioned in Section 17.5. Analyze how well your system performs as compared with state-of-the-art results on this corpus.

CHAPTER

18 Semantic Role Labeling

Sometime between the 7th and 4th centuries BCE, the Indian grammarian Pāṇini[1] wrote a famous treatise on Sanskrit grammar, the Aṣṭādhyāyī ('8 books'), a treatise that has been called "one of the greatest monuments of human intelligence" (Bloomfield, 1933b, 11). The work describes the linguistics of the Sanskrit language in the form of 3959 sutras, each very efficiently (since it had to be memorized!) expressing part of a formal rule system that brilliantly prefigured modern mechanisms of formal language theory (Penn and Kiparsky, 2012). One set of rules, relevant to our discussion in this chapter, describes the kārakas, semantic relationships between a verb and noun arguments, roles like *agent*, *instrument*, or *destination*. Pāṇini's work was the earliest we know of that tried to understand the linguistic realization of events and their participants. This task of understanding participants and their relationship to events—being able to answer the question "Who did what to whom" (and perhaps also "when and where")—is a central question of natural language understanding.

Let's move forward 2.5 millenia to the present and consider the very mundane goal of understanding text about a purchase of stock by XYZ Corporation. This purchasing event could take on a wide variety of surface forms. In the following sentences we see that it could be described by a verb (*sold, bought*) or a noun (*purchase*), and that XYZ Corp can be the syntactic subject (of *bought*), the indirect object (of *sold*), or in a genitive or noun compound relation (with the noun *purchase*) despite having notationally the same role in all of them:

- XYZ corporation bought the stock.
- They sold the stock to XYZ corporation.
- The stock was bought by XYZ corporation.
- The purchase of the stock by XYZ corporation...
- The stock purchase by XYZ corporation...

In this chapter we introduce a level of representation that lets us capture the commonality between these sentences. We will be able to represent the fact that there was a purchase event, that the participants in this event were XYZ Corp and some stock, and that XYZ Corp played a specific role, the role of acquiring the stock.

We call this shallow semantic representation level **semantic roles**. Semantic roles are representations that express the abstract role that arguments of a predicate can take in the event; these can be very specific, like the BUYER, abstract like the AGENT, or super-abstract (the PROTO-AGENT). These roles can both represent general semantic properties of the arguments and also express their likely relationship to the syntactic role of the argument in the sentence. AGENTS tend to be the subject of

[1] Figure shows a birch bark manuscript from Kashmir of the Rupavatra, a grammatical textbook based on the Sanskrit grammar of Panini. Image from the Wellcome Collection.

an active sentence, THEMES the direct object, and so on. These relations are codified in databases like PropBank and FrameNet. We'll introduce **semantic role labeling**, the task of assigning roles to the constituents or phrases in sentences. We'll also discuss **selectional restrictions**, the semantic sortal restrictions or preferences that each individual predicate can express about its potential arguments, such as the fact that the theme of the verb *eat* is generally something edible. Along the way, we'll describe the various ways these representations can help in language understanding tasks like question answering and machine translation.

18.1 Semantic Roles

Consider how in Chapter 14 we represented the meaning of arguments for sentences like these:

(18.1) Sasha broke the window.

(18.2) Pat opened the door.

A neo-Davidsonian event representation of these two sentences would be

$$\begin{aligned}
&\exists e,x,y\ Breaking(e) \wedge Breaker(e,Sasha)\\
&\qquad \wedge BrokenThing(e,y) \wedge Window(y)\\
&\exists e,x,y\ Opening(e) \wedge Opener(e,Pat)\\
&\qquad \wedge OpenedThing(e,y) \wedge Door(y)
\end{aligned}$$

deep roles

In this representation, the roles of the subjects of the verbs *break* and *open* are *Breaker* and *Opener* respectively. These **deep roles** are specific to each event; *Breaking* events have *Breakers*, *Opening* events have *Openers*, and so on.

If we are going to be able to answer questions, perform inferences, or do any further kinds of natural language understanding of these events, we'll need to know a little more about the semantics of these arguments. *Breakers* and *Openers* have something in common. They are both volitional actors, often animate, and they have direct causal responsibility for their events.

thematic roles
agents

Thematic roles are a way to capture this semantic commonality between *Breakers* and *Eaters*. We say that the subjects of both these verbs are **agents**. Thus, AGENT is the thematic role that represents an abstract idea such as volitional causation. Similarly, the direct objects of both these verbs, the *BrokenThing* and *OpenedThing*, are both prototypically inanimate objects that are affected in some way by the action. The semantic role for these participants is **theme**.

theme

Although thematic roles are one of the oldest linguistic models, as we saw above, their modern formulation is due to Fillmore (1968) and Gruber (1965). Although there is no universally agreed-upon set of roles, Figs. 18.1 and 18.2 list some thematic roles that have been used in various computational papers, together with rough definitions and examples. Most thematic role sets have about a dozen roles, but we'll see sets with smaller numbers of roles with even more abstract meanings, and sets with very large numbers of roles that are specific to situations. We'll use the general term **semantic roles** for all sets of roles, whether small or large.

semantic roles

Thematic Role	Definition
AGENT	The volitional causer of an event
EXPERIENCER	The experiencer of an event
FORCE	The non-volitional causer of the event
THEME	The participant most directly affected by an event
RESULT	The end product of an event
CONTENT	The proposition or content of a propositional event
INSTRUMENT	An instrument used in an event
BENEFICIARY	The beneficiary of an event
SOURCE	The origin of the object of a transfer event
GOAL	The destination of an object of a transfer event

Figure 18.1 Some commonly used thematic roles with their definitions.

Thematic Role	Example
AGENT	*The waiter* spilled the soup.
EXPERIENCER	*John* has a headache.
FORCE	*The wind* blows debris from the mall into our yards.
THEME	Only after Benjamin Franklin broke *the ice*...
RESULT	The city built a *regulation-size baseball diamond*...
CONTENT	Mona asked *"You met Mary Ann at a supermarket?"*
INSTRUMENT	He poached catfish, stunning them *with a shocking device*...
BENEFICIARY	Whenever Ann Callahan makes hotel reservations *for her boss*...
SOURCE	I flew in *from Boston*.
GOAL	I drove *to Portland*.

Figure 18.2 Some prototypical examples of various thematic roles.

18.2 Diathesis Alternations

The main reason computational systems use semantic roles is to act as a shallow meaning representation that can let us make simple inferences that aren't possible from the pure surface string of words, or even from the parse tree. To extend the earlier examples, if a document says that *Company A acquired Company B*, we'd like to know that this answers the query *Was Company B acquired?* despite the fact that the two sentences have very different surface syntax. Similarly, this shallow semantics might act as a useful intermediate language in machine translation.

Semantic roles thus help generalize over different surface realizations of predicate arguments. For example, while the AGENT is often realized as the subject of the sentence, in other cases the THEME can be the subject. Consider these possible realizations of the thematic arguments of the verb *break*:

(18.3) *John* *broke the window.*
AGENT THEME

(18.4) *John* *broke the window with a rock.*
AGENT THEME INSTRUMENT

(18.5) *The rock* *broke the window.*
INSTRUMENT THEME

(18.6) *The window broke.*
THEME

(18.7) *The window was broken by John.*
THEME AGENT

These examples suggest that *break* has (at least) the possible arguments AGENT, THEME, and INSTRUMENT. The set of thematic role arguments taken by a verb is often called the **thematic grid**, θ-grid, or **case frame**. We can see that there are (among others) the following possibilities for the realization of these arguments of *break*:

thematic grid
case frame

AGENT/Subject, THEME/Object
AGENT/Subject, THEME/Object, INSTRUMENT/PP_{with}
INSTRUMENT/Subject, THEME/Object
THEME/Subject

It turns out that many verbs allow their thematic roles to be realized in various syntactic positions. For example, verbs like *give* can realize the THEME and GOAL arguments in two different ways:

(18.8) a. *Doris* *gave the book* *to Cary.*
AGENT THEME GOAL

b. *Doris* *gave Cary* *the book.*
AGENT GOAL THEME

These multiple argument structure realizations (the fact that *break* can take AGENT, INSTRUMENT, or THEME as subject, and *give* can realize its THEME and GOAL in either order) are called **verb alternations** or **diathesis alternations**. The alternation we showed above for *give*, the **dative alternation**, seems to occur with particular semantic classes of verbs, including "verbs of future having" (*advance*, *allocate*, *offer*, *owe*), "send verbs" (*forward*, *hand*, *mail*), "verbs of throwing" (*kick*, *pass*, *throw*), and so on. Levin (1993) lists for 3100 English verbs the semantic classes to which they belong (47 high-level classes, divided into 193 more specific classes) and the various alternations in which they participate. These lists of verb classes have been incorporated into the online resource VerbNet (Kipper et al., 2000), which links each verb to both WordNet and FrameNet entries.

verb alternation
dative alternation

18.3 Semantic Roles: Problems with Thematic Roles

Representing meaning at the thematic role level seems like it should be useful in dealing with complications like diathesis alternations. Yet it has proved quite difficult to come up with a standard set of roles, and equally difficult to produce a formal definition of roles like AGENT, THEME, or INSTRUMENT.

For example, researchers attempting to define role sets often find they need to fragment a role like AGENT or THEME into many specific roles. Levin and Rappaport Hovav (2005) summarize a number of such cases, such as the fact there seem to be at least two kinds of INSTRUMENTS, *intermediary* instruments that can appear as subjects and *enabling* instruments that cannot:

(18.9) a. The cook opened the jar with the new gadget.
b. The new gadget opened the jar.

(18.10) a. Shelly ate the sliced banana with a fork.
b. *The fork ate the sliced banana.

In addition to the fragmentation problem, there are cases in which we'd like to reason about and generalize across semantic roles, but the finite discrete lists of roles don't let us do this.

Finally, it has proved difficult to formally define the thematic roles. Consider the AGENT role; most cases of AGENTS are animate, volitional, sentient, causal, but any individual noun phrase might not exhibit all of these properties.

semantic role

These problems have led to alternative **semantic role** models that use either many fewer or many more roles.

proto-agent
proto-patient

The first of these options is to define **generalized semantic roles** that abstract over the specific thematic roles. For example, PROTO-AGENT and PROTO-PATIENT are generalized roles that express roughly agent-like and roughly patient-like meanings. These roles are defined, not by necessary and sufficient conditions, but rather by a set of heuristic features that accompany more agent-like or more patient-like meanings. Thus, the more an argument displays agent-like properties (being volitionally involved in the event, causing an event or a change of state in another participant, being sentient or intentionally involved, moving) the greater the likelihood that the argument can be labeled a PROTO-AGENT. The more patient-like the properties (undergoing change of state, causally affected by another participant, stationary relative to other participants, etc.), the greater the likelihood that the argument can be labeled a PROTO-PATIENT.

The second direction is instead to define semantic roles that are specific to a particular verb or a particular group of semantically related verbs or nouns.

In the next two sections we describe two commonly used lexical resources that make use of these alternative versions of semantic roles. **PropBank** uses both proto-roles and verb-specific semantic roles. **FrameNet** uses semantic roles that are specific to a general semantic idea called a *frame*.

18.4 The Proposition Bank

PropBank

The **Proposition Bank**, generally referred to as **PropBank**, is a resource of sentences annotated with semantic roles. The English PropBank labels all the sentences in the Penn TreeBank; the Chinese PropBank labels sentences in the Penn Chinese TreeBank. Because of the difficulty of defining a universal set of thematic roles, the semantic roles in PropBank are defined with respect to an individual verb sense. Each sense of each verb thus has a specific set of roles, which are given only numbers rather than names: **Arg0**, **Arg1**, **Arg2**, and so on. In general, **Arg0** represents the PROTO-AGENT, and **Arg1**, the PROTO-PATIENT. The semantics of the other roles are less consistent, often being defined specifically for each verb. Nonetheless there are some generalization; the **Arg2** is often the benefactive, instrument, attribute, or end state, the **Arg3** the start point, benefactive, instrument, or attribute, and the **Arg4** the end point.

Here are some slightly simplified PropBank entries for one sense each of the verbs *agree* and *fall*. Such PropBank entries are called **frame files**; note that the definitions in the frame file for each role (“Other entity agreeing”, “Extent, amount fallen”) are informal glosses intended to be read by humans, rather than being formal definitions.

(18.11) **agree.01**

Arg0: Agreer
Arg1: Proposition
Arg2: Other entity agreeing

Ex1: [$_{\text{Arg0}}$ The group] *agreed* [$_{\text{Arg1}}$ it wouldn't make an offer].
Ex2: [$_{\text{ArgM-TMP}}$ Usually] [$_{\text{Arg0}}$ John] *agrees* [$_{\text{Arg2}}$ with Mary] [$_{\text{Arg1}}$ on everything].

(18.12) **fall.01**
Arg1: Logical subject, patient, thing falling
Arg2: Extent, amount fallen
Arg3: start point
Arg4: end point, end state of arg1
Ex1: [$_{\text{Arg1}}$ Sales] *fell* [$_{\text{Arg4}}$ to \$25 million] [$_{\text{Arg3}}$ from \$27 million].
Ex2: [$_{\text{Arg1}}$ The average junk bond] *fell* [$_{\text{Arg2}}$ by 4.2%].

Note that there is no Arg0 role for *fall*, because the normal subject of *fall* is a PROTO-PATIENT.

The PropBank semantic roles can be useful in recovering shallow semantic information about verbal arguments. Consider the verb *increase*:

(18.13) **increase.01** "go up incrementally"
Arg0: causer of increase
Arg1: thing increasing
Arg2: amount increased by, EXT, or MNR
Arg3: start point
Arg4: end point

A PropBank semantic role labeling would allow us to infer the commonality in the event structures of the following three examples, that is, that in each case *Big Fruit Co.* is the AGENT and *the price of bananas* is the THEME, despite the differing surface forms.

(18.14) [$_{\text{Arg0}}$ Big Fruit Co.] increased [$_{\text{Arg1}}$ the price of bananas].
(18.15) [$_{\text{Arg1}}$ The price of bananas] was increased again [$_{\text{Arg0}}$ by Big Fruit Co.]
(18.16) [$_{\text{Arg1}}$ The price of bananas] increased [$_{\text{Arg2}}$ 5%].

PropBank also has a number of non-numbered arguments called **ArgMs**, (ArgM-TMP, ArgM-LOC, etc) which represent modification or adjunct meanings. These are relatively stable across predicates, so aren't listed with each frame file. Data labeled with these modifiers can be helpful in training systems to detect temporal, location, or directional modification across predicates. Some of the ArgM's include:

TMP	when?	yesterday evening, now
LOC	where?	at the museum, in San Francisco
DIR	where to/from?	down, to Bangkok
MNR	how?	clearly, with much enthusiasm
PRP/CAU	why?	because ... , in response to the ruling
REC		themselves, each other
ADV	miscellaneous	
PRD	secondary predication	...ate the meat raw

NomBank While PropBank focuses on verbs, a related project, **NomBank** (Meyers et al., 2004) adds annotations to noun predicates. For example the noun *agreement* in *Apple's agreement with IBM* would be labeled with Apple as the Arg0 and IBM as

the Arg2. This allows semantic role labelers to assign labels to arguments of both verbal and nominal predicates.

18.5 FrameNet

While making inferences about the semantic commonalities across different sentences with *increase* is useful, it would be even more useful if we could make such inferences in many more situations, across different verbs, and also between verbs and nouns. For example, we'd like to extract the similarity among these three sentences:

(18.17) [$_{\text{Arg1}}$ The price of bananas] increased [$_{\text{Arg2}}$ 5%].
(18.18) [$_{\text{Arg1}}$ The price of bananas] rose [$_{\text{Arg2}}$ 5%].
(18.19) There has been a [$_{\text{Arg2}}$ 5%] rise [$_{\text{Arg1}}$ in the price of bananas].

Note that the second example uses the different verb *rise*, and the third example uses the noun rather than the verb *rise*. We'd like a system to recognize that *the price of bananas* is what went up, and that *5%* is the amount it went up, no matter whether the *5%* appears as the object of the verb *increased* or as a nominal modifier of the noun *rise*.

FrameNet The **FrameNet** project is another semantic-role-labeling project that attempts to address just these kinds of problems (Baker et al. 1998, Fillmore et al. 2003, Fillmore and Baker 2009, Ruppenhofer et al. 2016). Whereas roles in the PropBank project are specific to an individual verb, roles in the FrameNet project are specific to a **frame**.

What is a frame? Consider the following set of words:

reservation, flight, travel, buy, price, cost, fare, rates, meal, plane

There are many individual lexical relations of hyponymy, synonymy, and so on between many of the words in this list. The resulting set of relations does not, however, add up to a complete account of how these words are related. They are clearly all defined with respect to a coherent chunk of common-sense background information concerning air travel.

frame We call the holistic background knowledge that unites these words a **frame** (Fillmore, 1985). The idea that groups of words are defined with respect to some background information is widespread in artificial intelligence and cognitive science, where besides **frame** we see related works like a **model** (Johnson-Laird, 1983), or even **script** (Schank and Abelson, 1977).

model **script**

A frame in FrameNet is a background knowledge structure that defines a set of frame-specific semantic roles, called **frame elements**, and includes a set of predicates that use these roles. Each word evokes a frame and profiles some aspect of the frame and its elements. The FrameNet dataset includes a set of frames and frame elements, the lexical units associated with each frame, and a set of labeled example sentences. For example, the **change_position_on_a_scale** frame is defined as follows:

frame elements

> This frame consists of words that indicate the change of an Item's position on a scale (the Attribute) from a starting point (Initial_value) to an end point (Final_value).

Some of the semantic roles (frame elements) in the frame are defined as in Fig. 18.3. Note that these are separated into **core roles**, which are frame specific, and

core roles

non-core roles

non-core roles, which are more like the Arg-M arguments in PropBank, expressed more general properties of time, location, and so on.

Core Roles	
ATTRIBUTE	The ATTRIBUTE is a scalar property that the ITEM possesses.
DIFFERENCE	The distance by which an ITEM changes its position on the scale.
FINAL_STATE	A description that presents the ITEM's state after the change in the ATTRIBUTE's value as an independent predication.
FINAL_VALUE	The position on the scale where the ITEM ends up.
INITIAL_STATE	A description that presents the ITEM's state before the change in the ATTRIBUTE's value as an independent predication.
INITIAL_VALUE	The initial position on the scale from which the ITEM moves away.
ITEM	The entity that has a position on the scale.
VALUE_RANGE	A portion of the scale, typically identified by its end points, along which the values of the ATTRIBUTE fluctuate.
Some Non-Core Roles	
DURATION	The length of time over which the change takes place.
SPEED	The rate of change of the VALUE.
GROUP	The GROUP in which an ITEM changes the value of an ATTRIBUTE in a specified way.

Figure 18.3 The frame elements in the **change_position_on_a_scale** frame from the FrameNet Labelers Guide (Ruppenhofer et al., 2016).

Here are some example sentences:

(18.20) [$_{\text{ITEM}}$ Oil] *rose* [$_{\text{ATTRIBUTE}}$ in price] [$_{\text{DIFFERENCE}}$ by 2%].

(18.21) [$_{\text{ITEM}}$ It] has *increased* [$_{\text{FINAL\_STATE}}$ to having them 1 day a month].

(18.22) [$_{\text{ITEM}}$ Microsoft shares] *fell* [$_{\text{FINAL\_VALUE}}$ to 7 5/8].

(18.23) [$_{\text{ITEM}}$ Colon cancer incidence] *fell* [$_{\text{DIFFERENCE}}$ by 50%] [$_{\text{GROUP}}$ among men].

(18.24) a steady *increase* [$_{\text{INITIAL\_VALUE}}$ from 9.5] [$_{\text{FINAL\_VALUE}}$ to 14.3] [$_{\text{ITEM}}$ in dividends]

(18.25) a [$_{\text{DIFFERENCE}}$ 5%] [$_{\text{ITEM}}$ dividend] *increase*...

Note from these example sentences that the frame includes target words like *rise*, *fall*, and *increase*. In fact, the complete frame consists of the following words:

VERBS:	dwindle	move	soar	escalation	shift
advance	edge	mushroom	swell	explosion	tumble
climb	explode	plummet	swing	fall	
decline	fall	reach	triple	fluctuation	**ADVERBS:**
decrease	fluctuate	rise	tumble	gain	increasingly
diminish	gain	rocket		growth	
dip	grow	shift	**NOUNS:**	hike	
double	increase	skyrocket	decline	increase	
drop	jump	slide	decrease	rise	

FrameNet also codes relationships between frames, allowing frames to inherit from each other, or representing relations between frames like causation (and generalizations among frame elements in different frames can be representing by inheritance as well). Thus, there is a **Cause_change_of_position_on_a_scale** frame that is linked to the **Change_of_position_on_a_scale** frame by the **cause** relation, but that adds an AGENT role and is used for causative examples such as the following:

(18.26) [$_{\text{AGENT}}$ They] *raised* [$_{\text{ITEM}}$ the price of their soda] [$_{\text{DIFFERENCE}}$ by 2%].

Together, these two frames would allow an understanding system to extract the common event semantics of all the verbal and nominal causative and non-causative usages.

FrameNets have also been developed for many other languages including Spanish, German, Japanese, Portuguese, Italian, and Chinese.

18.6 Semantic Role Labeling

semantic role labeling

Semantic role labeling (sometimes shortened as **SRL**) is the task of automatically finding the **semantic roles** of each argument of each predicate in a sentence. Current approaches to semantic role labeling are based on supervised machine learning, often using the FrameNet and PropBank resources to specify what counts as a predicate, define the set of roles used in the task, and provide training and test sets.

Recall that the difference between these two models of semantic roles is that FrameNet (18.27) employs many frame-specific frame elements as roles, while PropBank (18.28) uses a smaller number of numbered argument labels that can be interpreted as verb-specific labels, along with the more general ARGM labels. Some examples:

(18.27)	[You]	can't	[blame]	[the program]	[for being unable to identify it]
	COGNIZER		TARGET	EVALUEE	REASON

(18.28)	[The San Francisco Examiner]	issued	[a special edition]	[yesterday]
	ARG0	TARGET	ARG1	ARGM-TMP

18.6.1 A Feature-based Algorithm for Semantic Role Labeling

A simplified feature-based semantic role labeling algorithm is sketched in Fig. 18.4. Feature-based algorithms—from the very earliest systems like (Simmons, 1973)—begin by parsing, using broad-coverage parsers to assign a parse to the input string. Figure 18.5 shows a parse of (18.28) above. The parse is then traversed to find all words that are predicates.

For each of these predicates, the algorithm examines each node in the parse tree and uses supervised classification to decide the semantic role (if any) it plays for this predicate. Given a labeled training set such as PropBank or FrameNet, a feature vector is extracted for each node, using feature templates described in the next subsection. A 1-of-N classifier is then trained to predict a semantic role for each constituent given these features, where N is the number of potential semantic roles plus an extra NONE role for non-role constituents. Any standard classification algorithms can be used. Finally, for each test sentence to be labeled, the classifier is run on each relevant constituent.

Instead of training a single-stage classifier as in Fig. 18.5, the node-level classification task can be broken down into multiple steps:

1. **Pruning:** Since only a small number of the constituents in a sentence are arguments of any given predicate, many systems use simple heuristics to prune unlikely constituents.
2. **Identification:** a binary classification of each node as an argument to be labeled or a NONE.

```
function SEMANTICROLELABEL(words) returns labeled tree

    parse ← PARSE(words)
    for each predicate in parse do
        for each node in parse do
            featurevector ← EXTRACTFEATURES(node, predicate, parse)
            CLASSIFYNODE(node, featurevector, parse)
```

Figure 18.4 A generic semantic-role-labeling algorithm. CLASSIFYNODE is a 1-of-*N* classifier that assigns a semantic role (or NONE for non-role constituents), trained on labeled data such as FrameNet or PropBank.

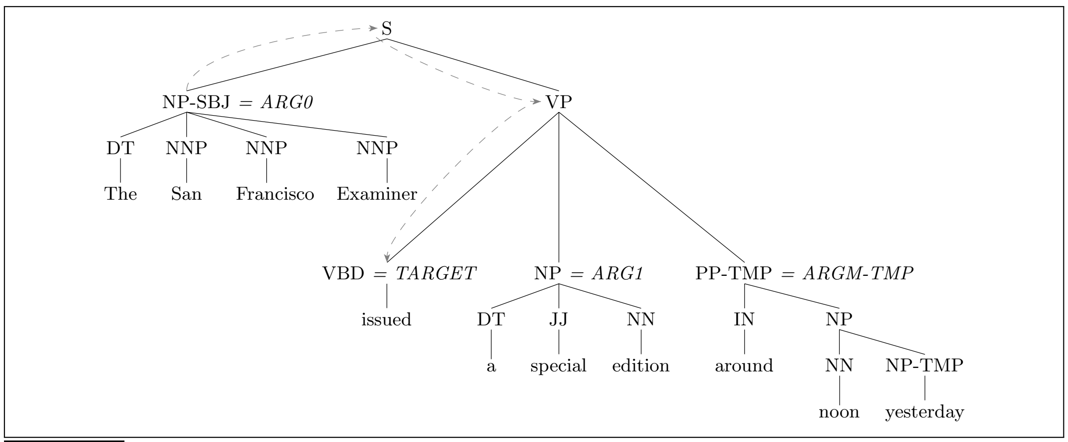


Figure 18.5 Parse tree for a PropBank sentence, showing the PropBank argument labels. The dotted line shows the **path** feature NP↑S↓VP↓VBD for ARG0, the NP-SBJ constituent *The San Francisco Examiner.*

3. **Classification:** a 1-of-*N* classification of all the constituents that were labeled as arguments by the previous stage

The separation of identification and classification may lead to better use of features (different features may be useful for the two tasks) or to computational efficiency.

Global Optimization

The classification algorithm of Fig. 18.5 classifies each argument separately ('locally'), making the simplifying assumption that each argument of a predicate can be labeled independently. This assumption is false; there are interactions between arguments that require a more 'global' assignment of labels to constituents. For example, constituents in FrameNet and PropBank are required to be non-overlapping. More significantly, the semantic roles of constituents are not independent. For example PropBank does not allow multiple identical arguments; two constituents of the same verb cannot both be labeled ARG0 .

Role labeling systems thus often add a fourth step to deal with global consistency across the labels in a sentence. For example, the local classifiers can return a list of possible labels associated with probabilities for each constituent, and a second-pass Viterbi decoding or re-ranking approach can be used to choose the best consensus label. Integer linear programming (ILP) is another common way to choose a solution that conforms best to multiple constraints.

Features for Semantic Role Labeling

Most systems use some generalization of the core set of features introduced by Gildea and Jurafsky (2000). Common basic features templates (demonstrated on the *NP-SBJ* constituent *The San Francisco Examiner* in Fig. 18.5) include:

- The governing **predicate**, in this case the verb *issued*. The predicate is a crucial feature since labels are defined only with respect to a particular predicate.
- The **phrase type** of the constituent, in this case, *NP* (or *NP-SBJ*). Some semantic roles tend to appear as *NP*s, others as *S* or *PP*, and so on.
- The **headword** of the constituent, *Examiner*. The headword of a constituent can be computed with standard head rules, such as those given in Chapter 10 in Fig. 10.12. Certain headwords (e.g., pronouns) place strong constraints on the possible semantic roles they are likely to fill.
- The **headword part of speech** of the constituent, *NNP*.
- The **path** in the parse tree from the constituent to the predicate. This path is marked by the dotted line in Fig. 18.5. Following Gildea and Jurafsky (2000), we can use a simple linear representation of the path, NP↑S↓VP↓VBD. ↑ and ↓ represent upward and downward movement in the tree, respectively. The path is very useful as a compact representation of many kinds of grammatical function relationships between the constituent and the predicate.
- The **voice** of the clause in which the constituent appears, in this case, **active** (as contrasted with **passive**). Passive sentences tend to have strongly different linkings of semantic roles to surface form than do active ones.
- The binary **linear position** of the constituent with respect to the predicate, either **before** or **after**.
- The **subcategorization** of the predicate, the set of expected arguments that appear in the verb phrase. We can extract this information by using the phrase-structure rule that expands the immediate parent of the predicate; VP → VBD NP PP for the predicate in Fig. 18.5.
- The named entity type of the constituent.
- The first words and the last word of the constituent.

The following feature vector thus represents the first NP in our example (recall that most observations will have the value NONE rather than, for example, ARG0, since most constituents in the parse tree will not bear a semantic role):

> ARG0: [issued, NP, Examiner, NNP, NP↑S↓VP↓VBD, active, before, VP → NP PP, ORG, The, Examiner]

Other features are often used in addition, such as sets of n-grams inside the constituent, or more complex versions of the path features (the upward or downward halves, or whether particular nodes occur in the path).

It's also possible to use dependency parses instead of constituency parses as the basis of features, for example using dependency parse paths instead of constituency paths.

18.6.2 A Neural Algorithm for Semantic Role Labeling

The standard neural algorithm for semantic role labeling is based on the bi-LSTM IOB tagger introduced in Chapter 9, which we've seen applied to part-of-speech tagging and named entity tagging, among other tasks. Recall that with IOB tagging,

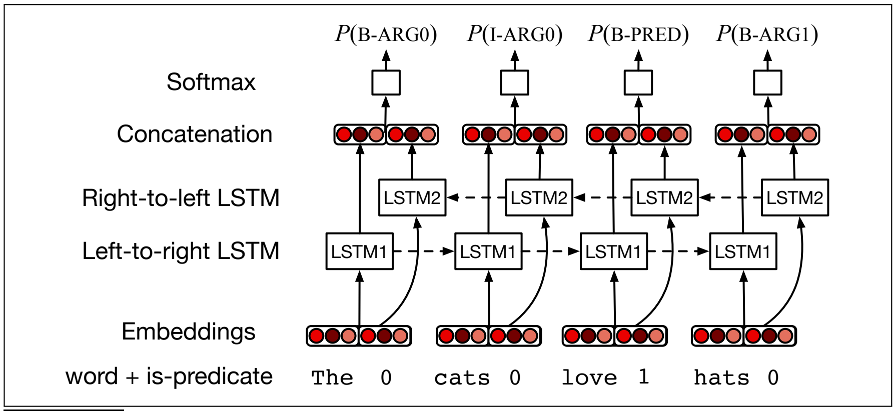


Figure 18.6 A bi-LSTM approach to semantic role labeling. Most actual networks are much deeper than shown in this figure; 3 to 4 bi-LSTM layers (6 to 8 total LSTMs) are common. The input is a concatenation of an embedding for the input word and an embedding of a binary variable which is 1 for the predicate to 0 for all other words. After He et al. (2017).

we have a begin and end tag for each possible role (B-ARG0, I-ARG0; B-ARG1, I-ARG1, and so on), plus an outside tag O.

As with all the taggers, the goal is to compute the highest probability tag sequence $\hat{y}$, given the input sequence of words w:

$$\hat{y} = \underset{y \in T}{\operatorname{argmax}} P(\mathbf{y}|\mathbf{w})$$

In algorithms like He et al. (2017), each input word is mapped to pre-trained embeddings, and also associated with an embedding for a flag (0/1) variable indicating whether that input word is the predicate. These concatenated embeddings are passed through multiple layers of bi-directional LSTM. State-of-the-art algorithms tend to be deeper than for POS or NER tagging, using 3 to 4 layers (6 to 8 total LSTMs). Highway layers can be used to connect these layers as well.

Output from the last bi-LSTM can then be turned into an IOB sequence as for POS or NER tagging. Tags can be locally optimized by taking the bi-LSTM output, passing it through a single layer into a softmax for each word that creates a probability distribution over all SRL tags and the most likely tag for word x_i is chosen as t_i, computing for each word essentially:

$$\hat{y}_i = \underset{t \in tags}{\operatorname{argmax}} P(t|w_i)$$

However, just as feature-based SRL tagging, this local approach to decoding doesn't exploit the global constraints between tags; a tag I-ARG0, for example, must follow another I-ARG0 or B-ARG0.

As we saw for POS and NER tagging, there are many ways to take advantage of these global constraints. A CRF layer can be used instead of a softmax layer on top of the bi-LSTM output, and the Viterbi decoding algorithm can be used to decode from the CRF.

An even simpler Viterbi decoding algorithm that may perform equally well and doesn't require adding CRF complexity to the training process is to start with the simple softmax. The softmax output (the entire probability distribution over tags) for each word is then treated it as a lattice and we can do Viterbi decoding through the lattice. The hard IOB constraints can act as the transition probabilities in the

Viterbi decoding (Thus the transition from state I-ARG0 to I-ARG1 would have probability 0). Alternatively, the training data can be used to learn bigram or trigram tag transition probabilities as if doing HMM decoding. Fig. 18.6 shows a sketch of the algorithm.

18.6.3 Evaluation of Semantic Role Labeling

The standard evaluation for semantic role labeling is to require that each argument label must be assigned to the exactly correct word sequence or parse constituent, and then compute precision, recall, and *F*-measure. Identification and classification can also be evaluated separately. Two common datasets used for evaluation are CoNLL-2005 (Carreras and Màrquez, 2005) and CoNLL-2012 (Pradhan et al., 2013).

18.7 Selectional Restrictions

selectional restriction

We turn in this section to another way to represent facts about the relationship between predicates and arguments. A **selectional restriction** is a semantic type constraint that a verb imposes on the kind of concepts that are allowed to fill its argument roles. Consider the two meanings associated with the following example:

(18.29) I want to eat someplace nearby.

There are two possible parses and semantic interpretations for this sentence. In the sensible interpretation, *eat* is intransitive and the phrase *someplace nearby* is an adjunct that gives the location of the eating event. In the nonsensical *speaker-as-Godzilla* interpretation, *eat* is transitive and the phrase *someplace nearby* is the direct object and the THEME of the eating, like the NP *Malaysian food* in the following sentences:

(18.30) I want to eat Malaysian food.

How do we know that *someplace nearby* isn't the direct object in this sentence? One useful cue is the semantic fact that the THEME of EATING events tends to be something that is *edible*. This restriction placed by the verb *eat* on the filler of its THEME argument is a selectional restriction.

Selectional restrictions are associated with senses, not entire lexemes. We can see this in the following examples of the lexeme *serve*:

(18.31) The restaurant serves green-lipped mussels.

(18.32) Which airlines serve Denver?

Example (18.31) illustrates the offering-food sense of *serve*, which ordinarily restricts its THEME to be some kind of food Example (18.32) illustrates the *provides a commercial service to* sense of *serve*, which constrains its THEME to be some type of appropriate location.

Selectional restrictions vary widely in their specificity. The verb *imagine*, for example, imposes strict requirements on its AGENT role (restricting it to humans and other animate entities) but places very few semantic requirements on its THEME role. A verb like *diagonalize*, on the other hand, places a very specific constraint on the filler of its THEME role: it has to be a matrix, while the arguments of the adjectives *odorless* are restricted to concepts that could possess an odor:

(18.33) In rehearsal, I often ask the musicians to *imagine* a tennis game.

(18.34) Radon is an *odorless* gas that can't be detected by human senses.

(18.35) To *diagonalize* a matrix is to find its eigenvalues.

These examples illustrate that the set of concepts we need to represent selectional restrictions (being a matrix, being able to possess an odor, etc) is quite open ended. This distinguishes selectional restrictions from other features for representing lexical knowledge, like parts-of-speech, which are quite limited in number.

18.7.1 Representing Selectional Restrictions

One way to capture the semantics of selectional restrictions is to use and extend the event representation of Chapter 14. Recall that the neo-Davidsonian representation of an event consists of a single variable that stands for the event, a predicate denoting the kind of event, and variables and relations for the event roles. Ignoring the issue of the λ-structures and using thematic roles rather than deep event roles, the semantic contribution of a verb like *eat* might look like the following:

$$\exists e,x,y\; Eating(e) \wedge Agent(e,x) \wedge Theme(e,y)$$

With this representation, all we know about y, the filler of the THEME role, is that it is associated with an *Eating* event through the *Theme* relation. To stipulate the selectional restriction that y must be something edible, we simply add a new term to that effect:

$$\exists e,x,y\; Eating(e) \wedge Agent(e,x) \wedge Theme(e,y) \wedge EdibleThing(y)$$

When a phrase like *ate a hamburger* is encountered, a semantic analyzer can form the following kind of representation:

$$\exists e,x,y\; Eating(e) \wedge Eater(e,x) \wedge Theme(e,y) \wedge EdibleThing(y) \wedge Hamburger(y)$$

This representation is perfectly reasonable since the membership of y in the category *Hamburger* is consistent with its membership in the category *EdibleThing*, assuming a reasonable set of facts in the knowledge base. Correspondingly, the representation for a phrase such as *ate a takeoff* would be ill-formed because membership in an event-like category such as *Takeoff* would be inconsistent with membership in the category *EdibleThing*.

While this approach adequately captures the semantics of selectional restrictions, there are two problems with its direct use. First, using FOL to perform the simple task of enforcing selectional restrictions is overkill. Other, far simpler, formalisms can do the job with far less computational cost. The second problem is that this approach presupposes a large, logical knowledge base of facts about the concepts that make up selectional restrictions. Unfortunately, although such common-sense knowledge bases are being developed, none currently have the kind of coverage necessary to the task.

A more practical approach is to state selectional restrictions in terms of WordNet synsets rather than as logical concepts. Each predicate simply specifies a WordNet synset as the selectional restriction on each of its arguments. A meaning representation is well-formed if the role filler word is a hyponym (subordinate) of this synset.

For our *ate a hamburger* example, for instance, we could set the selectional restriction on the THEME role of the verb *eat* to the synset {**food, nutrient**}, glossed as *any substance that can be metabolized by an animal to give energy and build*

```
Sense 1
hamburger, beefburger --
(a fried cake of minced beef served on a bun)
=> sandwich
   => snack food
      => dish
         => nutriment, nourishment, nutrition...
            => food, nutrient
               => substance
                  => matter
                     => physical entity
                        => entity
```

Figure 18.7 Evidence from WordNet that hamburgers are edible.

tissue. Luckily, the chain of hypernyms for *hamburger* shown in Fig. 18.7 reveals that hamburgers are indeed food. Again, the filler of a role need not match the restriction synset exactly; it just needs to have the synset as one of its superordinates.

We can apply this approach to the THEME roles of the verbs *imagine*, *lift*, and *diagonalize*, discussed earlier. Let us restrict *imagine*'s THEME to the synset {entity}, *lift*'s THEME to {physical entity}, and *diagonalize* to {matrix}. This arrangement correctly permits *imagine a hamburger* and *lift a hamburger*, while also correctly ruling out *diagonalize a hamburger*.

18.7.2 Selectional Preferences

In the earliest implementations, selectional restrictions were considered strict constraints on the kind of arguments a predicate could take (Katz and Fodor 1963, Hirst 1987). For example, the verb *eat* might require that its THEME argument be `[+FOOD]`. Early word sense disambiguation systems used this idea to rule out senses that violated the selectional restrictions of their governing predicates.

Very quickly, however, it became clear that these selectional restrictions were better represented as preferences rather than strict constraints (Wilks 1975c, Wilks 1975b). For example, selectional restriction violations (like inedible arguments of eat) often occur in well-formed sentences, for example because they are negated (18.36), or because selectional restrictions are overstated (18.37):

(18.36) But it fell apart in 1931, perhaps because people realized you can't **eat** gold for lunch if you're hungry.

(18.37) In his two championship trials, Mr. Kulkarni **ate** glass on an empty stomach, accompanied only by water and tea.

Modern systems for selectional preferences therefore specify the relation between a predicate and its possible arguments with soft constraints of some kind.

Selectional Association

selectional preference strength

One of the most influential has been the **selectional association** model of Resnik (1993). Resnik defines the idea of **selectional preference strength** as the general amount of information that a predicate tells us about the semantic class of its arguments. For example, the verb *eat* tells us a lot about the semantic class of its direct objects, since they tend to be edible. The verb *be*, by contrast, tells us less about its direct objects. The selectional preference strength can be defined by the difference in information between two distributions: the distribution of expected semantic

classes $P(c)$ (how likely is it that a direct object will fall into class c) and the distribution of expected semantic classes for the particular verb $P(c|v)$ (how likely is it that the direct object of the specific verb v will fall into semantic class c). The greater the difference between these distributions, the more information the verb is giving us about possible objects. The difference between these two distributions can be quantified by **relative entropy**, or the Kullback-Leibler divergence (Kullback and Leibler, 1951). The Kullback-Leibler or **KL divergence** $D(P||Q)$ expresses the difference between two probability distributions P and Q (we'll return to this when we discuss distributional models of meaning in Chapter 6).

relative entropy

KL divergence

$$D(P||Q) = \sum_x P(x)\log\frac{P(x)}{Q(x)} \tag{18.38}$$

The selectional preference $S_R(v)$ uses the KL divergence to express how much information, in bits, the verb v expresses about the possible semantic class of its argument.

$$\begin{aligned} S_R(v) &= D(P(c|v)||P(c)) \\ &= \sum_c P(c|v)\log\frac{P(c|v)}{P(c)} \end{aligned} \tag{18.39}$$

selectional association

Resnik then defines the **selectional association** of a particular class and verb as the relative contribution of that class to the general selectional preference of the verb:

$$A_R(v,c) = \frac{1}{S_R(v)}P(c|v)\log\frac{P(c|v)}{P(c)} \tag{18.40}$$

The selectional association is thus a probabilistic measure of the strength of association between a predicate and a class dominating the argument to the predicate. Resnik estimates the probabilities for these associations by parsing a corpus, counting all the times each predicate occurs with each argument word, and assuming that each word is a partial observation of all the WordNet concepts containing the word. The following table from Resnik (1996) shows some sample high and low selectional associations for verbs and some WordNet semantic classes of their direct objects.

Verb	Direct Object Semantic Class	Assoc	Direct Object Semantic Class	Assoc
read	WRITING	6.80	ACTIVITY	-.20
write	WRITING	7.26	COMMERCE	0
see	ENTITY	5.79	METHOD	-0.01

Selectional Preference via Conditional Probability

An alternative to using selectional association between a verb and the WordNet class of its arguments, is to simply use the conditional probability of an argument word given a predicate verb. This simple model of selectional preferences can be used to directly model the strength of association of one verb (predicate) with one noun (argument).

The conditional probability model can be computed by parsing a very large corpus (billions of words), and computing co-occurrence counts: how often a given verb occurs with a given noun in a given relation. The conditional probability of an

argument noun given a verb for a particular relation $P(n|v,r)$ can then be used as a selectional preference metric for that pair of words (Brockmann and Lapata, 2003):

$$P(n|v,r) = \begin{cases} \frac{C(n,v,r)}{C(v,r)} & \text{if } C(n,v,r) > 0 \\ 0 & \text{otherwise} \end{cases}$$

The inverse probability $P(v|n,r)$ was found to have better performance in some cases (Brockmann and Lapata, 2003):

$$P(v|n,r) = \begin{cases} \frac{C(n,v,r)}{C(n,r)} & \text{if } C(n,v,r) > 0 \\ 0 & \text{otherwise} \end{cases}$$

In cases where it's not possible to get large amounts of parsed data, another option, at least for direct objects, is to get the counts from simple part-of-speech based approximations. For example pairs can be extracted using the pattern "V Det N", where V is any form of the verb, Det is *the—a—*ϵ and N is the singular or plural form of the noun (Keller and Lapata, 2003).

An even simpler approach is to use the simple log co-occurrence frequency of the predicate with the argument $\log count(v,n,r)$ instead of conditional probability; this seems to do better for extracting preferences for syntactic subjects rather than objects (Brockmann and Lapata, 2003).

Evaluating Selectional Preferences

pseudowords

One way to evaluate models of selectional preferences is to use **pseudowords** (Gale et al. 1992c, Schütze 1992a). A pseudoword is an artificial word created by concatenating a test word in some context (say *banana*) with a confounder word (say *door*) to create *banana-door*). The task of the system is to identify which of the two words is the original word. To evaluate a selectional preference model (for example on the relationship between a verb and a direct object) we take a test corpus and select all verb tokens. For each verb token (say *drive*) we select the direct object (e.g., *car*), concatenated with a confounder word that is its *nearest neighbor*, the noun with the frequency closest to the original (say *house*), to make *car/house*). We then use the selectional preference model to choose which of *car* and *house* are more preferred objects of *drive*, and compute how often the model chooses the correct original object (e.g., (*car*) (Chambers and Jurafsky, 2010).

Another evaluation metric is to get human preferences for a test set of verb-argument pairs, and have them rate their degree of plausibility. This is usually done by using magnitude estimation, a technique from psychophysics, in which subjects rate the plausibility of an argument proportional to a modulus item. A selectional preference model can then be evaluated by its correlation with the human preferences (Keller and Lapata, 2003).

18.8 Primitive Decomposition of Predicates

componential analysis

One way of thinking about the semantic roles we have discussed through the chapter is that they help us define the roles that arguments play in a decompositional way, based on finite lists of thematic roles (agent, patient, instrument, proto-agent, proto-patient, etc.) This idea of decomposing meaning into sets of primitive semantics elements or features, called **primitive decomposition** or **componential analysis**,

has been taken even further, and focused particularly on predicates.

Consider these examples of the verb *kill*:

(18.41) Jim killed his philodendron.

(18.42) Jim did something to cause his philodendron to become not alive.

There is a truth-conditional ('propositional semantics') perspective from which these two sentences have the same meaning. Assuming this equivalence, we could represent the meaning of *kill* as:

(18.43) KILL(x,y) ⇔ CAUSE(x, BECOME(NOT(ALIVE(y))))

thus using semantic primitives like *do*, *cause*, *become not*, and *alive*.

Indeed, one such set of potential semantic primitives has been used to account for some of the verbal alternations discussed in Section 18.2 (Lakoff 1965, Dowty 1979). Consider the following examples.

(18.44) John opened the door. ⇒ CAUSE(John, BECOME(OPEN(door)))

(18.45) The door opened. ⇒ BECOME(OPEN(door))

(18.46) The door is open. ⇒ OPEN(door)

The decompositional approach asserts that a single state-like predicate associated with *open* underlies all of these examples. The differences among the meanings of these examples arises from the combination of this single predicate with the primitives CAUSE and BECOME.

While this approach to primitive decomposition can explain the similarity between states and actions or causative and non-causative predicates, it still relies on having a large number of predicates like *open*. More radical approaches choose to break down these predicates as well. One such approach to verbal predicate decomposition that played a role in early natural language understanding systems is **conceptual dependency** (CD), a set of ten primitive predicates, shown in Fig. 18.8.

conceptual dependency

Primitive	Definition
ATRANS	The abstract transfer of possession or control from one entity to another
PTRANS	The physical transfer of an object from one location to another
MTRANS	The transfer of mental concepts between entities or within an entity
MBUILD	The creation of new information within an entity
PROPEL	The application of physical force to move an object
MOVE	The integral movement of a body part by an animal
INGEST	The taking in of a substance by an animal
EXPEL	The expulsion of something from an animal
SPEAK	The action of producing a sound
ATTEND	The action of focusing a sense organ

Figure 18.8 A set of conceptual dependency primitives.

Below is an example sentence along with its CD representation. The verb *brought* is translated into the two primitives ATRANS and PTRANS to indicate that the waiter both physically conveyed the check to Mary and passed control of it to her. Note that CD also associates a fixed set of thematic roles with each primitive to represent the various participants in the action.

(18.47) The waiter brought Mary the check.

$$\exists x,y\, Atrans(x) \wedge Actor(x,Waiter) \wedge Object(x,Check) \wedge To(x,Mary)$$
$$\wedge Ptrans(y) \wedge Actor(y,Waiter) \wedge Object(y,Check) \wedge To(y,Mary)$$

18.9 Summary

- **Semantic roles** are abstract models of the role an argument plays in the event described by the predicate.
- **Thematic roles** are a model of semantic roles based on a single finite list of roles. Other semantic role models include per-verb semantic role lists and **proto-agent/proto-patient**, both of which are implemented in **PropBank**, and per-frame role lists, implemented in **FrameNet**.
- **Semantic role labeling** is the task of assigning semantic role labels to the constituents of a sentence. The task is generally treated as a supervised machine learning task, with models trained on PropBank or FrameNet. Algorithms generally start by parsing a sentence and then automatically tag each parse tree node with a semantic role.
- Semantic **selectional restrictions** allow words (particularly predicates) to post constraints on the semantic properties of their argument words. **Selectional preference** models (like **selectional association** or simple conditional probability) allow a weight or probability to be assigned to the association between a predicate and an argument word or class.

Bibliographical and Historical Notes

Although the idea of semantic roles dates back to Pāṇini, they were re-introduced into modern linguistics by Gruber (1965), Fillmore (1966) and Fillmore (1968)). Fillmore, interestingly, had become interested in argument structure by studying Lucien Tesnière's groundbreaking *Éléments de Syntaxe Structurale* (Tesnière, 1959) in which the term 'dependency' was introduced and the foundations were laid for dependency grammar. Following Tesnière's terminology, Fillmore first referred to argument roles as *actants* (Fillmore, 1966) but quickly switched to the term *case*, (see Fillmore (2003)) and proposed a universal list of semantic roles or cases (Agent, Patient, Instrument, etc.), that could be taken on by the arguments of predicates. Verbs would be listed in the lexicon with their **case frame**, the list of obligatory (or optional) case arguments.

The idea that semantic roles could provide an intermediate level of semantic representation that could help map from syntactic parse structures to deeper, more fully-specified representations of meaning was quickly adopted in natural language processing, and systems for extracting case frames were created for machine translation (Wilks, 1973), question-answering (Hendrix et al., 1973), spoken-language understanding (Nash-Webber, 1975), and dialogue systems (Bobrow et al., 1977). General-purpose semantic role labelers were developed. The earliest ones (Simmons, 1973) first parsed a sentence by means of an ATN (Augmented Transition

Network) parser. Each verb then had a set of rules specifying how the parse should be mapped to semantic roles. These rules mainly made reference to grammatical functions (subject, object, complement of specific prepositions) but also checked constituent internal features such as the animacy of head nouns. Later systems assigned roles from pre-built parse trees, again by using dictionaries with verb-specific case frames (Levin 1977, Marcus 1980).

By 1977 case representation was widely used and taught in AI and NLP courses, and was described as a standard of natural language understanding in the first edition of Winston's (1977) textbook *Artificial Intelligence*.

In the 1980s Fillmore proposed his model of *frame semantics*, later describing the intuition as follows:

> "The idea behind frame semantics is that speakers are aware of possibly quite complex situation types, packages of connected expectations, that go by various names—frames, schemas, scenarios, scripts, cultural narratives, memes—and the words in our language are understood with such frames as their presupposed background." (Fillmore, 2012, p. 712)

The word *frame* seemed to be in the air for a suite of related notions proposed at about the same time by Minsky (1974), Hymes (1974), and Goffman (1974), as well as related notions with other names like *scripts* (Schank and Abelson, 1975) and *schemata* (Bobrow and Norman, 1975) (see Tannen (1979) for a comparison). Fillmore was also influenced by the semantic field theorists and by a visit to the Yale AI lab where he took notice of the lists of slots and fillers used by early information extraction systems like DeJong (1982) and Schank and Abelson (1977). In the 1990s Fillmore drew on these insights to begin the FrameNet corpus annotation project.

At the same time, Beth Levin drew on her early case frame dictionaries (Levin, 1977) to develop her book which summarized sets of verb classes defined by shared argument realizations (Levin, 1993). The VerbNet project built on this work (Kipper et al., 2000), leading soon afterwards to the PropBank semantic-role-labeled corpus created by Martha Palmer and colleagues (Palmer et al., 2005).

The combination of rich linguistic annotation and corpus-based approach instantiated in FrameNet and PropBank led to a revival of automatic approaches to semantic role labeling, first on FrameNet (Gildea and Jurafsky, 2000) and then on PropBank data (Gildea and Palmer, 2002, inter alia). The problem first addressed in the 1970s by hand-written rules was thus now generally recast as one of supervised machine learning enabled by large and consistent databases. Many popular features used for role labeling are defined in Gildea and Jurafsky (2002), Surdeanu et al. (2003), Xue and Palmer (2004), Pradhan et al. (2005), Che et al. (2009), and Zhao et al. (2009). The use of dependency rather than constituency parses was introduced in the CoNLL-2008 shared task (Surdeanu et al., 2008b). For surveys see Palmer et al. (2010) and Màrquez et al. (2008).

The use of neural approachess to semantic role labeling was pioneered by Collobert et al. (2011), who applied a CRF on top of a convolutional net. Early work like Foland, Jr. and Martin (2015) focused on using dependency features. Later work eschewed syntactic features altogether; (Zhou and Xu, 2015) introduced the use of a stacked (6-8 layer) bi-LSTM architecture, and (He et al., 2017) showed how to augment the bi-LSTM architecture with highway networks and also replace the CRF with A* decoding that make it possible to apply a wide variety of global constraints in SRL decoding.

Most semantic role labeling schemes only work within a single sentence, focusing on the object of the verbal (or nominal, in the case of NomBank) predicate.

implicit argument

However, in many cases, a verbal or nominal predicate may have an **implicit argument**: one that appears only in a contextual sentence, or perhaps not at all and must be inferred. In the two sentences *This house has a new owner. The sale was finalized 10 days ago.* the *sale* in the second sentence has no ARG1, but a reasonable reader would infer that the Arg1 should be the *house* mentioned in the prior sentence. Finding these arguments, **implicit argument detection** (sometimes shortened as **iSRL**) was introduced by Gerber and Chai (2010) and Ruppenhofer et al. (2010). See Do et al. (2017) for more recent neural models.

iSRL

To avoid the need for huge labeled training sets, unsupervised approaches for semantic role labeling attempt to induce the set of semantic roles by clustering over arguments. The task was pioneered by Riloff and Schmelzenbach (1998) and Swier and Stevenson (2004); see Grenager and Manning (2006), Titov and Klementiev (2012), Lang and Lapata (2014), Woodsend and Lapata (2015), and Titov and Khoddam (2014).

Selectional preference has been widely studied beyond the selectional association models of Resnik (1993) and Resnik (1996). Methods have included clustering (Rooth et al., 1999), discriminative learning (Bergsma et al., 2008), and topic models (Séaghdha 2010, Ritter et al. 2010), and constraints can be expressed at the level of words or classes (Agirre and Martinez, 2001). Selectional preferences have also been successfully integrated into semantic role labeling (Erk 2007, Zapirain et al. 2013, Do et al. 2017).

Exercises

CHAPTER 19

Computing with Word Senses

"When I use a word", Humpty Dumpty said in rather a scornful tone, "it means just what I choose it to mean – neither more nor less."
Lewis Carroll, Alice in Wonderland

The previous two chapters focused on meaning representations for entire sentences. In those discussions, we made a simplifying assumption by representing *word meanings* as unanalyzed symbols like EAT or JOHN or RED. But representing the meaning of a word by capitalizing it is a pretty unsatisfactory model. In this chapter we introduce a richer model of the semantics of words, drawing on the linguistic study of word meaning, a field called **lexical semantics**, as well as the computational study of these meanings, known as **computational lexical semantics**.

lexical semantics

In representing word meaning, we'll begin with the **lemma** or **citation form** which we said in Chapter 3 is the grammatical form of a word that is used to represent a word in dictionaries and thesaurus. Thus *carpet* is the lemma for *carpets*, and *sing* the lemma for *sing*, *sang*, *sung*. In many languages the infinitive form is used as the lemma for the verb, so Spanish *dormir* "to sleep" is the lemma for *duermes* "you sleep". The specific forms *sung* or *carpets* or *sing* or *duermes* are called **wordforms**.

lemma

citation form

wordform

But a lemma can still have many different meanings. The lemma *bank* can refer to a financial institution or to the sloping side of a river. We call each of these aspects of the meaning of *bank* a **word sense**. The fact that lemmas can be **homonymous** (have multiple senses) causes all sorts of problems in text processing. **Word sense disambiguation** is the task of determining which sense of a word is being used in a particular context, a task with a long history in computational linguistics and applications from machine translation to question answering. We give a number of algorithms for using features from the context for deciding which sense was intended in a particular context.

word sense disambiguation

We'll also introduce **WordNet**, a widely-used thesaurus for representing word senses themselves and for representing relations between senses, like the **IS-A** relation between *dog* and *mammal* or the part-whole relationship between *car* and *engine*. Finally, we'll introduce the task of computing **word similarity** and show how a sense-based thesaurus like WordNet can be used to decide whether two words have a similar meaning.

19.1 Word Senses

Consider the two uses of the lemma *bank* mentioned above, meaning something like "financial institution" and "sloping mound", respectively:

(19.1) Instead, a *bank* can hold the investments in a custodial account in the client's name.

(19.2) But as agriculture burgeons on the east *bank*, the river will shrink even more.

word sense

We represent this variation in usage by saying that the lemma *bank* has two **senses**.[1] A **sense** (or **word sense**) is a discrete representation of one aspect of the meaning of a word. Loosely following lexicographic tradition, we represent each sense by placing a superscript on the orthographic form of the lemma as in **bank**^{1} and **bank**^{2}.

Homonym

Homonymy

homographs

homophones

The senses of a word might not have any particular relation between them; it may be almost coincidental that they share an orthographic form. For example, the *financial institution* and *sloping mound* senses of bank seem relatively unrelated. In such cases we say that the two senses are **homonyms**, and the relation between the senses is one of **homonymy**. Thus **bank**^{1} ("financial institution") and **bank**^{2} ("sloping mound") are homonyms, as are the sense of *bat* meaning 'club for hitting a ball' and the one meaning 'nocturnal flying animal'. We say that these two uses of *bank* are **homographs**, as are the two uses of *bat*, because they are written the same. Two words can be homonyms in a different way if they are spelled differently but pronounced the same, like *write* and *right*, or *piece* and *peace*. We call these **homophones**; they are one cause of real-word spelling errors.

Homonymy causes problems in other areas of language processing as well. In question answering or information retrieval, we can do a much better job helping a user who typed "bat care" if we know whether they are vampires or just want to play baseball. And they will also have different translations; in Spanish the animal bat is a *murciélago* while the baseball bat is a *bate*. **Homographs** that are pronounced differently cause problems for speech synthesis (Chapter 28) such as these homographs of the word *bass*, the fish pronounced *b ae s* and the instrument pronounced *b ey s*.

(19.3) The expert angler from Dora, Mo., was fly-casting for **bass** rather than the traditional trout.

(19.4) The curtain rises to the sound of angry dogs baying and ominous **bass** chords sounding.

Sometimes there is also some semantic connection between the senses of a word. Consider the following example:

(19.5) While some banks furnish blood only to hospitals, others are less restrictive.

Although this is clearly not a use of the "sloping mound" meaning of *bank*, it just as clearly is not a reference to a charitable giveaway by a financial institution. Rather, *bank* has a whole range of uses related to repositories for various biological entities, as in *blood bank*, *egg bank*, and *sperm bank*. So we could call this "biological repository" sense **bank**^{3}. Now this new sense **bank**^{3} has some sort of relation to **bank**^{1}; both **bank**^{1} and **bank**^{3} are repositories for entities that can be deposited and taken out; in **bank**^{1} the entity is monetary, whereas in **bank**^{3} the entity is biological.

polysemy

When two senses are related semantically, we call the relationship between them **polysemy** rather than homonymy. In many cases of polysemy, the semantic relation between the senses is systematic and structured. For example, consider yet another sense of *bank*, exemplified in the following sentence:

(19.6) The bank is on the corner of Nassau and Witherspoon.

This sense, which we can call **bank**^{4}, means something like "the building belonging to a financial institution". It turns out that these two kinds of senses (an organization and the building associated with an organization) occur together for

[1] Confusingly, the word "lemma" is itself ambiguous; it is also sometimes used to mean these separate senses, rather than the citation form of the word. You should be prepared to see both uses in the literature.

many other words as well (*school*, *university*, *hospital*, etc.). Thus, there is a systematic relationship between senses that we might represent as

BUILDING ↔ ORGANIZATION

metonymy

This particular subtype of polysemy relation is often called **metonymy**. Metonymy is the use of one aspect of a concept or entity to refer to other aspects of the entity or to the entity itself. Thus, we are performing metonymy when we use the phrase *the White House* to refer to the administration whose office is in the White House. Other common examples of metonymy include the relation between the following pairings of senses:

Author (*Jane Austen wrote Emma*) ↔ Works of Author (*I really love Jane Austen*)
Tree (*Plums have beautiful blossoms*) ↔ Fruit (*I ate a preserved plum yesterday*)

While it can be useful to distinguish polysemy from unrelated homonymy, there is no hard threshold for how related two senses must be to be considered polysemous. Thus, the difference is really one of degree. This fact can make it very difficult to decide how many senses a word has, that is, whether to make separate senses for closely related usages. There are various criteria for deciding that the differing uses of a word should be represented with discrete senses. We might consider two senses discrete if they have independent truth conditions, different syntactic behavior, and independent sense relations, or if they exhibit antagonistic meanings.

Consider the following uses of the verb *serve* from the WSJ corpus:

(19.7) They rarely *serve* red meat, preferring to prepare seafood.

(19.8) He *served* as U.S. ambassador to Norway in 1976 and 1977.

(19.9) He might have *served* his time, come out and led an upstanding life.

The *serve* of *serving red meat* and that of *serving time* clearly have different truth conditions and presuppositions; the *serve* of *serve as ambassador* has the distinct subcategorization structure *serve as NP*. These heuristics suggest that these are probably three distinct senses of *serve*. One practical technique for determining if two senses are distinct is to conjoin two uses of a word in a single sentence; this kind of conjunction of antagonistic readings is called **zeugma**. Consider the following ATIS examples:

Zeugma

(19.10) Which of those flights serve breakfast?

(19.11) Does Midwest Express serve Philadelphia?

(19.12) ?Does Midwest Express serve breakfast and Philadelphia?

We use (?) to mark those examples that are semantically ill-formed. The oddness of the invented third example (a case of zeugma) indicates there is no sensible way to make a single sense of *serve* work for both breakfast and Philadelphia. We can use this as evidence that *serve* has two different senses in this case.

Dictionaries tend to use many fine-grained senses so as to capture subtle meaning differences, a reasonable approach given that the traditional role of dictionaries is aiding word learners. For computational purposes, we often don't need these fine distinctions, so we may want to group or cluster the senses; we have already done this for some of the examples in this chapter.

How can we define the meaning of a word sense? Can we just look in a dictionary? Consider the following fragments from the definitions of *right*, *left*, *red*, and *blood* from the *American Heritage Dictionary* (Morris, 1985).

right	*adj.* located nearer the right hand esp. being on the right when facing the same direction as the observer.
left	*adj.* located nearer to this side of the body than the right.
red	*n.* the color of blood or a ruby.
blood	*n.* the red liquid that circulates in the heart, arteries and veins of animals.

Note the circularity in these definitions. The definition of *right* makes two direct references to itself, and the entry for *left* contains an implicit self-reference in the phrase *this side of the body*, which presumably means the *left* side. The entries for *red* and *blood* avoid this kind of direct self-reference by instead referencing each other in their definitions. Such circularity is, of course, inherent in all dictionary definitions; these examples are just extreme cases. For humans, such entries are still useful since the user of the dictionary has sufficient grasp of these other terms.

For computational purposes, one approach to defining a sense is to make use of a similar approach to these dictionary definitions; defining a sense through its relationship with other senses. For example, the above definitions make it clear that *right* and *left* are similar kinds of lemmas that stand in some kind of alternation, or opposition, to one another. Similarly, we can glean that *red* is a color, that it can be applied to both *blood* and *rubies*, and that *blood* is a *liquid*. **Sense relations** of this sort are embodied in on-line databases like **WordNet**. Given a sufficiently large database of such relations, many applications are quite capable of performing sophisticated semantic tasks (even if they do not *really* know their right from their left).

19.2 Relations Between Senses

This section explores some of the relations that hold among word senses, focusing on a few that have received significant computational investigation: **synonymy**, **antonymy**, and **hypernymy**, as well as a brief mention of other relations like **meronymy**.

19.2.1 Synonymy and Antonymy

synonym

When two senses of two different words (lemmas) are identical, or nearly identical, we say the two senses are **synonyms**. Synonyms include such pairs as

couch/sofa vomit/throw up filbert/hazelnut car/automobile

A more formal definition of synonymy (between words rather than senses) is that two words are synonymous if they are substitutable one for the other in any sentence without changing the truth conditions of the sentence. We often say in this case that the two words have the same **propositional meaning**.

propositional meaning

While substitutions between some pairs of words like *car/automobile* or *water*/H_2O are truth preserving, the words are still not identical in meaning. Indeed, probably no two words are absolutely identical in meaning, and if we define synonymy as identical meanings and connotations in all contexts, there are probably no absolute synonyms. Besides propositional meaning, many other facets of meaning that distinguish these words are important. For example, H_2O is used in scientific contexts and would be inappropriate in a hiking guide; this difference in genre is part of the meaning of the word. In practice, the word *synonym* is therefore commonly used to describe a relationship of approximate or rough synonymy.

Synonymy is actually a relationship between senses rather than words. Considering the words *big* and *large*. These may seem to be synonyms in the following ATIS sentences, since we could swap *big* and *large* in either sentence and retain the same meaning:

(19.13) How big is that plane?

(19.14) Would I be flying on a large or small plane?

But note the following WSJ sentence in which we cannot substitute *large* for *big*:

(19.15) Miss Nelson, for instance, became a kind of big sister to Benjamin.

(19.16) ?Miss Nelson, for instance, became a kind of large sister to Benjamin.

This is because the word *big* has a sense that means being older or grown up, while *large* lacks this sense. Thus, we say that some senses of *big* and *large* are (nearly) synonymous while other ones are not.

antonym Synonyms are words with identical or similar meanings. **Antonyms**, by contrast, are words with opposite meaning such as the following:

long/short	*big/little*	*fast/slow*	*cold/hot*	*dark/light*
rise/fall	*up/down*	*in/out*		

Two senses can be antonyms if they define a binary opposition or are at opposite ends of some scale. This is the case for *long/short*, *fast/slow*, or *big/little*, which are at opposite ends of the *length* or *size* scale. Another group of antonyms, **reversives**, describe change or movement in opposite directions, such as *rise/fall* or *up/down*.

reversives

Antonyms thus differ completely with respect to one aspect of their meaning—their position on a scale or their direction—but are otherwise very similar, sharing almost all other aspects of meaning. Thus, automatically distinguishing synonyms from antonyms can be difficult.

19.2.2 Hyponymy

hyponym One sense is a **hyponym** of another sense if the first sense is more specific, denoting a subclass of the other. For example, *car* is a hyponym of *vehicle*; *dog* is a hyponym of *animal*, and *mango* is a hyponym of *fruit*. Conversely, we say that *vehicle* is a **hypernym** of *car*, and *animal* is a hypernym of *dog*. It is unfortunate that the two words (hypernym and hyponym) are very similar and hence easily confused; for this reason, the word **superordinate** is often used instead of **hypernym**.

hypernym

superordinate

Superordinate	vehicle	fruit	furniture	mammal
Hyponym	car	mango	chair	dog

We can define hypernymy more formally by saying that the class denoted by the superordinate extensionally includes the class denoted by the hyponym. Thus, the class of animals includes as members all dogs, and the class of moving actions includes all walking actions. Hypernymy can also be defined in terms of **entailment**. Under this definition, a sense A is a hyponym of a sense B if everything that is A is also B, and hence being an A entails being a B, or $\forall x \;\; A(x) \Rightarrow B(x)$. Hyponymy is usually a transitive relation; if A is a hyponym of B and B is a hyponym of C, then A is a hyponym of C. Another name for the hypernym/hyponym structure is the **IS-A** hierarchy, in which we say A IS-A B, or B **subsumes A**.

IS-A

meronymy **Meronymy** Another common relation is **meronymy**, the **part-whole** relation. A *leg* is part of a *chair*; a *wheel* is part of a *car*. We say that *wheel* is a **meronym** of *car*, and *car* is a **holonym** of *wheel*.

part-whole

meronym

holonym

19.3 WordNet: A Database of Lexical Relations

WordNet The most commonly used resource for English sense relations is the **WordNet** lexical database (Fellbaum, 1998). WordNet consists of three separate databases, one each for nouns and verbs and a third for adjectives and adverbs; closed class words are not included. Each database contains a set of lemmas, each one annotated with a set of senses. The WordNet 3.0 release has 117,798 nouns, 11,529 verbs, 22,479 adjectives, and 4,481 adverbs. The average noun has 1.23 senses, and the average verb has 2.16 senses. WordNet can be accessed on the Web or downloaded and accessed locally. Figure 19.1 shows the lemma entry for the noun and adjective *bass*.

The noun "bass" has 8 senses in WordNet.
1. bass^1 - (the lowest part of the musical range)
2. bass^2, bass part^1 - (the lowest part in polyphonic music)
3. bass^3, basso^1 - (an adult male singer with the lowest voice)
4. sea bass^1, bass^4 - (the lean flesh of a saltwater fish of the family Serranidae)
5. freshwater bass^1, bass^5 - (any of various North American freshwater fish with lean flesh (especially of the genus Micropterus))
6. bass^6, bass voice^1, basso^2 - (the lowest adult male singing voice)
7. bass^7 - (the member with the lowest range of a family of musical instruments)
8. bass^8 - (nontechnical name for any of numerous edible marine and freshwater spiny-finned fishes)

The adjective "bass" has 1 sense in WordNet.
1. bass^1, deep^6 - (having or denoting a low vocal or instrumental range)
"a deep voice"; "a bass voice is lower than a baritone voice"; "a bass clarinet"

Figure 19.1 A portion of the WordNet 3.0 entry for the noun *bass*.

gloss Note that there are eight senses for the noun and one for the adjective, each of which has a **gloss** (a dictionary-style definition), a list of synonyms for the sense, and sometimes also usage examples (shown for the adjective sense). Unlike dictionaries, WordNet doesn't represent pronunciation, so doesn't distinguish the pronunciation [b ae s] in **bass^4**, **bass^5**, and **bass^8** from the other senses pronounced [b ey s].

synset The set of near-synonyms for a WordNet sense is called a **synset** (for **synonym set**); synsets are an important primitive in WordNet. The entry for *bass* includes synsets like {*bass*^{1}, *deep*^{6}}, or {*bass*^{6}, *bass voice*^{1}, *basso*^{2}}. We can think of a synset as representing a concept of the type we discussed in Chapter 14. Thus, instead of representing concepts in logical terms, WordNet represents them as lists of the word senses that can be used to express the concept. Here's another synset example:

{chump^1, fool^2, gull^1, mark^9, patsy^1, fall guy^1,
sucker^1, soft touch^1, mug^2}

The gloss of this synset describes it as *a person who is gullible and easy to take advantage of*. Each of the lexical entries included in the synset can, therefore, be used to express this concept. Synsets like this one actually constitute the senses associated with WordNet entries, and hence it is synsets, not wordforms, lemmas, or individual senses, that participate in most of the lexical sense relations in WordNet.

WordNet represents all the kinds of sense relations discussed in the previous section, as illustrated in Fig. 19.2 and Fig. 19.3. WordNet hyponymy relations cor-

Relation	Also Called	Definition	Example
Hypernym	Superordinate	From concepts to superordinates	$\textit{breakfast}^1 \rightarrow \textit{meal}^1$
Hyponym	Subordinate	From concepts to subtypes	$\textit{meal}^1 \rightarrow \textit{lunch}^1$
Instance Hypernym	Instance	From instances to their concepts	$\textit{Austen}^1 \rightarrow \textit{author}^1$
Instance Hyponym	Has-Instance	From concepts to concept instances	$\textit{composer}^1 \rightarrow \textit{Bach}^1$
Member Meronym	Has-Member	From groups to their members	$\textit{faculty}^2 \rightarrow \textit{professor}^1$
Member Holonym	Member-Of	From members to their groups	$\textit{copilot}^1 \rightarrow \textit{crew}^1$
Part Meronym	Has-Part	From wholes to parts	$\textit{table}^2 \rightarrow \textit{leg}^3$
Part Holonym	Part-Of	From parts to wholes	$\textit{course}^7 \rightarrow \textit{meal}^1$
Substance Meronym		From substances to their subparts	$\textit{water}^1 \rightarrow \textit{oxygen}^1$
Substance Holonym		From parts of substances to wholes	$\textit{gin}^1 \rightarrow \textit{martini}^1$
Antonym		Semantic opposition between lemmas	$\textit{leader}^1 \Longleftrightarrow \textit{follower}^1$
Derivationally Related Form		Lemmas w/same morphological root	$\textit{destruction}^1 \Longleftrightarrow \textit{destroy}^1$

Figure 19.2 Noun relations in WordNet.

Relation	Definition	Example
Hypernym	From events to superordinate events	$\textit{fly}^9 \rightarrow \textit{travel}^5$
Troponym	From events to subordinate event (often via specific manner)	$\textit{walk}^1 \rightarrow \textit{stroll}^1$
Entails	From verbs (events) to the verbs (events) they entail	$\textit{snore}^1 \rightarrow \textit{sleep}^1$
Antonym	Semantic opposition between lemmas	$\textit{increase}^1 \Longleftrightarrow \textit{decrease}^1$
Derivationally Related Form	Lemmas with same morphological root	$\textit{destroy}^1 \Longleftrightarrow \textit{destruction}^1$

Figure 19.3 Verb relations in WordNet.

respond to the notion of immediate hyponymy discussed on page 330. Each synset is related to its immediately more general and more specific synsets through direct hypernym and hyponym relations. These relations can be followed to produce longer chains of more general or more specific synsets. Figure 19.4 shows hypernym chains for **bass^3** and **bass^7**.

In this depiction of hyponymy, successively more general synsets are shown on successive indented lines. The first chain starts from the concept of a human bass singer. Its immediate superordinate is a synset corresponding to the generic concept of a singer. Following this chain leads eventually to concepts such as *entertainer* and *person*. The second chain, which starts from musical instrument, has a completely different path leading eventually to such concepts as musical instrument, device, and physical object. Both paths do eventually join at the very abstract synset *whole, unit*, and then proceed together to *entity* which is the top (root) of the noun hierarchy (in WordNet this root is generally called the **unique beginner**).

unique beginner

19.4 Word Sense Disambiguation: Overview

Our discussion of compositional semantic analyzers in Chapter 15 pretty much ignored the issue of lexical ambiguity. It should be clear by now that this is an unreasonable approach. Without some means of selecting correct senses for the words in an input, the enormous amount of homonymy and polysemy in the lexicon would quickly overwhelm any approach in an avalanche of competing interpretations.

```
Sense 3
bass, basso --
(an adult male singer with the lowest voice)
=> singer, vocalist, vocalizer, vocaliser
   => musician, instrumentalist, player
      => performer, performing artist
         => entertainer
            => person, individual, someone...
               => organism, being
                  => living thing, animate thing,
                     => whole, unit
                        => object, physical object
                           => physical entity
                              => entity
               => causal agent, cause, causal agency
                  => physical entity
                     => entity

Sense 7
bass --
(the member with the lowest range of a family of
musical instruments)
=> musical instrument, instrument
   => device
      => instrumentality, instrumentation
         => artifact, artefact
            => whole, unit
               => object, physical object
                  => physical entity
                     => entity
```

Figure 19.4 Hyponymy chains for two separate senses of the lemma *bass*. Note that the chains are completely distinct, only converging at the very abstract level *whole, unit*.

word sense disambiguation
WSD

The task of selecting the correct sense for a word is called **word sense disambiguation**, or **WSD**. Disambiguating word senses has the potential to improve many natural language processing tasks, including **machine translation**, **question answering**, and **information retrieval**.

WSD algorithms take as input a word in context along with a fixed inventory of potential word senses and return as output the correct word sense for that use. The input and the senses depends on the task. For machine translation from English to Spanish, the sense tag inventory for an English word might be the set of different Spanish translations. If our task is automatic indexing of medical articles, the sense-tag inventory might be the set of MeSH (Medical Subject Headings) thesaurus entries.

When we are evaluating WSD in isolation, we can use the set of senses from a dictionary/thesaurus resource like WordNet. Figure 19.4 shows an example for the word *bass*, which can refer to a musical instrument or a kind of fish.[2]

lexical sample

It is useful to distinguish two variants of the generic WSD task. In the **lexical sample** task, a small pre-selected set of target words is chosen, along with an inventory of senses for each word from some lexicon. Since the set of words and

[2] The WordNet database includes eight senses; we have arbitrarily selected two for this example; we have also arbitrarily selected one of the many Spanish fishes that could translate English *sea bass*.

WordNet Sense	Spanish Translation	Roget Category	Target Word in Context
bass^4	lubina	FISH/INSECT	... fish as Pacific salmon and striped **bass** and...
bass^4	lubina	FISH/INSECT	... produce filets of smoked **bass** or sturgeon...
bass^7	bajo	MUSIC	... exciting jazz **bass** player since Ray Brown...
bass^7	bajo	MUSIC	... play **bass** because he doesn't have to solo...

Figure 19.5 Possible definitions for the inventory of sense tags for *bass*.

the set of senses are small, supervised machine learning approaches are often used to handle lexical sample tasks. For each word, a number of corpus instances (context sentences) can be selected and hand-labeled with the correct sense of the target word in each. Classifier systems can then be trained with these labeled examples. Unlabeled target words in context can then be labeled using such a trained classifier. Early work in word sense disambiguation focused solely on lexical sample tasks of this sort, building word-specific algorithms for disambiguating single words like *line*, *interest*, or *plant*.

all-words

In contrast, in the **all-words** task, systems are given entire texts and a lexicon with an inventory of senses for each entry and are required to disambiguate every content word in the text. The all-words task is similar to part-of-speech tagging, except with a much larger set of tags since each lemma has its own set. A consequence of this larger set of tags is a serious data sparseness problem; it is unlikely that adequate training data for every word in the test set will be available. Moreover, given the number of polysemous words in reasonably sized lexicons, approaches based on training one classifier per term are unlikely to be practical.

In the following sections we explore the application of various machine learning paradigms to word sense disambiguation.

19.5 Supervised Word Sense Disambiguation

If we have data that has been hand-labeled with correct word senses, we can use a **supervised learning** approach to the problem of sense disambiguation—extracting features from the text and training a classifier to assign the correct sense given these features. The output of training is thus a classifier system capable of assigning sense labels to unlabeled words in context.

For **lexical sample** tasks, there are various labeled corpora for individual words; these corpora consist of context sentences labeled with the correct sense for the target word. These include the *line-hard-serve* corpus containing 4,000 sense-tagged examples of *line* as a noun, *hard* as an adjective and *serve* as a verb (Leacock et al., 1993), and the *interest* corpus with 2,369 sense-tagged examples of *interest* as a noun (Bruce and Wiebe, 1994). The SENSEVAL project has also produced a number of such sense-labeled lexical sample corpora (SENSEVAL-1 with 34 words from the HECTOR lexicon and corpus (Kilgarriff and Rosenzweig 2000, Atkins 1993), SENSEVAL-2 and -3 with 73 and 57 target words, respectively (Palmer et al. 2001, Kilgarriff 2001).

semantic concordance

For training **all-word** disambiguation tasks we use a **semantic concordance**, a corpus in which each open-class word in each sentence is labeled with its word sense from a specific dictionary or thesaurus. One commonly used corpus is SemCor, a subset of the Brown Corpus consisting of over 234,000 words that were man-

ually tagged with WordNet senses (Miller et al. 1993, Landes et al. 1998). In addition, sense-tagged corpora have been built for the SENSEVAL all-word tasks. The SENSEVAL-3 English all-words test data consisted of 2081 tagged content word tokens, from 5,000 total running words of English from the WSJ and Brown corpora (Palmer et al., 2001).

The first step in supervised training is to extract features that are predictive of word senses. The insight that underlies all modern algorithms for word sense disambiguation was famously first articulated by Weaver (1955) in the context of machine translation:

> If one examines the words in a book, one at a time as through an opaque mask with a hole in it one word wide, then it is obviously impossible to determine, one at a time, the meaning of the words. [...] But if one lengthens the slit in the opaque mask, until one can see not only the central word in question but also say N words on either side, then if N is large enough one can unambiguously decide the meaning of the central word. [...] The practical question is : "What minimum value of N will, at least in a tolerable fraction of cases, lead to the correct choice of meaning for the central word?"

We first perform some processing on the sentence containing the window, typically including part-of-speech tagging, lemmatization, and, in some cases, syntactic parsing to reveal headwords and dependency relations. Context features relevant to the target word can then be extracted from this enriched input. A **feature vector** consisting of numeric or nominal values encodes this linguistic information as an input to most machine learning algorithms.

feature vector

Two classes of features are generally extracted from these neighboring contexts, both of which we have seen previously in part-of-speech tagging: collocational features and bag-of-words features. A **collocation** is a word or series of words in a position-specific relationship to a target word (i.e., exactly one word to the right, or the two words starting 3 words to the left, and so on). Thus, **collocational features** encode information about *specific* positions located to the left or right of the target word. Typical features extracted for these context words include the word itself, the root form of the word, and the word's part-of-speech. Such features are effective at encoding local lexical and grammatical information that can often accurately isolate a given sense.

collocation

collocational features

For example consider the ambiguous word *bass* in the following WSJ sentence:

(19.17) An electric guitar and **bass** player stand off to one side, not really part of the scene, just as a sort of nod to gringo expectations perhaps.

A collocational feature vector, extracted from a window of two words to the right and left of the target word, made up of the words themselves, their respective parts-of-speech, and pairs of words, that is,

$$[w_{i-2}, \mathrm{POS}_{i-2}, w_{i-1}, \mathrm{POS}_{i-1}, w_{i+1}, \mathrm{POS}_{i+1}, w_{i+2}, \mathrm{POS}_{i+2}, w_{i-2}^{i-1}, w_{i+1}^{i+2}] \quad (19.18)$$

would yield the following vector:

```
[guitar, NN, and, CC, player, NN, stand, VB, and guitar, player stand]
```

High performing systems generally use POS tags and word collocations of length 1, 2, and 3 from a window of words 3 to the left and 3 to the right (Zhong and Ng, 2010).

The second type of feature consists of **bag-of-words** information about neighboring words. A **bag-of-words** means an unordered set of words, with their exact

bag-of-words

position ignored. The simplest bag-of-words approach represents the context of a target word by a vector of features, each binary feature indicating whether a vocabulary word *w* does or doesn't occur in the context.

This vocabulary is typically pre-selected as some useful subset of words in a training corpus. In most WSD applications, the context region surrounding the target word is generally a small, symmetric, fixed-size window with the target word at the center. Bag-of-word features are effective at capturing the general topic of the discourse in which the target word has occurred. This, in turn, tends to identify senses of a word that are specific to certain domains. We generally don't use stopwords, punctuation, or number as features, and words are lemmatized and lower-cased. In some cases we may also limit the bag-of-words to consider only frequently used words. For example, a bag-of-words vector consisting of the 12 most frequent content words from a collection of *bass* sentences drawn from the WSJ corpus would have the following ordered word feature set:

[*fishing, big, sound, player, fly, rod, pound, double, runs, playing, guitar, band*]

Using these word features with a window size of 10, (19.17) would be represented by the following binary vector:

`[0,0,0,1,0,0,0,0,0,0,1,0]`

Given training data together with the extracted features, any supervised machine learning paradigm can be used to train a sense classifier.

19.5.1 Wikipedia as a source of training data

Supervised methods for WSD are very dependent on the amount of training data, especially because of their reliance on sparse lexical and collocation features. One way to increase the amount of training data is to use Wikipedia as a source of sense-labeled data. When a concept is mentioned in a Wikipedia article, the article text may contain an explicit link to the concept's Wikipedia page, which is named by a unique identifier. This link can be used as a sense annotation. For example, the ambiguous word *bar* is linked to a different Wikipedia article depending on its meaning in context, including the page BAR (LAW), the page BAR (MUSIC), and so on, as in the following Wikipedia examples (Mihalcea, 2007).

> In 1834, Sumner was admitted to the **[[bar (law)|bar]]** at the age of twenty-three, and entered private practice in Boston.
>
> It is danced in 3/4 time (like most waltzes), with the couple turning approx. 180 degrees every **[[bar (music)|bar]]**.
>
> Jenga is a popular beer in the **[[bar (establishment)|bar]]**s of Thailand.

These sentences can then be added to the training data for a supervised system. In order to use Wikipedia in this way, however, it is necessary to map from Wikipedia concepts to whatever inventory of senses is relevant for the WSD application. Automatic algorithms that map from Wikipedia to WordNet, for example, involve finding the WordNet sense that has the greatest lexical overlap with the Wikipedia sense, by comparing the vector of words in the WordNet synset, gloss, and related senses with the vector of words in the Wikipedia page title, outgoing links, and page category (Ponzetto and Navigli, 2010).

19.5.2 Evaluation

extrinsic evaluation

To evaluate WSD algorithms, it's better to consider **extrinsic**, **task-based**, or **end-**

to-end evaluation, in which we see whether some new WSD idea actually improves performance in some end-to-end application like question answering or machine translation. Nonetheless, because extrinsic evaluations are difficult and slow, WSD systems are typically evaluated with **intrinsic** evaluation. in which a WSD component is treated as an independent system. Common intrinsic evaluations are either exact-match **sense accuracy**—the percentage of words that are tagged identically with the hand-labeled sense tags in a test set—or with precision and recall if systems are permitted to pass on the labeling of some instances. In general, we evaluate by using held-out data from the same sense-tagged corpora that we used for training, such as the SemCor corpus discussed above or the various corpora produced by the SENSEVAL effort.

intrinsic

sense accuracy

Many aspects of sense evaluation have been standardized by the SENSEVAL and SEMEVAL efforts (Palmer et al. 2006, Kilgarriff and Palmer 2000). This framework provides a shared task with training and testing materials along with sense inventories for all-words and lexical sample tasks in a variety of languages.

most frequent sense

The normal baseline is to choose the **most frequent sense** for each word from the senses in a labeled corpus (Gale et al., 1992a). For WordNet, this corresponds to the first sense, since senses in WordNet are generally ordered from most frequent to least frequent. WordNet sense frequencies come from the SemCor sense-tagged corpus described above– WordNet senses that don't occur in SemCor are ordered arbitrarily after those that do. The most frequent sense baseline can be quite accurate, and is therefore often used as a default, to supply a word sense when a supervised algorithm has insufficient training data.

19.6 WSD: Dictionary and Thesaurus Methods

Supervised algorithms based on sense-labeled corpora are the best-performing algorithms for sense disambiguation. However, such labeled training data is expensive and limited. One alternative is to get indirect supervision from dictionaries and thesauruses, and so this method is also called **knowledge-based** WSD. Methods like this that do not use texts that have been hand-labeled with senses are also called weakly supervised.

19.6.1 The Lesk Algorithm

Lesk algorithm

Simplified Lesk

The most well-studied dictionary-based algorithm for sense disambiguation is the **Lesk algorithm**, really a family of algorithms that choose the sense whose dictionary gloss or definition shares the most words with the target word's neighborhood. Figure 19.6 shows the simplest version of the algorithm, often called the **Simplified Lesk** algorithm (Kilgarriff and Rosenzweig, 2000).

As an example of the Lesk algorithm at work, consider disambiguating the word *bank* in the following context:

(19.19) The **bank** can guarantee deposits will eventually cover future tuition costs because it invests in adjustable-rate mortgage securities.

given the following two WordNet senses:

```
function SIMPLIFIED LESK(word, sentence) returns best sense of word

 best-sense ← most frequent sense for word
 max-overlap ← 0
 context ← set of words in sentence
 for each sense in senses of word do
  signature ← set of words in the gloss and examples of sense
  overlap ← COMPUTEOVERLAP(signature, context)
  if overlap > max-overlap then
      max-overlap ← overlap
      best-sense ← sense
 end
 return(best-sense)
```

Figure 19.6 The Simplified Lesk algorithm. The COMPUTEOVERLAP function returns the number of words in common between two sets, ignoring function words or other words on a stop list. The original Lesk algorithm defines the *context* in a more complex way. The *Corpus Lesk* algorithm weights each overlapping word w by its $-\log P(w)$ and includes labeled training corpus data in the *signature*.

bank^1	Gloss:	a financial institution that accepts deposits and channels the money into lending activities
	Examples:	"he cashed a check at the bank", "that bank holds the mortgage on my home"
bank^2	Gloss:	sloping land (especially the slope beside a body of water)
	Examples:	"they pulled the canoe up on the bank", "he sat on the bank of the river and watched the currents"

Sense $\textbf{bank}^1$ has two non-stopwords overlapping with the context in (19.19): *deposits* and *mortgage*, while sense bank^2 has zero words, so sense $\textbf{bank}^1$ is chosen.

There are many obvious extensions to Simplified Lesk. The original Lesk algorithm (Lesk, 1986) is slightly more indirect. Instead of comparing a target word's signature with the context words, the target signature is compared with the signatures of each of the context words. For example, consider Lesk's example of selecting the appropriate sense of *cone* in the phrase *pine cone* given the following definitions for *pine* and *cone*.

pine	1	kinds of evergreen tree with needle-shaped leaves
	2	waste away through sorrow or illness
cone	1	solid body which narrows to a point
	2	something of this shape whether solid or hollow
	3	fruit of certain evergreen trees

In this example, Lesk's method would select $\textbf{cone}^3$ as the correct sense since two of the words in its entry, *evergreen* and *tree*, overlap with words in the entry for *pine*, whereas neither of the other entries has any overlap with words in the definition of *pine*. In general Simplified Lesk seems to work better than original Lesk.

The primary problem with either the original or simplified approaches, however, is that the dictionary entries for the target words are short and may not provide enough chance of overlap with the context.[3] One remedy is to expand the list of words used in the classifier to include words related to, but not contained in, their

[3] Indeed, Lesk (1986) notes that the performance of his system seems to roughly correlate with the length of the dictionary entries.

individual sense definitions. But the best solution, if any sense-tagged corpus data like SemCor is available, is to add all the words in the labeled corpus sentences for a word sense into the signature for that sense. This version of the algorithm, the **Corpus Lesk** algorithm, is the best-performing of all the Lesk variants (Kilgarriff and Rosenzweig 2000, Vasilescu et al. 2004) and is used as a baseline in the SENSEVAL competitions. Instead of just counting up the overlapping words, the **Corpus Lesk** algorithm also applies a weight to each overlapping word. The weight is the **inverse document frequency** or **IDF**, a standard information-retrieval measure introduced in Chapter 6. IDF measures how many different "documents" (in this case, glosses and examples) a word occurs in and is thus a way of discounting function words. Since function words like *the*, *of*, etc., occur in many documents, their IDF is very low, while the IDF of content words is high. Corpus Lesk thus uses IDF instead of a stop list.

Corpus Lesk

inverse document frequency
IDF

Formally, the IDF for a word i can be defined as

$$\text{idf}_i = \log\left(\frac{Ndoc}{nd_i}\right) \tag{19.20}$$

where $Ndoc$ is the total number of "documents" (glosses and examples) and nd_i is the number of these documents containing word i.

Finally, we can combine the Lesk and supervised approaches by adding new Lesk-like bag-of-words features. For example, the glosses and example sentences for the target sense in WordNet could be used to compute the supervised bag-of-words features in addition to the words in the SemCor context sentence for the sense (Yuret, 2004).

19.6.2 Graph-based Methods

Another way to use a thesaurus like WordNet is to make use of the fact that WordNet can be construed as a graph, with senses as nodes and relations between senses as edges. In addition to the hypernymy and other relations, it's possible to create links between senses and those words in the gloss that are unambiguous (have only one sense). Often the relations are treated as undirected edges, creating a large undirected WordNet graph. Fig. 19.7 shows a portion of the graph around the word $drink^1_v$.

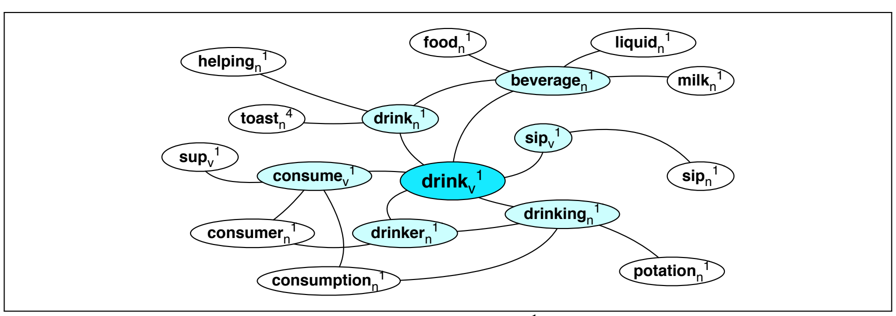


Figure 19.7 Part of the WordNet graph around drink^1_v, after Navigli and Lapata (2010)
.

There are various ways to use the graph for disambiguation, some using the whole graph, some using only a subpart. For example the target word and the words in its sentential context can all be inserted as nodes in the graph via a directed edge

to each of its senses. If we consider the sentence *She drank some milk*, Fig. 19.8 shows a portion of the WordNet graph between the senses drink_v^1 and milk_n^1.

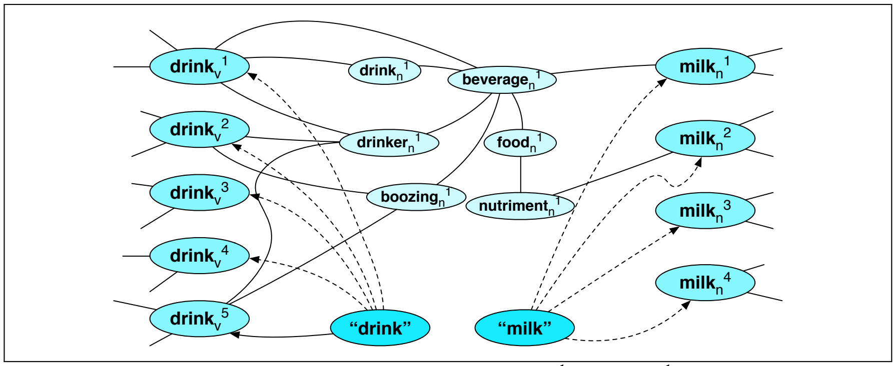


Figure 19.8 Part of the WordNet graph between drink_v^1 and milk_n^1, for disambiguating a sentence like *She drank some milk*, adapted from Navigli and Lapata (2010)
.

degree

personalized page rank

The correct sense is then the one which is the most important or *central* in some way in this graph. There are many different methods for deciding centrality. The simplest is **degree**, the number of edges into the node, which tends to correlate with the most frequent sense. Another algorithm for assigning probabilities across nodes is **personalized page rank**, a version of the well-known pagerank algorithm which uses some seed nodes. By inserting a uniform probability across the word nodes (*drink* and *milk* in the example) and computing the personalized page rank of the graph, the result will be a pagerank value for each node in the graph, and the sense with the maximum pagerank can then be chosen. See Agirre et al. (2014) and Navigli and Lapata (2010) for details.

19.7 Semi-Supervised WSD: Bootstrapping

bootstrapping

Yarowsky algorithm

Both the supervised approach and the dictionary-based approaches to WSD require large hand-built resources: supervised training sets in one case, large dictionaries in the other. We can instead use **bootstrapping** or **semi-supervised learning**, which needs only a very small hand-labeled training set.

A classic bootstrapping algorithm for WSD is the **Yarowsky algorithm** for learning a classifier for a target word (in a lexical-sample task) (Yarowsky, 1995). The algorithm is given a small seedset Λ_0 of labeled instances of each sense and a much larger unlabeled corpus V_0. The algorithm first trains an initial classifier on the seedset Λ_0. It then uses this classifier to label the unlabeled corpus V_0. The algorithm then selects the examples in V_0 that it is most confident about, removes them, and adds them to the training set (call it now Λ_1). The algorithm then trains a new classifier (a new set of rules) on Λ_1, and iterates by applying the classifier to the now-smaller unlabeled set V_1, extracting a new training set Λ_2, and so on. With each iteration of this process, the training corpus grows and the untagged corpus shrinks. The process is repeated until some sufficiently low error-rate on the training set is reached or until no further examples from the untagged corpus are above threshold.

one sense per collocation

Initial seeds can be selected by hand-labeling a small set of examples (Hearst, 1991), or by using the help of a heuristic. Yarowsky (1995) used the **one sense per collocation** heuristic, which relies on the intuition that certain words or phrases

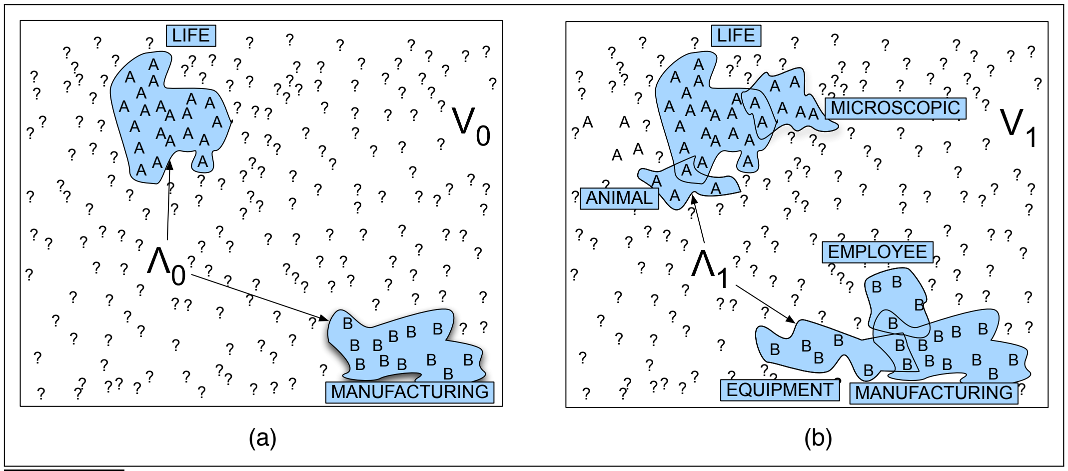


Figure 19.9 The Yarowsky algorithm disambiguating "plant" at two stages; "?" indicates an unlabeled observation, A and B are observations labeled as SENSE-A or SENSE-B. The initial stage (a) shows only seed sentences Λ_0 labeled by collocates ("life" and "manufacturing"). An intermediate stage is shown in (b) where more collocates have been discovered ("equipment", "microscopic", etc.) and more instances in V_0 have been moved into Λ_1, leaving a smaller unlabeled set V_1. Figure adapted from Yarowsky (1995).

We need more good teachers – right now, there are only a half a dozen who can **play** the free **bass** with ease. An electric guitar and **bass play**er stand off to one side, not really part of the scene, just as a sort of nod to gringo expectations perhaps.
The researchers said the worms spend part of their life cycle in such **fish** as Pacific salmon and striped **bass** and Pacific rockfish or snapper. And it all started when **fish**ermen decided the striped **bass** in Lake Mead were too skinny.

Figure 19.10 Samples of *bass* sentences extracted from the WSJ by using the simple correlates *play* and *fish*.

strongly associated with the target senses tend not to occur with the other sense. Yarowsky defines his seedset by choosing a single collocation for each sense.

For example, to generate seed sentences for the fish and musical musical senses of *bass*, we might come up with *fish* as a reasonable indicator of $\textbf{bass}^1$ and *play* as a reasonable indicator of $\textbf{bass}^2$. Figure 19.10 shows a partial result of such a search for the strings "fish" and "play" in a corpus of *bass* examples drawn from the WSJ.

one sense per discourse

The original Yarowsky algorithm also makes use of a second heuristic, called **one sense per discourse**, based on the work of Gale et al. (1992b), who noticed that a particular word appearing multiple times in a text or discourse often appeared with the same sense. This heuristic seems to hold better for coarse-grained senses and particularly for cases of homonymy rather than polysemy (Krovetz, 1998).

Nonetheless, it is still useful in a number of sense disambiguation situations. In fact, the *one sense per discourse* heuristic is an important one throughout language processing as it seems that many disambiguation tasks may be improved by a bias toward resolving an ambiguity the same way inside a discourse segment.

19.8 Unsupervised Word Sense Induction

It is expensive and difficult to build large corpora in which each word is labeled for its word sense. For this reason, an unsupervised approach to sense disambiguation, often called **word sense induction** or **WSI**, is an exciting and important research area. In unsupervised approaches, we don't use human-defined word senses. Instead, the set of "senses" of each word is created automatically from the instances of each word in the training set.

word sense induction

Most algorithms for word sense induction use some sort of clustering. For example, the early algorithm of Schütze (Schütze 1992b, Schütze 1998) represented each word as a context vector of bag-of-words features $\vec{c}$. (See Chapter 6 for a more complete introduction to such **vector** models of meaning.) Then in training, we use three steps.

1. For each token w_i of word w in a corpus, compute a context vector $\vec{c}$.
2. Use a **clustering algorithm** to **cluster** these word-token context vectors $\vec{c}$ into a predefined number of groups or clusters. Each cluster defines a sense of w.
3. Compute the **vector centroid** of each cluster. Each vector centroid $\vec{s}_j$ is a **sense vector** representing that sense of w.

Since this is an unsupervised algorithm, we don't have names for each of these "senses" of w; we just refer to the jth sense of w.

Now how do we disambiguate a particular token t of w? Again, we have three steps:

1. Compute a context vector $\vec{c}$ for t.
2. Retrieve all sense vectors s_j for w.
3. Assign t to the sense represented by the sense vector s_j that is closest to t.

All we need is a clustering algorithm and a distance metric between vectors. Clustering is a well-studied problem with a wide number of standard algorithms that can be applied to inputs structured as vectors of numerical values (Duda and Hart, 1973). A frequently used technique in language applications is known as **agglomerative clustering**. In this technique, each of the N training instances is initially assigned to its own cluster. New clusters are then formed in a bottom-up fashion by the successive merging of the two clusters that are most similar. This process continues until either a specified number of clusters is reached, or some global goodness measure among the clusters is achieved. In cases in which the number of training instances makes this method too expensive, random sampling can be used on the original training set to achieve similar results.

agglomerative clustering

topic modeling

LDA

Recent algorithms have also used **topic modeling** algorithms like **Latent Dirichlet Allocation** (**LDA**), another way to learn clusters of words based on their distributions (Lau et al., 2012).

How can we evaluate unsupervised sense disambiguation approaches? As usual, the best way is to do extrinsic evaluation embedded in some end-to-end system; one example used in a **SemEval** bakeoff is to improve search result clustering and diversification (Navigli and Vannella, 2013). Intrinsic evaluation requires a way to map the automatically derived sense classes into a hand-labeled gold-standard set so that we can compare a hand-labeled test set with a set labeled by our unsupervised classifier. Various such metrics have been tested, for example in the SemEval tasks (Manandhar et al. 2010, Navigli and Vannella 2013, Jurgens and Klapaftis 2013),

including cluster overlap metrics, or methods that map each sense cluster to a predefined sense by choosing the sense that (in some training set) has the most overlap with the cluster. However it is fair to say that no evaluation metric for this task has yet become standard.

19.9 Word Similarity: Thesaurus Methods

We turn now to the computation of various semantic relations that hold between words. We saw in Section 19.2 that such relations include synonymy, antonymy, hyponymy, hypernymy, and meronymy. Of these, the one that has been most computationally developed and has the greatest number of applications is the idea of word **synonymy** and **similarity**.

Synonymy is a binary relation between words; two words are either synonyms or not. For most computational purposes, we use instead a looser metric of **word similarity** or **semantic distance**. Two words are more similar if they share more features of meaning or are near-synonyms. Two words are less similar, or have greater semantic distance, if they have fewer common meaning elements. Although we have described them as relations between words, synonymy, similarity, and distance are actually relations between word *senses*. For example, of the two senses of *bank*, we might say that the financial sense is similar to one of the senses of *fund* and the riparian sense is more similar to one of the senses of *slope*. In the next few sections of this chapter, we will compute these relations over both words and senses.

word similarity
semantic distance

The ability to compute word similarity is a useful part of many language understanding applications. In **information retrieval** or **question answering**, we might want to retrieve documents whose words have meanings similar to the query words. In **summarization**, **generation**, and **machine translation**, we need to know whether two words are similar to know if we can substitute one for the other in particular contexts. In **language modeling**, we can use semantic similarity to cluster words for class-based models. One interesting class of applications for word similarity is automatic grading of student responses. For example, algorithms for **automatic essay grading** use word similarity to determine if an essay is similar in meaning to a correct answer. We can also use word similarity as part of an algorithm to *take* an exam, such as a multiple-choice vocabulary test. Automatically taking exams is useful in test designs in order to see how easy or hard a particular multiple-choice question or exam is.

Two classes of algorithms can be used to measure word similarity. This chapter focuses on **thesaurus-based** algorithms, in which we measure the distance between two senses in an on-line thesaurus like WordNet or MeSH. The next chapter focuses on **distributional** algorithms, in which we estimate word similarity by finding words that have similar distributions in a corpus.

The thesaurus-based algorithms use the structure of the thesaurus to define word similarity. In principle, we could measure similarity by using any information available in a thesaurus (meronymy, glosses, etc.). In practice, however, thesaurus-based word similarity algorithms generally use only the hypernym/hyponym (*is-a* or subsumption) hierarchy. In WordNet, verbs and nouns are in separate hypernym hierarchies, so a thesaurus-based algorithm for WordNet can thus compute only noun-noun similarity, or verb-verb similarity; we can't compare nouns to verbs or do anything with adjectives or other parts of speech.

word relatedness

We can distinguish **word similarity** from **word relatedness**. Two words are

similar if they are near-synonyms or roughly substitutable in context. Word relatedness characterizes a larger set of potential relationships between words; antonyms, for example, have high relatedness but low similarity. The words *car* and *gasoline* are closely related but not similar, while *car* and *bicycle* are similar. Word similarity is thus a subcase of word relatedness. In general, the five algorithms we describe in this section do not attempt to distinguish between similarity and semantic relatedness; for convenience, we will call them *similarity* measures, although some would be more appropriately described as relatedness measures.

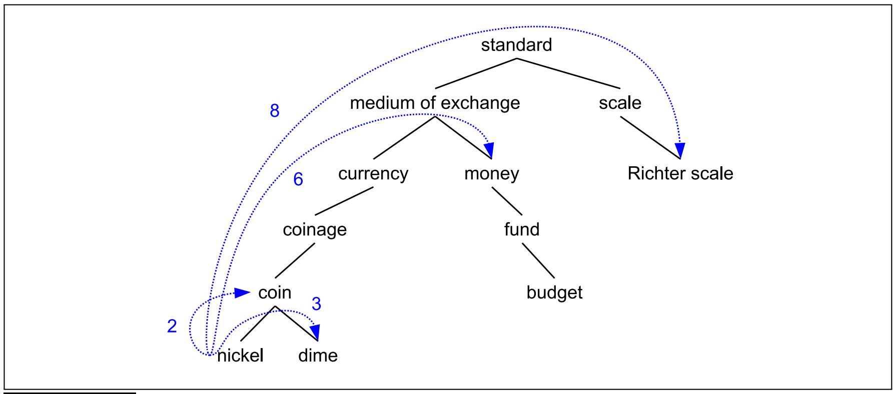


Figure 19.11 A fragment of the WordNet hypernym hierarchy, showing path lengths (number of edges plus 1) from *nickel* to *coin* (2), *dime* (3), *money* (6), and *Richter scale* (8).

The simplest thesaurus-based algorithms are based on the intuition that words or senses are more similar if there is a shorter **path** between them in the thesaurus graph, an intuition dating back to Quillian (1969). A word/sense is most similar to itself, then to its parents or siblings, and least similar to words that are far away. We make this notion operational by measuring the number of edges between the two concept nodes in the thesaurus graph and adding one. Figure 19.11 shows an intuition; the concept *dime* is most similar to *nickel* and *coin*, less similar to *money*, and even less similar to *Richter scale*. A formal definition:

> $\text{pathlen}(c_1, c_2) = 1$ + the number of edges in the shortest path in the thesaurus graph between the sense nodes c_1 and c_2

Path-based similarity can be defined as just the path length, transformed either by log (Leacock and Chodorow, 1998) or, more often, by an inverse, resulting in the following common definition of **path-length based similarity**:

path-length based similarity

$$\text{sim}_{\text{path}}(c_1, c_2) = \frac{1}{\text{pathlen}(c_1, c_2)} \tag{19.21}$$

For most applications, we don't have sense-tagged data, and thus we need our algorithm to give us the similarity between words rather than between senses or concepts. For any of the thesaurus-based algorithms, following Resnik (1995), we can approximate the correct similarity (which would require sense disambiguation) by just using the pair of senses for the two words that results in maximum sense similarity. Thus, based on sense similarity, we can define **word similarity** as follows:

word similarity

$$\text{wordsim}(w_1, w_2) = \max_{\substack{c_1 \in \text{senses}(w_1) \\ c_2 \in \text{senses}(w_2)}} \text{sim}(c_1, c_2) \tag{19.22}$$

The basic path-length algorithm makes the implicit assumption that each link in the network represents a uniform distance. In practice, this assumption is not appropriate. Some links (e.g., those that are deep in the WordNet hierarchy) often seem to represent an intuitively narrow distance, while other links (e.g., higher up in the WordNet hierarchy) represent an intuitively wider distance. For example, in Fig. 19.11, the distance from *nickel* to *money* (5) seems intuitively much shorter than the distance from *nickel* to an abstract word *standard*; the link between *medium of exchange* and *standard* seems wider than that between, say, *coin* and *coinage*.

It is possible to refine path-based algorithms with normalizations based on depth in the hierarchy (Wu and Palmer, 1994), but in general we'd like an approach that lets us independently represent the distance associated with each edge.

information-content

A second class of thesaurus-based similarity algorithms attempts to offer just such a fine-grained metric. These **information-content word-similarity** algorithms still rely on the structure of the thesaurus but also add probabilistic information derived from a corpus.

Following Resnik (1995) we'll define $P(c)$ as the probability that a randomly selected word in a corpus is an instance of concept c (i.e., a separate random variable, ranging over words, associated with each concept). This implies that $P(root) = 1$ since any word is subsumed by the root concept. Intuitively, the lower a concept in the hierarchy, the lower its probability. We train these probabilities by counting in a corpus; each word in the corpus counts as an occurrence of each concept that contains it. For example, in Fig. 19.11 above, an occurrence of the word *dime* would count toward the frequency of *coin*, *currency*, *standard*, etc. More formally, Resnik computes $P(c)$ as follows:

$$P(c) = \frac{\sum_{w \in \text{words}(c)} count(w)}{N} \tag{19.23}$$

where $\text{words}(c)$ is the set of words subsumed by concept c, and N is the total number of words in the corpus that are also present in the thesaurus.

Figure 19.12, from Lin (1998), shows a fragment of the WordNet concept hierarchy augmented with the probabilities $P(c)$.

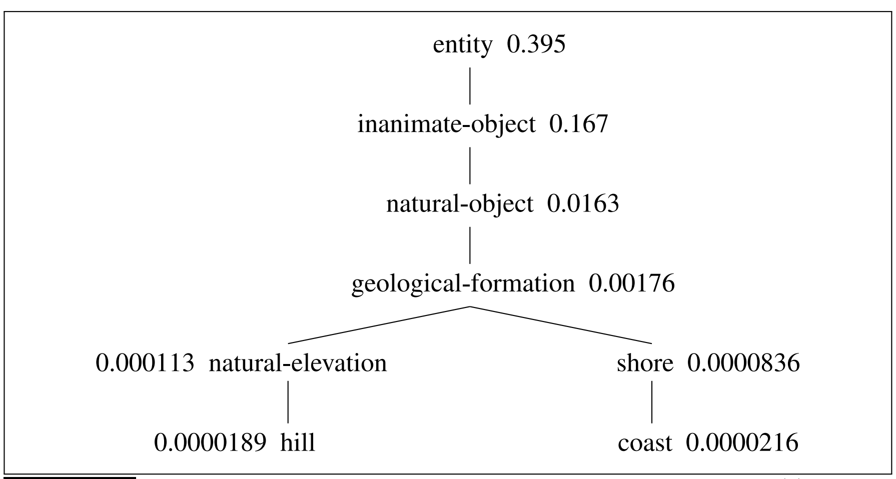


Figure 19.12 A fragment of the WordNet hierarchy, showing the probability $P(c)$ attached to each content, adapted from a figure from Lin (1998).

We now need two additional definitions. First, following basic information the-

ory, we define the information content (IC) of a concept c as

$$\text{IC}(c) = -\log P(c) \tag{19.24}$$

Lowest common subsumer LCS

Second, we define the **lowest common subsumer** or **LCS** of two concepts:

> $\text{LCS}(c_1, c_2)$ = the lowest common subsumer, that is, the lowest node in the hierarchy that subsumes (is a hypernym of) both c_1 and c_2

There are now a number of ways to use the information content of a node in a word similarity metric. The simplest way was first proposed by Resnik (1995). We think of the similarity between two words as related to their common information; the more two words have in common, the more similar they are. Resnik proposes to estimate the common amount of information by the **information content of the lowest common subsumer of the two nodes**. More formally, the **Resnik similarity** measure is

Resnik similarity

$$\text{sim}_{\text{resnik}}(c_1, c_2) = -\log P(\text{LCS}(c_1, c_2)) \tag{19.25}$$

Lin (1998) extended the Resnik intuition by pointing out that a similarity metric between objects A and B needs to do more than measure the amount of information in common between A and B. For example, he additionally pointed out that the more **differences** between A and B, the less similar they are. In summary:

- **Commonality:** the more information A and B have in common, the more similar they are.
- **Difference:** the more differences between the information in A and B, the less similar they are.

Lin measures the commonality between A and B as the information content of the proposition that states the commonality between A and B:

$$\text{IC}(\text{common(A,B)}) \tag{19.26}$$

He measures the difference between A and B as

$$\text{IC}(\text{description(A,B)}) - \text{IC}(\text{common(A,B)}) \tag{19.27}$$

where description(A,B) describes A and B. Given a few additional assumptions about similarity, Lin proves the following theorem:

> Similarity Theorem: The similarity between A and B is measured by the ratio between the amount of information needed to state the commonality of A and B and the information needed to fully describe what A and B are.

$$\text{sim}_{\text{Lin}}(A, B) = \frac{\text{common(A,B)}}{\text{description(A,B)}} \tag{19.28}$$

Applying this idea to the thesaurus domain, Lin shows (in a slight modification of Resnik's assumption) that the information in common between two concepts is twice the information in the lowest common subsumer $\text{LCS}(c_1, c_2)$. Adding in the above definitions of the information content of thesaurus concepts, the final **Lin similarity** function is

Lin similarity

$$\text{sim}_{\text{Lin}}(c_1, c_2) = \frac{2 \times \log P(LCS(c_1, c_2))}{\log P(c_1) + \log P(c_2)} \tag{19.29}$$

For example, using sim_{Lin}, Lin (1998) shows that the similarity between the concepts of *hill* and *coast* from Fig. 19.12 is

$$\text{sim}_{\text{Lin}}(\text{hill}, \text{coast}) = \frac{2 \times \log P(\text{geological-formation})}{\log P(\text{hill}) + \log P(\text{coast})} = 0.59 \tag{19.30}$$

Jiang-Conrath distance

A similar formula, **Jiang-Conrath distance** (Jiang and Conrath, 1997), although derived in a completely different way from Lin and expressed as a distance rather than similarity function, has been shown to work as well as or better than all the other thesaurus-based methods:

$$\text{dist}_{\text{JC}}(c_1, c_2) = 2 \times \log P(\text{LCS}(c_1, c_2)) - (\log P(c_1) + \log P(c_2)) \tag{19.31}$$

We can transform dist_{JC} into a similarity by taking the reciprocal.

Finally, we describe a **dictionary-based** method, an extension of the Lesk algorithm for word sense disambiguation described in Section 19.6.1. We call this a dictionary rather than a thesaurus method because it makes use of glosses, which are, in general, a property of dictionaries rather than thesauruses (although WordNet does have glosses). Like the Lesk algorithm, the intuition of this **extended gloss overlap**, or **extended Lesk** measure (Banerjee and Pedersen, 2003) is that two concepts/senses in a thesaurus are similar if their glosses contain overlapping words. We'll begin by sketching an overlap function for two glosses. Consider these two concepts, with their glosses:

Extended gloss overlap

extended Lesk

- *drawing paper:* paper that is specially prepared for use in drafting
- *decal:* the art of transferring designs from specially prepared paper to a wood or glass or metal surface.

For each n-word phrase that occurs in both glosses, Extended Lesk adds in a score of n^2 (the relation is non-linear because of the Zipfian relationship between lengths of phrases and their corpus frequencies; longer overlaps are rare, so they should be weighted more heavily). Here, the overlapping phrases are *paper* and *specially prepared*, for a total similarity score of $1^2 + 2^2 = 5$.

Given such an overlap function, when comparing two concepts (synsets), Extended Lesk not only looks for overlap between their glosses but also between the glosses of the senses that are hypernyms, hyponyms, meronyms, and other relations of the two concepts. For example, if we just considered hyponyms and defined gloss(hypo(A)) as the concatenation of all the glosses of all the hyponym senses of A, the total relatedness between two concepts A and B might be

$$\begin{aligned}\text{similarity(A,B)} &= \text{overlap(gloss(A), gloss(B))} \\ &\quad +\text{overlap(gloss(hypo(A)), gloss(hypo(B)))} \\ &\quad +\text{overlap(gloss(A), gloss(hypo(B)))} \\ &\quad +\text{overlap(gloss(hypo(A)),gloss(B))}\end{aligned}$$

Let RELS be the set of possible WordNet relations whose glosses we compare; assuming a basic overlap measure as sketched above, we can then define the **Extended Lesk** overlap measure as

$$\text{sim}_{\text{eLesk}}(c_1, c_2) = \sum_{r,q \in \text{RELS}} \text{overlap}(\text{gloss}(r(c_1)), \text{gloss}(q(c_2))) \tag{19.32}$$

$$\begin{aligned}
\text{sim}_{\text{path}}(c_1,c_2) &= \frac{1}{\text{pathlen}(c_1,c_2)} \\
\text{sim}_{\text{Resnik}}(c_1,c_2) &= -\log P(\text{LCS}(c_1,c_2)) \\
\text{sim}_{\text{Lin}}(c_1,c_2) &= \frac{2\times\log P(LCS(c_1,c_2))}{\log P(c_1)+\log P(c_2)} \\
\text{sim}_{\text{JC}}(c_1,c_2) &= \frac{1}{2\times\log P(\text{LCS}(c_1,c_2)) - (\log P(c_1)+\log P(c_2))} \\
\text{sim}_{\text{eLesk}}(c_1,c_2) &= \sum_{r,q\in\text{RELS}} \text{overlap}(\text{gloss}(r(c_1)),\text{gloss}(q(c_2)))
\end{aligned}$$

Figure 19.13 Five thesaurus-based (and dictionary-based) similarity measures.

Figure 19.13 summarizes the five similarity measures we have described in this section.

Evaluating Thesaurus-Based Similarity

Which of these similarity measures is best? Word similarity measures have been evaluated in two ways. The most common intrinsic evaluation metric computes the correlation coefficient between an algorithm's word similarity scores and word similarity ratings assigned by humans. There are a variety of such human-labeled datasets: the RG-65 dataset of human similarity ratings on 65 word pairs (Rubenstein and Goodenough, 1965), the MC-30 dataset of 30 word pairs (Miller and Charles, 1991). The WordSim-353 (Finkelstein et al., 2002) is a commonly used set of of ratings from 0 to 10 for 353 noun pairs; for example (*plane*, *car*) had an average score of 5.77. SimLex-999 (?) is a more difficult dataset that quantifies similarity (*cup, mug*) rather than relatedness (*cup, coffee*), and including both concrete and abstract adjective, noun and verb pairs. Another common intrinic similarity measure is the TOEFL dataset, a set of 80 questions, each consisting of a target word with 4 additional word choices; the task is to choose which is the correct synonym, as in the example: *Levied is closest in meaning to: imposed, believed, requested, correlated* (Landauer and Dumais, 1997). All of these datasets present words without context.

Slightly more realistic are intrinsic similarity tasks that include context. The Stanford Contextual Word Similarity (SCWS) dataset (Huang et al., 2012) offers a richer evaluation scenario, giving human judgments on 2,003 pairs of words in their sentential context, including nouns, verbs, and adjectives. This dataset enables the evaluation of word similarity algorithms that can make use of context words. The *semantic textual similarity* task (Agirre et al. 2012, Agirre et al. 2015) evaluates the performance of sentence-level similarity algorithms, consisting of a set of pairs of sentences, each pair with human-labeled similarity scores.

Alternatively, the similarity measure can be embedded in some end-application, such as question answering (Surdeanu et al., 2011), spell-checking (Jones and Martin 1997, Budanitsky and Hirst 2006, Hirst and Budanitsky 2005), web search result clustering (Di Marco and Navigli, 2013), or text simplification (Biran et al., 2011), and different measures can be evaluated by how much they improve the end application.

We'll return to evaluation metrics in the next chapter when we consider distributional semantics and similarity.

19.10 Summary

This chapter has covered a wide range of issues concerning the meanings associated with lexical items. The following are among the highlights:

- **Lexical semantics** is the study of the meaning of words and the systematic meaning-related connections between words.
- A **word sense** is the locus of word meaning; definitions and meaning relations are defined at the level of the word sense rather than wordforms.
- **Homonymy** is the relation between unrelated senses that share a form, and **polysemy** is the relation between related senses that share a form.
- **Synonymy** holds between different words with the same meaning.
- **Hyponymy** and **hypernymy** relations hold between words that are in a class-inclusion relationship.
- **WordNet** is a large database of lexical relations for English.
- **Word-sense disambiguation** (**WSD**) is the task of determining the correct sense of a word in context. Supervised approaches make use of sentences in which individual words (**lexical sample task**) or all words (**all-words task**) are hand-labeled with senses from a resource like WordNet. Classifiers for supervised WSD are generally trained on **collocational** and **bag-of-words** features that describe the surrounding words.
- An important baseline for WSD is the **most frequent sense**, equivalent, in WordNet, to **take the first sense**.
- The **Lesk algorithm** chooses the sense whose dictionary definition shares the most words with the target word's neighborhood.
- Graph-based algorithms view the thesaurus as a graph and choose the sense that is most central in some way.
- **Word similarity** can be computed by measuring the **link distance** in a thesaurus or by various measures of the **information content** of the two nodes.

Bibliographical and Historical Notes

Word sense disambiguation traces its roots to some of the earliest applications of digital computers. We saw above Warren Weaver's (1955) suggestion to disambiguate a word by looking at a small window around it, in the context of machine translation. Other notions first proposed in this early period include the use of a thesaurus for disambiguation (Masterman, 1957), supervised training of Bayesian models for disambiguation (Madhu and Lytel, 1965), and the use of clustering in word sense analysis (Sparck Jones, 1986).

An enormous amount of work on disambiguation was conducted within the context of early AI-oriented natural language processing systems. Quillian (1968) and Quillian (1969) proposed a graph-based approach to language understanding, in which the dictionary definition of words was represented by a network of word nodes connected by syntactic and semantic relations. He then proposed to do sense disambiguation by finding the shortest path between senses in the conceptual graph. Simmons (1973) is another influential early semantic network approach. Wilks proposed

one of the earliest non-discrete models with his *Preference Semantics* (Wilks 1975c, Wilks 1975b, Wilks 1975a), and Small and Rieger (1982) and Riesbeck (1975) proposed understanding systems based on modeling rich procedural information for each word. Hirst's ABSITY system (Hirst and Charniak 1982, Hirst 1987, Hirst 1988), which used a technique called marker passing based on semantic networks, represents the most advanced system of this type. As with these largely symbolic approaches, early neural network (often called 'connectionist') approaches to word sense disambiguation relied on small lexicons with hand-coded representations (Cottrell 1985, Kawamoto 1988).

Considerable work on sense disambiguation has been conducted in the areas of cognitive science and psycholinguistics. Appropriately enough, this work is generally described by a different name: lexical ambiguity resolution. Small et al. (1988) present a variety of papers from this perspective.

The earliest implementation of a robust empirical approach to sense disambiguation is due to Kelly and Stone (1975), who directed a team that hand-crafted a set of disambiguation rules for 1790 ambiguous English words. Lesk (1986) was the first to use a machine-readable dictionary for word sense disambiguation. The problem of dictionary senses being too fine-grained or lacking an appropriate organization has been addressed with models of clustering word senses (Dolan 1994, Chen and Chang 1998, Mihalcea and Moldovan 2001, Agirre and de Lacalle 2003, Chklovski and Mihalcea 2003, Palmer et al. 2004, Navigli 2006, Snow et al. 2007). Clustered senses are often called **coarse senses**. Corpora with clustered word senses for training clustering algorithms include Palmer et al. (2006) and **OntoNotes** (Hovy et al., 2006).

coarse senses

OntoNotes

Modern interest in supervised machine learning approaches to disambiguation began with Black (1988), who applied decision tree learning to the task. The need for large amounts of annotated text in these methods led to investigations into the use of bootstrapping methods (Hearst 1991, Yarowsky 1995).

Diab and Resnik (2002) give a semi-supervised algorithm for sense disambiguation based on aligned parallel corpora in two languages. For example, the fact that the French word *catastrophe* might be translated as English *disaster* in one instance and *tragedy* in another instance can be used to disambiguate the senses of the two English words (i.e., to choose senses of *disaster* and *tragedy* that are similar). Abney (2002) and Abney (2004) explore the mathematical foundations of the Yarowsky algorithm and its relation to co-training. The most-frequent-sense heuristic is an extremely powerful one but requires large amounts of supervised training data.

The earliest use of clustering in the study of word senses was by Sparck Jones (1986). Zernik (1991) applied a standard information retrieval clustering algorithm to the problem and evaluated it according to improvements in retrieval performance and Pedersen and Bruce (1997), Schütze (1997b), and Schütze (1998) applied distributional methods. Recent work on word sense induction has applied Latent Dirichlet Allocation (LDA) (Boyd-Graber et al. 2007, Brody and Lapata 2009, Lau et al. 2012). and large co-occurrence graphs (Di Marco and Navigli, 2013).

Cruse (2004) is a useful introductory linguistic text on lexical semantics. A collection of work concerning WordNet can be found in Fellbaum (1998). Many efforts have been made to use existing dictionaries as lexical resources. One of the earliest was Amsler's (1981) use of the Merriam Webster dictionary. The machine-readable version of Longman's *Dictionary of Contemporary English* has also been used (Boguraev and Briscoe, 1989).

Navigli (2009) is a comprehensive survey article on WSD, Agirre and Edmonds

(2006) edited a volume that summarizes the state of the art, and Ide and Véronis (1998) review the earlier history of word sense disambiguation up to 1998. Resnik (2006) describes potential applications of WSD. One recent application has been to improve machine translation (Chan et al. 2007, Carpuat and Wu 2007).

See Pustejovsky (1995), Pustejovsky and Boguraev (1996), Martin (1986), and Copestake and Briscoe (1995), inter alia, for computational approaches to the representation of polysemy. Pustejovsky's theory of the **generative lexicon**, and in particular his theory of the **qualia structure** of words, is another way of accounting for the dynamic systematic polysemy of words in context.

generative lexicon
qualia structure

Another important recent direction is the addition of sentiment and connotation to knowledge bases (Wiebe et al. 2005, Qiu et al. 2009, Velikovich et al. 2010) including SentiWordNet (Baccianella et al., 2010) and ConnotationWordNet (Kang et al., 2014).

Exercises

19.1 Collect a small corpus of example sentences of varying lengths from any newspaper or magazine. Using WordNet or any standard dictionary, determine how many senses there are for each of the open-class words in each sentence. How many distinct combinations of senses are there for each sentence? How does this number seem to vary with sentence length?

19.2 Using WordNet or a standard reference dictionary, tag each open-class word in your corpus with its correct tag. Was choosing the correct sense always a straightforward task? Report on any difficulties you encountered.

19.3 Using your favorite dictionary, simulate the original Lesk word overlap disambiguation algorithm described on page 338 on the phrase *Time flies like an arrow*. Assume that the words are to be disambiguated one at a time, from left to right, and that the results from earlier decisions are used later in the process.

19.4 Build an implementation of your solution to the previous exercise. Using WordNet, implement the original Lesk word overlap disambiguation algorithm described on page 338 on the phrase *Time flies like an arrow*.

CHAPTER

20 Lexicons for Sentiment and Affect Extraction

"[W]e write, not with the fingers, but with the whole person. The nerve which controls the pen winds itself about every fibre of our being, threads the heart, pierces the liver."

Virginia Woolf, Orlando

"She runs the gamut of emotions from A to B."

Dorothy Parker, reviewing Hepburn's performance in *Little Women*

affective

subjectivity

In this chapter we turn to tools for interpreting **affective** meaning, extending our study of sentiment analysis in Chapter 4. We use the word 'affective', following the tradition in *affective computing* (Picard, 1995) to mean emotion, sentiment, personality, mood, and attitudes. Affective meaning is closely related to **subjectivity**, the study of a speaker or writer's evaluations, opinions, emotions, and speculations (Wiebe et al., 1999).

How should affective meaning be defined? One influential typology of affective states comes from Scherer (2000), who defines each class of affective states by factors like its cognition realization and time course:

Emotion: Relatively brief episode of response to the evaluation of an external or internal event as being of major significance.
(*angry, sad, joyful, fearful, ashamed, proud, elated, desperate*)

Mood: Diffuse affect state, most pronounced as change in subjective feeling, of low intensity but relatively long duration, often without apparent cause.
(*cheerful, gloomy, irritable, listless, depressed, buoyant*)

Interpersonal stance: Affective stance taken toward another person in a specific interaction, colouring the interpersonal exchange in that situation.
(*distant, cold, warm, supportive, contemptuous, friendly*)

Attitude: Relatively enduring, affectively colored beliefs, preferences, and predispositions towards objects or persons.
(*liking, loving, hating, valuing, desiring*)

Personality traits: Emotionally laden, stable personality dispositions and behavior tendencies, typical for a person.
(*nervous, anxious, reckless, morose, hostile, jealous*)

Figure 20.1 The Scherer typology of affective states, with descriptions from Scherer (2000).

We can design extractors for each of these kinds of affective states. Chapter 4 already introduced *sentiment analysis*, the task of extracting the positive or negative orientation that a writer expresses toward some object. This corresponds in Scherer's typology to the extraction of **attitudes**: figuring out what people like or dislike, whether from consumer reviews of books or movies, newspaper editorials, or public sentiment from blogs or tweets.

Detecting **emotion** and **moods** is useful for detecting whether a student is confused, engaged, or certain when interacting with a tutorial system, whether a caller to a help line is frustrated, whether someone's blog posts or tweets indicated depression. Detecting emotions like fear in novels, for example, could help us trace what groups or situations are feared and how that changes over time.

Detecting different **interpersonal stances** can be useful when extracting information from human-human conversations. The goal here is to detect stances like friendliness or awkwardness in interviews or friendly conversations, or even to detect flirtation in dating. For the task of automatically summarizing meetings, we'd like to be able to automatically understand the social relations between people, who is friendly or antagonistic to whom. A related task is finding parts of a conversation where people are especially excited or engaged, conversational **hot spots** that can help a summarizer focus on the correct region.

Detecting the **personality** of a user—such as whether the user is an **extrovert** or the extent to which they are **open to experience**— can help improve conversational agents, which seem to work better if they match users' personality expectations (Mairesse and Walker, 2008).

Affect is important for generation as well as recognition; synthesizing affect is important for conversational agents in various domains, including literacy tutors such as children's storybooks, or computer games.

In Chapter 4 we introduced the use of Naive Bayes classification to classify a document's sentiment, an approach that has been successfully applied to many of these tasks. In that approach, all the words in the training set are used as features for classifying sentiment.

In this chapter we focus on an alternative model, in which instead of using every word as a feature, we focus only on certain words, ones that carry particularly strong cues to sentiment or affect. We call these lists of words **sentiment or affective lexicons**. In the next sections we introduce lexicons for sentiment, semi-supervised algorithms for inducing them, and simple algorithms for using lexicons to perform sentiment analysis.

We then turn to the extraction of other kinds of affective meaning, beginning with emotion, and the use of online tools for crowdsourcing emotion lexicons, and then proceeding to other kinds of affective meaning like interpersonal stance and personality.

20.1 Available Sentiment Lexicons

The most basic lexicons label words along one dimension of semantic variability, called "sentiment", "valence", or "semantic orientation".

In the simplest lexicons this dimension is represented in a binary fashion, with a wordlist for positive words and a wordlist for negative words. The oldest is the

General Inquirer

General Inquirer (Stone et al., 1966), which drew on early work in the cognition psychology of word meaning (Osgood et al., 1957) and on work in content analysis.

The General Inquirer is a freely available web resource with lexicons of 1915 positive words and 2291 negative words (and also includes other lexicons we'll discuss in the next section).

The MPQA Subjectivity lexicon (Wilson et al., 2005) has 2718 positive and 4912 negative words drawn from a combination of sources, including the General Inquirer lists, the output of the Hatzivassiloglou and McKeown (1997) system described below, and a bootstrapped list of subjective words and phrases (Riloff and Wiebe, 2003) that was then hand-labeled for sentiment. Each phrase in the lexicon is also labeled for reliability (strongly subjective or weakly subjective). The polarity lexicon of (Hu and Liu, 2004b) gives 2006 positive and 4783 negative words, drawn from product reviews, labeled using a bootstrapping method from WordNet described in the next section.

Positive	admire, amazing, assure, celebration, charm, eager, enthusiastic, excellent, fancy, fantastic, frolic, graceful, happy, joy, luck, majesty, mercy, nice, patience, perfect, proud, rejoice, relief, respect, satisfactorily, sensational, super, terrific, thank, vivid, wise, wonderful, zest
Negative	abominable, anger, anxious, bad, catastrophe, cheap, complaint, condescending, deceit, defective, disappointment, embarrass, fake, fear, filthy, fool, guilt, hate, idiot, inflict, lazy, miserable, mourn, nervous, objection, pest, plot, reject, scream, silly, terrible, unfriendly, vile, wicked

Figure 20.2 Some samples of words with consistent sentiment across three sentiment lexicons: the General Inquirer (Stone et al., 1966), the MPQA Subjectivity lexicon (Wilson et al., 2005), and the polarity lexicon of Hu and Liu (2004b).

20.2 Semi-supervised induction of sentiment lexicons

Some affective lexicons are built by having humans assign ratings to words; this was the technique for building the General Inquirer starting in the 1960s (Stone et al., 1966), and for modern lexicons based on crowd-sourcing to be described in Section 20.5.1. But one of the most powerful ways to learn lexicons is to use semi-supervised learning.

In this section we introduce three methods for semi-supervised learning that are important in sentiment lexicon extraction. The three methods all share the same intuitive algorithm which is sketched in Fig. 20.3.

```
function BUILDSENTIMENTLEXICON(posseeds,negseeds) returns poslex,neglex

    poslex ← posseeds
    neglex ← negseeds
    Until done
        poslex ← poslex + FINDSIMILARWORDS(poslex)
        neglex ← neglex + FINDSIMILARWORDS(neglex)
    poslex,neglex ← POSTPROCESS(poslex,neglex)
```

Figure 20.3 Schematic for semi-supervised sentiment lexicon induction. Different algorithms differ in the how words of similar polarity are found, in the stopping criterion, and in the post-processing.

As we will see, the methods differ in the intuitions they use for finding words with similar polarity, and in steps they take to use machine learning to improve the quality of the lexicons.

20.2.1 Using seed words and adjective coordination

The Hatzivassiloglou and McKeown (1997) algorithm for labeling the polarity of adjectives is the same semi-supervised architecture described above. Their algorithm has four steps.

Step 1: Create seed lexicon: Hand-label a seed set of 1336 adjectives (all words that occurred more than 20 times in the 21 million word WSJ corpus). They labeled 657 positive adjectives (e.g., *adequate, central, clever, famous, intelligent, remarkable, reputed, sensitive, slender, thriving*) and 679 negative adjectives (e.g., *contagious, drunken, ignorant, lanky, listless, primitive, strident, troublesome, unresolved, unsuspecting*).

Step 2: Find cues to candidate similar words: Choose words that are similar or different to the seed words, using the intuition that adjectives conjoined by the words *and* tend to have the same polarity. Thus we might expect to see instances of positive adjectives coordinated with positive, or negative with negative:

fair and legitimate, corrupt and brutal

but less likely to see positive adjectives coordinated with negative:

*fair and brutal, *corrupt and legitimate

By contrast, adjectives conjoined by *but* are likely to be of opposite polarity:

fair but brutal

The idea that simple patterns like coordination via *and* and *but* are good tools for finding lexical relations like same-polarity and opposite-polarity is an application of the pattern-based approach to relation extraction described in Chapter 17.

Another cue to opposite polarity comes from morphological negation (*un-*, *im-*, *-less*). Adjectives with the same root but differing in a morphological negative (*adequate/inadequate*, *thoughtful/thoughtless*) tend to be of opposite polarity.

Step 3: Build a polarity graph

These cues are integrated by building a graph with nodes for words and links representing how likely the two words are to have the same polarity, as shown in Fig. 20.4.

A simple way to build a graph would predict an opposite-polarity link if the two adjectives are connected by at least one *but*, and a same-polarity link otherwise (for any two adjectives connected by at least one conjunction). The more sophisticated method used by Hatzivassiloglou and McKeown (1997) is to build a supervised classifier that predicts whether two words are of the same or different polarity, by using these 3 features (occurrence with *and*, occurrence with *but*, and morphological negations).

The classifier is trained on a subset of the hand-labeled seed words, and returns a probability that each pair of words is of the same or opposite polarity. This ‘polarity similarity’ of each word pair can be viewed as the strength of the positive or negative links between them in a graph.

Step 4: Clustering the graph Finally, any of various graph clustering algorithms can be used to divide the graph into two subsets with the same polarity; a graphical intuition is shown in Fig. 20.5.

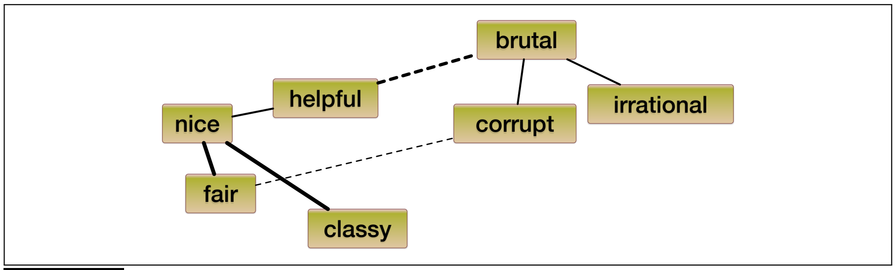


Figure 20.4 A graph of polarity similarity between all pairs of words; words are notes and links represent polarity association between words. Continuous lines are same-polarity and dotted lines are opposite-polarity; the width of lines represents the strength of the polarity.

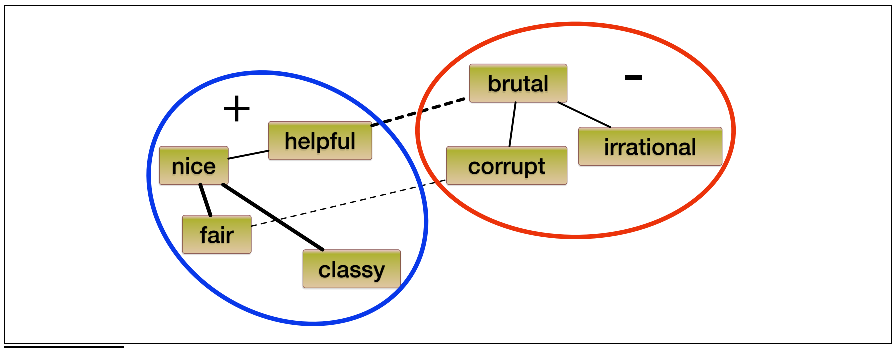


Figure 20.5 The graph from Fig. 20.4 clustered into two groups, using the polarity similarity between two words (visually represented as the edge line strength and continuity) as a distance metric for clustering.

Some sample output from the Hatzivassiloglou and McKeown (1997) algorithm is shown below, showing system errors in red.

> **Positive:** bold decisive disturbing generous good honest important large mature patient peaceful positive proud sound stimulating straightforward strange talented vigorous witty
>
> **Negative:** ambiguous cautious cynical evasive harmful hypocritical inefficient insecure irrational irresponsible minor outspoken pleasant reckless risky selfish tedious unsupported vulnerable wasteful

20.2.2 Pointwise mutual information

Where the first method for finding words with similar polarity relied on patterns of conjunction, we turn now to a second method that uses neighborhood co-occurrence as proxy for polarity similarity. This algorithm assumes that words with similar polarity tend to occur nearby each other, using the pointwise mutual information (PMI) algorithm defined in Chapter 6.

The method of Turney (2002) uses this method to assign polarity to both words and two-word phrases.

In a prior step, two-word phrases are extracted based on simple part-of-speech regular expressions. The expressions select nouns with preceding adjectives, verbs with preceding adverbs, and adjectival heads (adjectives with no following noun) preceded by adverbs, adjectives or nouns:

Word 1 POS	Word 2 POS
JJ	NN|NNS
RB|RBR|RBS	VB|VBD|VBN|VBG
RB|RBR|RBS|JJ|NN|NNS	JJ (only if following word is not NN|NNS)

To measure the polarity of each extracted phrase, we start by choosing positive and negative seed words. For example we might choose a single positive seed word *excellent* and a single negative seed word *poor*. We then make use of the intuition that positive phrases will in general tend to co-occur more with *excellent*. Negative phrases co-occur more with *poor*.

pointwise mutual information

The PMI measure can be used to measure this co-occurrence. Recall from Chapter 6 that the **pointwise mutual information** (Fano, 1961) is a measure of how often two events x and y occur, compared with what we would expect if they were independent:

$$PMI(x,y) = \log_2 \frac{P(x,y)}{P(x)P(y)} \tag{20.1}$$

This intuition can be applied to measure the co-occurrence of two words by defining the pointwise mutual information association between a seed word s and another word w as:

$$\text{PMI}(w,s) = \log_2 \frac{P(w,s)}{P(w)P(s)} \tag{20.2}$$

Turney (2002) estimated the probabilities needed by Eq. 20.2 using a search engine with a `NEAR` operator, specifying that a word has to be *near* another word. The probabilities are then estimated as follows:

$$\begin{aligned} P(w) &= \frac{\text{hits}(w)}{N} && (20.3) \\ P(w1,w2) &= \frac{\text{hits}(w1 \text{ NEAR } w2)}{kN} && (20.4) \end{aligned}$$

That is, we estimate the probability of a word as the count returned from the search engine, normalized by the total number of words in the entire web corpus N. (It doesn't matter that we don't know what N is, since it turns out it will cancel out nicely). The bigram probability is the number of bigram hits normalized by kN—although there are N unigrams and also approximately N bigrams in a corpus of length N, there are kN "NEAR" bigrams in which the two words are separated by a distance of up to k.

The PMI between two words w and s is then:

$$\text{PMI}(w,s) = \log_2 \frac{\frac{1}{kN}\text{hits}(w \text{ NEAR } s)}{\frac{1}{N}\text{hits}(w)\frac{1}{N}\text{hits}(s)} \tag{20.5}$$

The insight of Turney (2002) is then to define the polarity of a word by how much it occurs with the positive seeds and doesn't occur with the negative seeds:

$$
\begin{aligned}
\text{Polarity}(w) &= \text{PMI}(w, \text{“excellent”}) - \text{PMI}(w, \text{“poor”}) \\
&= \log_2 \frac{\frac{1}{kN}\text{hits}(w \text{ NEAR “excellent”})}{\frac{1}{N}\text{hits}(w)\frac{1}{N}\text{hits}(\text{“excellent”})} - \log_2 \frac{\frac{1}{kN}\text{hits}(w \text{ NEAR “poor”})}{\frac{1}{N}\text{hits}(w)\frac{1}{N}\text{hits}(\text{“poor”})} \\
&= \log_2 \left(\frac{\text{hits}(w \text{ NEAR “excellent”})}{\text{hits}(w)\,\text{hits}(\text{“excellent”})} \frac{\text{hits}(w)\,\text{hits}(\text{“poor”})}{\text{hits}(w \text{ NEAR “poor”})} \right) \\
&= \log_2 \left(\frac{\text{hits}(w \text{ NEAR “excellent”})\ \text{hits}(\text{“poor”})}{\text{hits}(\text{“excellent”})\ \text{hits}(w \text{ NEAR “poor”})} \right)
\end{aligned}
\tag{20.6}
$$

The table below from Turney (2002) shows sample examples of phrases learned by the PMI method (from reviews of banking services), showing those with both positive and negative polarity:

Extracted Phrase	Polarity
online experience	2.3
very handy	1.4
low fees	0.3
inconveniently located	-1.5
other problems	-2.8
unethical practices	-8.5

20.2.3 Using WordNet synonyms and antonyms

A third method for finding words that have a similar polarity to seed words is to make use of word synonymy and antonymy. The intuition is that a word's synonyms probably share its polarity while a word's antonyms probably have the opposite polarity.

Since WordNet has these relations, it is often used (Kim and Hovy 2004, Hu and Liu 2004b). After a seed lexicon is built, each lexicon is updated as follows, possibly iterated.

Lex^+ : Add synonyms of positive words (*well*) and antonyms (like *fine*) of negative words

Lex^- : Add synonyms of negative words (*awful*) and antonyms (like *evil*) of positive words

SentiWordNet

An extension of this algorithm has been applied to assign polarity to WordNet senses, called **SentiWordNet** (Baccianella et al., 2010). Fig. 20.6 shows some examples.

In this algorithm, polarity is assigned to entire synsets rather than words. A positive lexicon is built from all the synsets associated with 7 positive words, and a negative lexicon from synsets associated with 7 negative words. Both are expanded by drawing in synsets related by WordNet relations like antonymy or see-also. A classifier is then trained from this data to take a WordNet gloss and decide if the sense being defined is positive, negative or neutral. A further step (involving a random-walk algorithm) assigns a score to each WordNet synset for its degree of positivity, negativity, and neutrality.

In summary, we've seen three distinct ways to use semisupervised learning to induce a sentiment lexicon. All begin with a seed set of positive and negative words, as small as 2 words (Turney, 2002) or as large as a thousand (Hatzivassiloglou and

Synset	Pos	Neg	Obj
good#6 ‘agreeable or pleasing’	1	0	0
respectable#2 honorable#4 good#4 estimable#2 ‘deserving of esteem’	0.75	0	0.25
estimable#3 computable#1 ‘may be computed or estimated’	0	0	1
sting#1 burn#4 bite#2 ‘cause a sharp or stinging pain’	0	0.875	.125
acute#6 ‘of critical importance and consequence’	0.625	0.125	.250
acute#4 ‘of an angle; less than 90 degrees’	0	0	1
acute#1 ‘having or experiencing a rapid onset and short but severe course’	0	0.5	0.5

Figure 20.6 Examples from SentiWordNet 3.0 (Baccianella et al., 2010). Note the differences between senses of homonymous words: *estimable#3* is purely objective, while *estimable#2* is positive; *acute* can be positive (*acute#6*), negative (*acute#1*), or neutral (*acute #4*)

McKeown, 1997). More words of similar polarity are then added, using pattern-based methods, PMI-weighted document co-occurrence, or WordNet synonyms and antonyms. Classifiers can also be used to combine various cues to the polarity of new words, by training on the seed training sets, or early iterations.

20.3 Supervised learning of word sentiment

The previous section showed semi-supervised ways to learn sentiment when there is no supervision signal, by expanding a hand-built seed set using cues to polarity similarity. An alternative to semi-supervision is to do supervised learning, making direct use of a powerful source of supervision for word sentiment: *online reviews*.

The web contains an enormous number of online reviews for restaurants, movies, books, or other products, each of which have the text of the review along with an associated review score: a value that may range from 1 star to 5 stars, or scoring 1 to 10. Fig. 20.7 shows samples extracted from restaurant, book, and movie reviews.

We can use this review score as supervision: positive words are more likely to appear in 5-star reviews; negative words in 1-star reviews. And instead of just a binary polarity, this kind of supervision allows us to assign a word a more complex representation of its polarity: its distribution over stars (or other scores).

Thus in a ten-star system we could represent the sentiment of each word as a 10-tuple, each number a score representing the word's association with that polarity level. This association can be a raw count, or a likelihood $P(w|c)$, or some other function of the count, for each class c from 1 to 10.

For example, we could compute the IMDB likelihood of a word like *disappoint(ed/ing)* occuring in a 1 star review by dividing the number of times *disappoint(ed/ing)* occurs in 1-star reviews in the IMDB dataset (8,557) by the total number of words occurring in 1-star reviews (25,395,214), so the IMDB estimate of $P(disappointing|1)$ is .0003.

A slight modification of this weighting, the normalized likelihood, can be used as an illuminating visualization (Potts, 2011)[1]:

[1] Potts shows that the normalized likelihood is an estimate of the posterior $P(c|w)$ if we make the incorrect but simplifying assumption that all categories c have equal probability.

	Movie review excerpts (IMDB)
10	A great movie. This film is just a wonderful experience. It's surreal, zany, witty and slapstick all at the same time. And terrific performances too.
1	This was probably the worst movie I have ever seen. The story went nowhere even though they could have done some interesting stuff with it.
	Restaurant review excerpts (Yelp)
5	The service was impeccable. The food was cooked and seasoned perfectly... The watermelon was perfectly square ... The grilled octopus was ... mouthwatering...
2	...it took a while to get our waters, we got our entree before our starter, and we never received silverware or napkins until we requested them...
	Book review excerpts (GoodReads)
1	I am going to try and stop being deceived by eye-catching titles. I so wanted to like this book and was so disappointed by it.
5	This book is hilarious. I would recommend it to anyone looking for a satirical read with a romantic twist and a narrator that keeps butting in
	Product review excerpts (Amazon)
5	The lid on this blender though is probably what I like the best about it... enables you to pour into something without even taking the lid off! ... the perfect pitcher! ... works fantastic.
1	I hate this blender... It is nearly impossible to get frozen fruit and ice to turn into a smoothie... You have to add a TON of liquid. I also wish it had a spout ...

Figure 20.7 Excerpts from some reviews from various review websites, all on a scale of 1 to 5 stars except IMDB, which is on a scale of 1 to 10 stars.

$$
\begin{aligned}
P(w|c) &= \frac{count(w,c)}{\sum_{w\in C} count(w,c)} \\
PottsScore(w) &= \frac{P(w|c)}{\sum_c P(w|c)}
\end{aligned}
\tag{20.7}
$$

Dividing the IMDB estimate $P(disappointing|1)$ of .0003 by the sum of the likelihood $P(w|c)$ over all categories gives a Potts score of 0.10. The word *disappointing* thus is associated with the vector [.10, .12, .14, .14, .13, .11, .08, .06, .06, .05]. The **Potts diagram** (Potts, 2011) is a visualization of these word scores, representing the prior sentiment of a word as a distribution over the rating categories.

Potts diagram

Fig. 20.8 shows the Potts diagrams for 3 positive and 3 negative scalar adjectives. Note that the curve for strongly positive scalars have the shape of the letter J, while strongly negative scalars look like a reverse J. By contrast, weakly positive and negative scalars have a hump-shape, with the maximum either below the mean (weakly negative words like *disappointing*) or above the mean (weakly positive words like *good*). These shapes offer an illuminating typology of affective word meaning.

Fig. 20.9 shows the Potts diagrams for emphasizing and attenuating adverbs. Again we see generalizations in the characteristic curves associated with words of particular meanings. Note that emphatics tend to have a J-shape (most likely to occur in the most positive reviews) or a U-shape (most likely to occur in the strongly positive and negative). Attenuators all have the hump-shape, emphasizing the middle of the scale and downplaying both extremes.

The diagrams can be used both as a typology of lexical sentiment, and also play a role in modeling sentiment compositionality.

In addition to functions like posterior $P(c|w)$, likelihood $P(w|c)$, or normalized

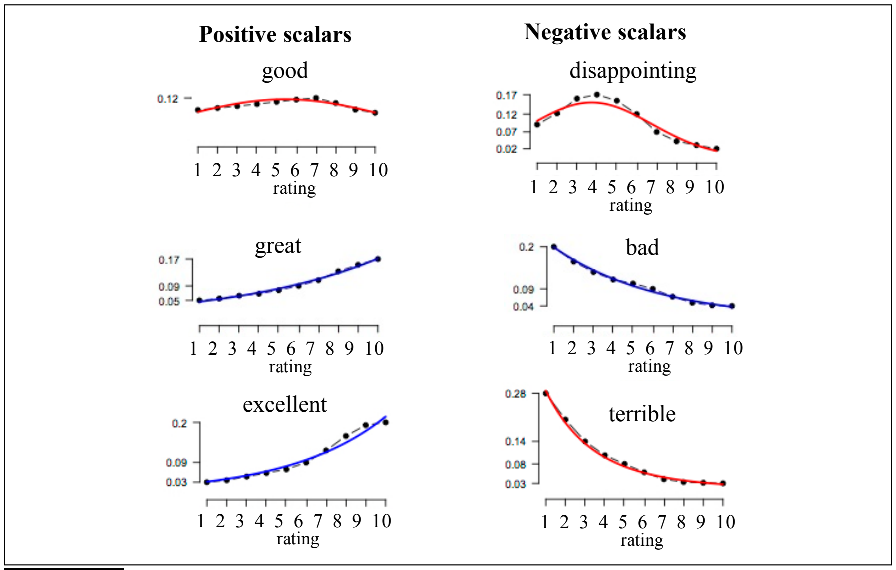


Figure 20.8 Potts diagrams (Potts, 2011) for positive and negative scalar adjectives, showing the J-shape and reverse J-shape for strongly positive and negative adjectives, and the hump-shape for more weakly polarized adjectives.

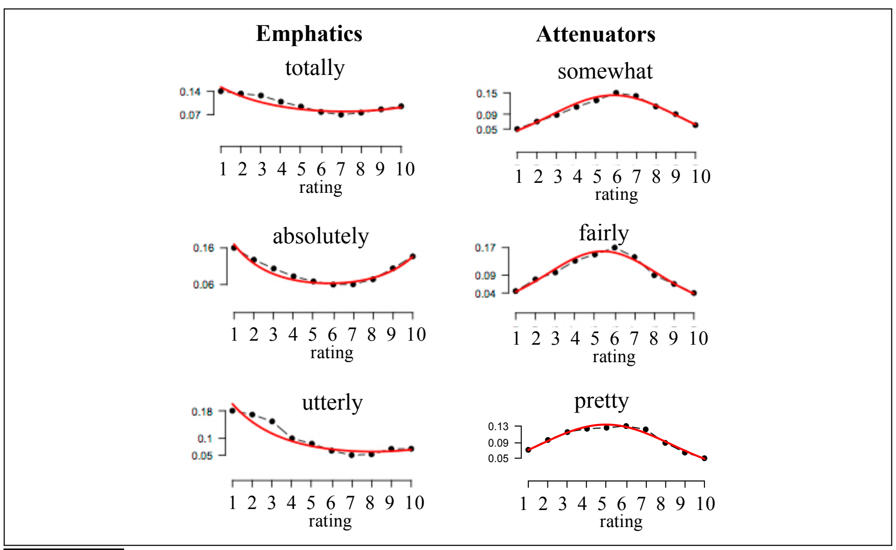


Figure 20.9 Potts diagrams (Potts, 2011) for emphatic and attenuating adverbs.

likelihood (Eq. 20.7) many other functions of the count of a word occuring with a sentiment label have been used. We'll introduce some of these on page 369, including ideas like normalizing the counts per writer in Eq. 20.15.

20.3.1 Log odds ratio informative Dirichlet prior

One thing we often want to do with word polarity is to distinguish between words that are more likely to be used in one category of texts than in another. We may, for example, want to know the words most associated with 1 star reviews versus those associated with 5 star reviews. These differences may not be just related to sentiment. We might want to find words used more often by Democratic than Republican

members of Congress, or words used more often in menus of expensive restaurants than cheap restaurants.

Given two classes of documents, to find words more associated with one category than another, we might choose to just compute the difference in frequencies (is a word w more frequent in class A or class B?). Or instead of the difference in frequencies we might want to compute the ratio of frequencies, or the log odds ratio (the log of the ratio between the odds of the two words). Then we can sort words by whichever of these associations with the category we use, (sorting from words overrepresented in category A to words overrepresented in category B).

The problem with simple log-likelihood or log odds methods is that they don't work well for very rare words or very frequent words; for words that are very frequent, all differences seem large, and for words that are very rare, no differences seem large.

In this section we walk through the details of one solution ot this problem ¡ the "log odds ratio informative Dirichlet prior" method of Monroe et al. (2008) that is a particularly useful method for finding words that are statistically overrepresented in one particular category of texts compared to another. It's based on the idea of using another large corpus to get a prior estimate of what we expect the frequency of each word to be.

log likelihood ratio

Let's start with the goal: assume we want to know whether the word *horrible* occurs more in corpus i or corpus j. We could compute the **log likelihood ratio**, using $f^i(w)$ to mean the frequency of word w in corpus i, and n^i to mean the total number of words in corpus i:

$$\begin{aligned}
\text{llr}(horrible) &= \log \frac{P^i(horrible)}{P^j(horrible)} \\
&= \log P^i(horrible) - \log P^j(horrible) \\
&= \log \frac{\text{f}^i(horrible)}{n^i} - \log \frac{\text{f}^j(horrible)}{n^j}
\end{aligned} \tag{20.8}$$

log odds ratio

Instead, let's compute the **log odds ratio**: does *horrible* have higher odds in i or in j:

$$\begin{aligned}
\text{lor}(horrible) &= \log\left(\frac{P^i(horrible)}{1-P^i(horrible)}\right) - \log\left(\frac{P^j(horrible)}{1-P^j(horrible)}\right) \\
&= \log\left(\frac{\frac{\text{f}^i(horrible)}{n^i}}{1-\frac{\text{f}^i(horrible)}{n^i}}\right) - \log\left(\frac{\frac{\text{f}^j(horrible)}{n^j}}{1-\frac{\text{f}^j(horrible)}{n^j}}\right) \\
&= \log\left(\frac{\text{f}^i(horrible)}{n^i-\text{f}^i(horrible)}\right) - \log\left(\frac{\text{f}^j(horrible)}{n^j-\text{f}^j(horrible)}\right)
\end{aligned} \tag{20.9}$$

The Dirichlet intuition is to use a large background corpus to get a prior estimate of what we expect the frequency of each word w to be. We'll do this very simply by adding the counts from that corpus to the numerator and denominator, so that we're essentially shrinking the counts toward that prior. It's like asking how large are the differences between i and j given what we would expect given their frequencies in a well-estimated large background corpus.

The method estimates the difference between the frequency of word w in two corpora i and j via the prior-modified log odds ratio for w, $\delta_w^{(i-j)}$, which is estimated

as:

$$\delta_w^{(i-j)} = \log\left(\frac{f_w^i + \alpha_w}{n^i + \alpha_0 - (f_w^i + \alpha_w)}\right) - \log\left(\frac{f_w^j + \alpha_w}{n^j + \alpha_0 - (f_w^j + \alpha_w)}\right) \tag{20.10}$$

(where n^i is the size of corpus i, n^j is the size of corpus j, f_w^i is the count of word w in corpus i, f_w^j is the count of word w in corpus j, α_0 is the size of the background corpus, and α_w is the count of word w in the background corpus.)

In addition, Monroe et al. (2008) make use of an estimate for the variance of the log–odds–ratio:

$$\sigma^2\left(\hat{\delta}_w^{(i-j)}\right) \approx \frac{1}{f_w^i + \alpha_w} + \frac{1}{f_w^j + \alpha_w} \tag{20.11}$$

The final statistic for a word is then the z–score of its log–odds–ratio:

$$\frac{\hat{\delta}_w^{(i-j)}}{\sqrt{\sigma^2\left(\hat{\delta}_w^{(i-j)}\right)}} \tag{20.12}$$

The Monroe et al. (2008) method thus modifies the commonly used log odds ratio in two ways: it uses the z-scores of the log odds ratio, which controls for the amount of variance in a words frequency, and it uses counts from a background corpus to provide a prior count for words.

Fig. 20.10 shows the method applied to a dataset of restaurant reviews from Yelp, comparing the words used in 1-star reviews to the words used in 5-star reviews (Jurafsky et al., 2014). The largest difference is in obvious sentiment words, with the 1-star reviews using negative sentiment words like *worse, bad, awful* and the 5-star reviews using positive sentiment words like *great, best, amazing*. But there are other illuminating differences. 1-star reviews use logical negation (*no, not*), while 5-star reviews use emphatics and emphasize universality (*very, highly, every, always*). 1-star reviews use first person plurals (*we, us, our*) while 5 star reviews use the second person. 1-star reviews talk about people (*manager, waiter, customer*) while 5-star reviews talk about dessert and properties of expensive restaurants like courses and atmosphere. See Jurafsky et al. (2014) for more details.

20.4 Using Lexicons for Sentiment Recognition

In Chapter 4 we introduced the naive Bayes algorithm for sentiment analysis. The lexicons we have focused on throughout the chapter so far can be used in a number of ways to improve sentiment detection.

In the simplest case, lexicons can be used when we don't have sufficient training data to build a supervised sentiment analyzer; it can often be expensive to have a human assign sentiment to each document to train the supervised classifier.

In such situations, lexicons can be used in a simple rule-based algorithm for classification. The simplest version is just to use the ratio of positive to negative words: if a document has more positive than negative words (using the lexicon to decide the polarity of each word in the document), it is classified as positive. Often a threshold λ is used, in which a document is classified as positive only if the ratio

Class	Words in 1-star reviews	Class	Words in 5-star reviews
Negative	*worst, rude, terrible, horrible, bad, awful, disgusting, bland, tasteless, gross, mediocre, overpriced, worse, poor*	**Positive**	*great, best, love(d), delicious, amazing, favorite, perfect, excellent, awesome, friendly, fantastic, fresh, wonderful, incredible, sweet, yum(my)*
Negation	*no, not*	**Emphatics/ universals**	*very, highly, perfectly, definitely, absolutely, everything, every, always*
1Pl pro	*we, us, our*	**2 pro**	*you*
3 pro	*she, he, her, him*	**Articles**	*a, the*
Past verb	*was, were, asked, told, said, did, charged, waited, left, took*	**Advice**	*try, recommend*
Sequencers	*after, then*	**Conjunct**	*also, as, well, with, and*
Nouns	*manager, waitress, waiter, customer, customers, attitude, waste, poisoning, money, bill, minutes*	**Nouns**	*atmosphere, dessert, chocolate, wine, course, menu*
Irrealis modals	*would, should*	**Auxiliaries**	*is/'s, can, 've, are*
Comp	*to, that*	**Prep, other**	*in, of, die, city, mouth*

Figure 20.10 The top 50 words associated with one–star and five-star restaurant reviews in a Yelp dataset of 900,000 reviews, using the Monroe et al. (2008) method (Jurafsky et al., 2014).

is greater than λ. If the sentiment lexicon includes positive and negative weights for each word, θ^+_w and θ^-_w, these can be used as well. Here's a simple such sentiment algorithm:

$$
\begin{aligned}
f^+ &= \sum_{w \text{ s.t. } w \in positivelexicon} \theta^+_w count(w) \\
f^- &= \sum_{w \text{ s.t. } w \in negativelexicon} \theta^-_w count(w) \\
sentiment &= \begin{cases} + & \text{if } \frac{f^+}{f^-} > \lambda \\ - & \text{if } \frac{f^-}{f^+} > \lambda \\ 0 & \text{otherwise.} \end{cases}
\end{aligned}
\qquad (20.13)
$$

If supervised training data is available, these counts computed from sentiment lexicons, sometimes weighted or normalized in various ways, can also be used as features in a classifier along with other lexical or non-lexical features. We return to such algorithms in Section 20.7.

20.5 Emotion and other classes

emotion One of the most important affective classes is **emotion**, which Scherer (2000) defines as a "relatively brief episode of response to the evaluation of an external or internal event as being of major significance".

Detecting emotion has the potential to improve a number of language processing tasks. Automatically detecting emotions in reviews or customer responses (anger, dissatisfaction, trust) could help businesses recognize specific problem areas or ones that are going well. Emotion recognition could help dialog systems like tutoring systems detect that a student was unhappy, bored, hesitant, confident, and so on.

Emotion can play a role in medical informatics tasks like detecting depression or suicidal intent. Detecting emotions expressed toward characters in novels might play a role in understanding how different social groups were viewed by society at different times.

basic emotions

There are two widely-held families of theories of emotion. In one family, emotions are viewed as fixed atomic units, limited in number, and from which others are generated, often called **basic emotions** (Tomkins 1962, Plutchik 1962). Perhaps most well-known of this family of theories are the 6 emotions proposed by (Ekman, 1999) as a set of emotions that is likely to be universally present in all cultures: *surprise, happiness, anger, fear, disgust, sadness*. Another atomic theory is the (Plutchik, 1980) wheel of emotion, consisting of 8 basic emotions in four opposing pairs: *joy–sadness*, *anger–fear*, *trust–disgust*, and *anticipation–surprise*, together with the emotions derived from them, shown in Fig. 20.11.

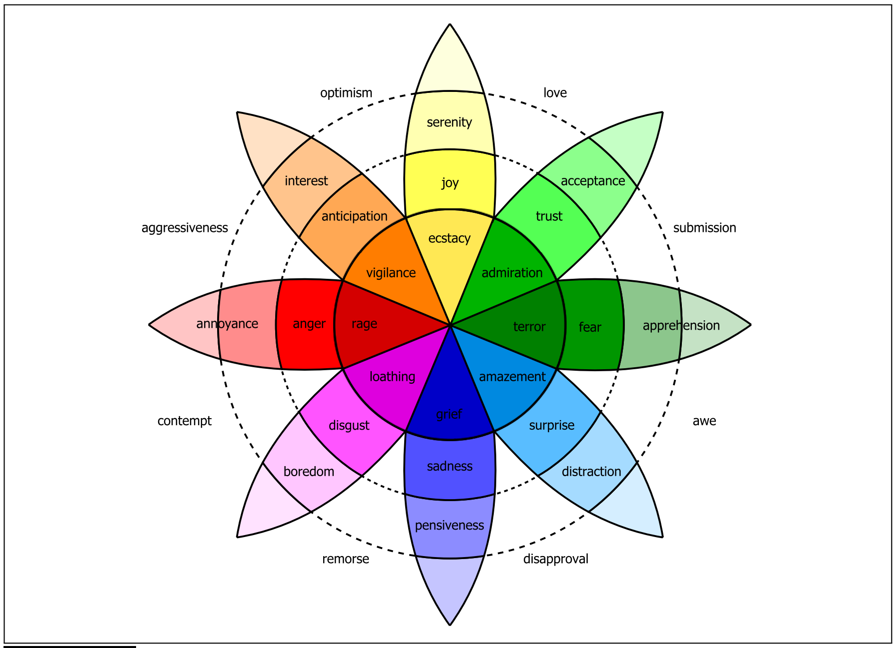


Figure 20.11 Plutchik wheel of emotion.

The second class of emotion theories views emotion as a space in 2 or 3 dimensions (Russell, 1980). Most models include the two dimensions **valence** and **arousal**, and many add a third, **dominance**. These can be defined as:

valence: the pleasantness of the stimulus

arousal: the intensity of emotion provoked by the stimulus

dominance: the degree of control exerted by the stimulus

Practical lexicons have been built for both kinds of theories of emotion.

20.5.1 Lexicons for emotion and other affective states

crowdsourcing

While semi-supervised algorithms are the norm in sentiment and polarity, the most common way to build emotional lexicons is to have humans label the words. This is most commonly done using **crowdsourcing**: breaking the task into small pieces and distributing them to a large number of annotators. Let's take a look at one

crowdsourced emotion lexicon from each of the two common theoretical models of emotion.

EmoLex

The NRC Word-Emotion Association Lexicon, also called **EmoLex** (Mohammad and Turney, 2013), uses the Plutchik (1980) 8 basic emotions defined above. The lexicon includes around 14,000 words chosen partly from the prior lexicons (the General Inquirer and WordNet Affect Lexicons) and partly from the Macquarie Thesaurus, from which the 200 most frequent words were chosen from four parts of speech: nouns, verbs, adverbs, and adjectives (using frequencies from the Google n-gram count).

In order to ensure that the annotators were judging the correct sense of the word, they first answered a multiple-choice synonym question that primed the correct sense of the word (without requiring the annotator to read a potentially confusing sense definition). These were created automatically using the headwords associated with the thesaurus category of the sense in question in the Macquarie dictionary and the headwords of 3 random distractor categories. An example:

```
Which word is closest in meaning (most related) to startle?
• automobile
• shake
• honesty
• entertain
```

For each word (e.g. *startle*), the annotator was asked to rate how associated that word is with each of the 8 emotions (*joy*, *fear*, *anger*, etc.). The associations were rated on a scale of *not*, *weakly*, *moderately*, and *strongly* associated. Outlier ratings were removed, and then each term was assigned the class chosen by the majority of the annotators, with ties broken by choosing the stronger intensity, and then the 4 levels were mapped into a binary label for each word (no and weak mapped to 0, moderate and strong mapped to 1). Values from the lexicon for some sample words:

Word	**anger**	**anticipation**	**disgust**	**fear**	**joy**	**sadness**	**surprise**	**trust**	**positive**	**negative**
reward	0	1	0	0	1	0	1	1	1	0
worry	0	1	0	1	0	1	0	0	0	1
tenderness	0	0	0	0	1	0	0	0	1	0
sweetheart	0	1	0	0	1	1	0	1	1	0
suddenly	0	0	0	0	0	0	1	0	0	0
thirst	0	1	0	0	0	1	1	0	0	0
garbage	0	0	1	0	0	0	0	0	0	1

A second lexicon, also built using crowdsourcing, assigns values on three dimensions (valence/arousal/dominance) to 14,000 words (Warriner et al., 2013).

The annotators marked each word with a value from 1-9 on each of the dimensions, with the scale defined for them as follows:

- valence (the pleasantness of the stimulus)
  9: happy, pleased, satisfied, contented, hopeful
  1: unhappy, annoyed, unsatisfied, melancholic, despaired, or bored
- arousal (the intensity of emotion provoked by the stimulus)
  9: stimulated, excited, frenzied, jittery, wide-awake, or aroused
  1: relaxed, calm, sluggish, dull, sleepy, or unaroused;

- dominance (the degree of control exerted by the stimulus)
  9: in control, influential, important, dominant, autonomous, or controlling
  1: controlled, influenced, cared-for, awed, submissive, or guided

Some examples are shown in Fig. 20.12

Valence		Arousal		Dominance	
vacation	8.53	rampage	7.56	self	7.74
happy	8.47	tornado	7.45	incredible	7.74
whistle	5.7	zucchini	4.18	skillet	5.33
conscious	5.53	dressy	4.15	concur	5.29
torture	1.4	dull	1.67	earthquake	2.14

Figure 20.12 Samples of the values of selected words on the three emotional dimensions from Warriner et al. (2013).

There are various other hand-built lexicons of words related in various ways to the emotions. The General Inquirer includes lexicons like strong vs. weak, active vs. passive, overstated vs. understated, as well as lexicons for categories like pleasure, pain, virtue, vice, motivation, and cognitive orientation.

concrete
abstract

Another useful feature for various tasks is the distinction between **concrete** words like *banana* or *bathrobe* and **abstract** words like *belief* and *although*. The lexicon in (Brysbaert et al., 2014) used crowdsourcing to assign a rating from 1 to 5 of the concreteness of 40,000 words, thus assigning *banana*, *bathrobe*, and *bagel* 5, *belief* 1.19, *although* 1.07, and in between words like *brisk* a 2.5.

LIWC

LIWC, **Linguistic Inquiry and Word Count**, is another set of 73 lexicons containing over 2300 words (Pennebaker et al., 2007), designed to capture aspects of lexical meaning relevant for social psychological tasks. In addition to sentiment-related lexicons like ones for negative emotion (*bad, weird, hate, problem, tough*) and positive emotion (*love, nice, sweet*), LIWC includes lexicons for categories like anger, sadness, cognitive mechanisms, perception, tentative, and inhibition, shown in Fig. 20.13.

Positive Emotion	Negative Emotion	Insight	Inhibition	Family	Negate
appreciat*	anger*	aware*	avoid*	brother*	aren't
comfort*	bore*	believe	careful*	cousin*	cannot
great	cry	decid*	hesitat*	daughter*	didn't
happy	despair*	feel	limit*	family	neither
interest	fail*	figur*	oppos*	father*	never
joy*	fear	know	prevent*	grandf*	no
perfect*	griev*	knew	reluctan*	grandm*	nobod*
please*	hate*	means	safe*	husband	none
safe*	panic*	notice*	stop	mom	nor
terrific	suffers	recogni*	stubborn*	mother	nothing
value	terrify	sense	wait	niece*	nowhere
wow*	violent*	think	wary	wife	without

Figure 20.13 Samples from 5 of the 73 lexical categories in LIWC (Pennebaker et al., 2007). The * means the previous letters are a word prefix and all words with that prefix are included in the category.

20.6 Other tasks: Personality

personality

Many other kinds of affective meaning can be extracted from text and speech. For example detecting a person's **personality** from their language can be useful for dialog systems (users tend to prefer agents that match their personality), and can play a useful role in computational social science questions like understanding how personality is related to other kinds of behavior.

Many theories of human personality are based around a small number of dimensions, such as various versions of the "Big Five" dimensions (Digman, 1990):

Extroversion vs. Introversion: sociable, assertive, playful vs. aloof, reserved, shy

Emotional stability vs. Neuroticism: calm, unemotional vs. insecure, anxious

Agreeableness vs. Disagreeableness: friendly, cooperative vs. antagonistic, fault-finding

Conscientiousness vs. Unconscientiousness: self-disciplined, organized vs. inefficient, careless

Openness to experience: intellectual, insightful vs. shallow, unimaginative

A few corpora of text and speech have been labeled for the personality of their author by having the authors take a standard personality test. The essay corpus of Pennebaker and King (1999) consists of 2,479 essays (1.9 million words) from psychology students who were asked to "write whatever comes into your mind" for 20 minutes. The EAR (Electronically Activated Recorder) corpus of Mehl et al. (2006) was created by having volunteers wear a recorder throughout the day, which randomly recorded short snippets of conversation throughout the day, which were then transcribed. The Facebook corpus of (Schwartz et al., 2013) includes 309 million words of Facebook posts from 75,000 volunteers.

For example, here are samples from Pennebaker and King (1999) from an essay written by someone on the neurotic end of the neurotic/emotionally stable scale,

> One of my friends just barged in, and I jumped in my seat. This is crazy. I should tell him not to do that again. I'm not that fastidious actually. But certain things annoy me. The things that would annoy me would actually annoy any normal human being, so I know I'm not a freak.

and someone on the emotionally stable end of the scale:

> I should excel in this sport because I know how to push my body harder than anyone I know, no matter what the test I always push my body harder than everyone else. I want to be the best no matter what the sport or event. I should also be good at this because I love to ride my bike.

interpersonal stance

Another kind of affective meaning is what Scherer (2000) calls **interpersonal stance**, the 'affective stance taken toward another person in a specific interaction coloring the interpersonal exchange'. Extracting this kind of meaning means automatically labeling participants for whether they are friendly, supportive, distant. For example Ranganath et al. (2013) studied a corpus of speed-dates, in which participants went on a series of 4-minute romantic dates, wearing microphones. Each participant labeled each other for how flirtatious, friendly, awkward, or assertive they were. Ranganath et al. (2013) then used a combination of lexicons and other features to detect these interpersonal stances from text.

20.7 Affect Recognition

Detection of emotion, personality, interactional stance, and the other kinds of affective meaning described by Scherer (2000) can be done by generalizing the algorithms described above for detecting sentiment.

The most common algorithms involve supervised classification: a training set is labeled for the affective meaning to be detected, and a classifier is built using features extracted from the training set. As with sentiment analysis, if the training set is large enough, and the test set is sufficiently similar to the training set, simply using all the words or all the bigrams as features in a powerful classifier like SVM or logistic regression, as described in Fig. 4.2 in Chapter 4, is an excellent algorithm whose performance is hard to beat. Thus we can treat affective meaning classification of a text sample as simple document classification.

Some modifications are nonetheless often necessary for very large datasets. For example, the Schwartz et al. (2013) study of personality, gender, and age using 700 million words of Facebook posts used only a subset of the n-grams of lengths 1-3. Only words and phrases used by at least 1% of the subjects were included as features, and 2-grams and 3-grams were only kept if they had sufficiently high PMI (PMI greater than $2 * length$, where *length* is the number of words):

$$\text{pmi}(phrase) = \log \frac{p(phrase)}{\prod\limits_{w \in phrase} p(w)} \tag{20.14}$$

Various weights can be used for the features, including the raw count in the training set, or some normalized probability or log probability. Schwartz et al. (2013), for example, turn feature counts into phrase likelihoods by normalizing them by each subject's total word use.

$$p(phrase|subject) = \frac{\text{freq}(phrase, subject)}{\sum\limits_{phrase' \in \text{vocab}(subject)} \text{freq}(phrase', subject)} \tag{20.15}$$

If the training data is sparser, or not as similar to the test set, any of the lexicons we've discussed can play a helpful role, either alone or in combination with all the words and n-grams.

Many possible values can be used for lexicon features. The simplest is just an indicator function, in which the value of a feature $f_{\mathscr{L}}$ takes the value 1 if a particular text has any word from the relevant lexicon $\mathscr{L}$. Using the notation of Chapter 4, in which a feature value is defined for a particular output class c and document x.

$$f_{\mathscr{L}}(c,x) \;=\; \begin{cases} 1 & \text{if } \exists w : w \in \mathscr{L} \;\&\; w \in x \;\&\; class = c \\ 0 & \text{otherwise} \end{cases} \tag{20.16}$$

Alternatively the value of a feature $f_{\mathscr{L}}$ for a particular lexicon $\mathscr{L}$ can be the total number of word *tokens* in the document that occur in $\mathscr{L}$:

$$f_{\mathscr{L}} = \sum_{w \in \mathscr{L}} count(w)$$

For lexica in which each word is associated with a score or weight, the count can be multiplied by a weight $\theta_w^{\mathscr{L}}$:

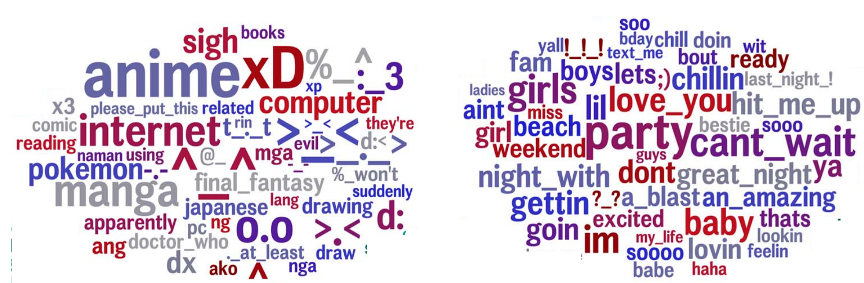
$$f_{\mathscr{L}} = \sum_{w \in \mathscr{L}} \theta_w^{\mathscr{L}} count(w)$$

Counts can alternatively be logged or normalized per writer as in Eq. 20.15.

However they are defined, these lexicon features are then used in a supervised classifier to predict the desired affective category for the text or document. Once a classifier is trained, we can examine which lexicon features are associated with which classes. For a classifier like logistic regression the feature weight gives an indication of how associated the feature is with the class.

Thus, for example, (Mairesse and Walker, 2008) found that for classifying personality, for the dimension *Agreeable*, the LIWC lexicons *Family* and *Home* were positively associated while the LIWC lexicons *anger* and *swear* were negatively associated. By contrast, Extroversion was positively associated with the *Friend*, *Religion* and *Self* lexicons, and Emotional Stability was positively associated with *Sports* and negatively associated with *Negative Emotion*.

In the situation in which we use all the words and phrases in the document as potential features, we can use the resulting weights from the learned regression classifier as the basis of an affective lexicon. Thus, for example, in the Extroversion/Introversion classifier of Schwartz et al. (2013), ordinary least-squares regression is used to predict the value of a personality dimension from all the words and phrases. The resulting regression coefficient for each word or phrase can be used as an association value with the predicted dimension. The word clouds in Fig. 20.14 show an example of words associated with introversion (a) and extroversion (b).



(a) (b)

Figure 20.14 Word clouds from Schwartz et al. (2013), showing words highly associated with introversion (left) or extroversion (right). The size of the word represents the association strength (the regression coefficient), while the color (ranging from cold to hot) represents the relative frequency of the word/phrase (from low to high).

20.8 Summary

- Many kinds of affective states can be distinguished, including *emotions*, *moods*, *attitudes* (which include *sentiment*), *interpersonal stance*, and *personality*.
- Words have **connotational** aspects related to these affective states, and this connotational aspect of word meaning can be represented in lexicons.

- Affective lexicons can be built by hand, using **crowd sourcing** to label the affective content of each word.
- Lexicons can be built **semi-supervised**, bootstrapping from seed words using similarity metrics like the frequency two words are conjoined by *and* or *but*, the two words' pointwise mutual information, or their association via WordNet synonymy or antonymy relations.
- Lexicons can be learned in a **fully supervised** manner, when a convenient training signal can be found in the world, such as ratings assigned by users on a review site.
- Words can be assigned weights in a lexicon by using various functions of word counts in training texts, and ratio metrics like **log odds ratio informative Dirichlet prior**.
- **Emotion** can be represented by fixed atomic units often called **basic emotions**, or as points in space defined by dimensions like **valence** and **arousal**.
- Personality is often represented as a point in 5-dimensional space.
- Affect can be detected, just like sentiment, by using standard supervised **text classification** techniques, using all the words or bigrams in a text as features. Additional features can be drawn from counts of words in lexicons.
- Lexicons can also be used to detect affect in a **rule-based classifier** by picking the simple majority sentiment based on counts of words in each lexicon.

Bibliographical and Historical Notes

The idea of formally representing the subjective meaning of words began with Osgood et al. (1957), the same pioneering study that first proposed the vector space model of meaning described in Chapter 6. Osgood et al. (1957) had participants rate words on various scales, and ran factor analysis on the ratings. The most significant factor they uncovered was the evaluative dimension, which distinguished between pairs like *good/bad*, *valuable/worthless*, *pleasant/unpleasant*. This work influenced the development of early dictionaries of sentiment and affective meaning in the field of **content analysis** (Stone et al., 1966).

subjectivity

Wiebe (1994) began an influential line of work on detecting **subjectivity** in text, beginning with the task of identifying subjective sentences and the subjective characters who are described in the text as holding private states, beliefs or attitudes. Learned sentiment lexicons such as the polarity lexicons of (Hatzivassiloglou and McKeown, 1997) were shown to be a useful feature in subjectivity detection (Hatzivassiloglou and Wiebe 2000, Wiebe 2000).

The term **sentiment** seems to have been introduced in 2001 by Das and Chen (2001), to describe the task of measuring market sentiment by looking at the words in stock trading message boards. In the same paper Das and Chen (2001) also proposed the use of a sentiment lexicon. The list of words in the lexicon was created by hand, but each word was assigned weights according to how much it discriminated a particular class (say buy versus sell) by maximizing across-class variation and minimizing within-class variation. The term *sentiment*, and the use of lexicons, caught on quite quickly (e.g., inter alia, Turney 2002). Pang et al. (2002) first showed the power of using all the words without a sentiment lexicon; see also Wang and Manning (2012).

The semi-supervised methods we describe for extending sentiment dictionaries all drew on the early idea that synonyms and antonyms tend to co-occur in the same sentence. (Miller and Charles 1991, Justeson and Katz 1991). Other semi-supervized methods for learning cues to affective meaning rely on information extraction techniques, like the AutoSlog pattern extractors (Riloff and Wiebe, 2003).

For further information on sentiment analysis, including discussion of lexicons, see the useful surveys of Pang and Lee (2008) and Liu (2015).

CHAPTER

21 Coreference Resolution and Entity Linking

Placeholder

CHAPTER

22 Discourse Coherence

Placeholder

CHAPTER

23 Machine Translation and Seq2Seq Models

Placeholder

CHAPTER

24 Question Answering

The quest for knowledge is deeply human, and so it is not surprising that practically as soon as there were computers, and certainly as soon as there was natural language processing, we were trying to use computers to answer textual questions. By the early 1960s, there were systems implementing the two major modern paradigms of question answering—**IR-based question answering** and **knowledge-based question answering** to answer questions about baseball statistics or scientific facts. Even imaginary computers got into the act. Deep Thought, the computer that Douglas Adams invented in *The Hitchhiker's Guide to the Galaxy*, managed to answer "the Great Question Of Life The Universe and Everything" (the answer was 42, but unfortunately the details of the question were never revealed).

More recently, IBM's Watson question-answering system won the TV game-show *Jeopardy!* in 2011, beating humans at the task of answering questions like

> WILLIAM WILKINSON'S "AN ACCOUNT OF THE PRINCIPALITIES OF WALLACHIA AND MOLDOVIA" INSPIRED THIS AUTHOR'S MOST FAMOUS NOVEL[1]

Although the goal of quiz shows is entertainment, the technology used to answer these questions both draws on and extends the state of the art in practical question answering, as we will see.

Most current question answering systems focus on **factoid questions**. Factoid questions are questions that can be answered with simple facts expressed in short text answers. The following factoid questions, for example, can be answered with a short string expressing a personal name, temporal expression, or location:

(24.1) Who founded Virgin Airlines?

(24.2) What is the average age of the onset of autism?

(24.3) Where is Apple Computer based?

In this chapter we describe the two major modern paradigms to question answering, focusing on their application to factoid questions.

The first paradigm is called **IR-based question answering** or sometimes **text-based question answering**, and relies on the enormous amounts of information available as text on the Web or in specialized collections such as PubMed. Given a user question, information retrieval techniques extract passages directly from these documents, guided by the text of the question.

The method processes the question to determine the likely **answer type** (often a named entity like a person, location, or time), and formulates queries to send to a search engine. The search engine returns ranked documents which are broken up into suitable passages and reranked. Finally candidate answer strings are extracted from the passages and ranked.

---

1 The answer, of course, is Bram Stoker, and the novel was the fantastically Gothic *Dracula*.

In the second paradigm, **knowledge-based question answering**, we instead build a semantic representation of the query. The meaning of a query can be a full predicate calculus statement. So the question *What states border Texas?*—taken from the GeoQuery database of questions on U.S. Geography (Zelle and Mooney, 1996)— might have the representation:

$\lambda x.state(x) \wedge borders(x, texas)$

Alternatively the meaning of a question could be a single relation between a known and an unknown entity. Thus the representation of the question *When was Ada Lovelace born?* could be `birth-year (Ada Lovelace, ?x)`.

Whatever meaning representation we choose, we'll be using it to query databases of facts. These might be complex databases, perhaps of scientific facts or geospatial information, that need powerful logical or SQL queries. Or these might be databases of simple relations, **triple stores** like Freebase or DBpedia introduced in Chapter 17.

triple stores

Large practical systems like the DeepQA system in IBM's Watson generally are hybrid systems, using both text datasets and structured knowledge bases to answer questions. DeepQA extracts a wide variety of meanings from the question (parses, relations, named entities, ontological information), and then finds large numbers of candidate answers in both knowledge bases and in textual sources like Wikipedia or newspapers. Each candidate answer is then scored using a wide variety of knowledge sources, such as geospatial databases, temporal reasoning, taxonomical classification, and various textual sources.

We'll explore all three of these approaches: IR-based, knowledge-based, and the Watson DeepQA system, in the next three sections.

24.1 IR-based Factoid Question Answering

The goal of IR-based question answering is to answer a user's question by finding short text segments on the Web or some other collection of documents. Figure 24.1 shows some sample factoid questions and their answers.

Question	Answer
Where is the Louvre Museum located?	in Paris, France
What's the abbreviation for limited partnership?	L.P.
What are the names of Odin's ravens?	Huginn and Muninn
What currency is used in China?	the yuan
What kind of nuts are used in marzipan?	almonds
What instrument does Max Roach play?	drums
What's the official language of Algeria?	Arabic
How many pounds are there in a stone?	14

Figure 24.1 Some sample factoid questions and their answers.

Figure 24.2 shows the three phases of an IR-based factoid question-answering system: question processing, passage retrieval and ranking, and answer processing.

24.1.1 Question Processing

The goal of the question-processing phase is to extract a number of pieces of information from the question. The **answer type** specifies the kind of entity the answer consists of (person, location, time, etc.). The **query** specifies the keywords that should be used for the IR system to use in searching for documents. Some systems

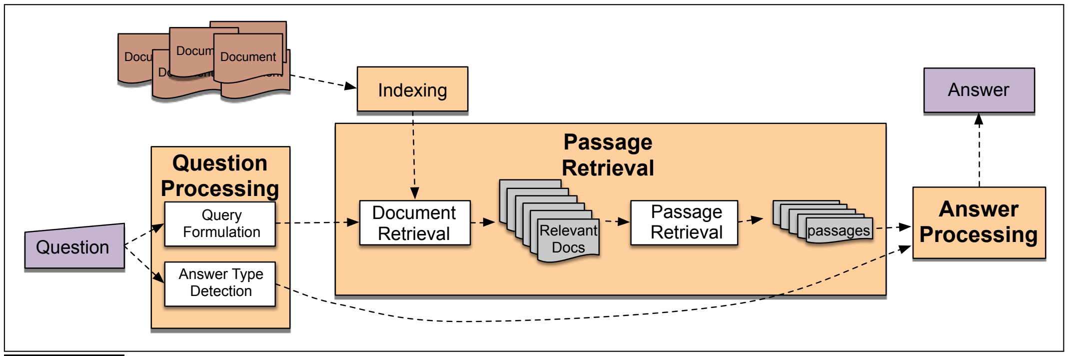


Figure 24.2 IR-based factoid question answering has three stages: question processing, passage retrieval, and answer processing.

also extract a **focus**, which is the string of words in the question that are likely to be replaced by the answer in any answer string found. Some systems also classify the **question type**: is this a definition question, a math question, a list question? For example, for the following question:

Which US state capital has the largest population?

The query processing should produce results like the following:

Answer Type: city
Query: US state capital, largest, population
Focus: state capital

In the next two sections we summarize the two most commonly used tasks, answer type detection and query formulation.

24.1.2 Answer Type Detection (Question Classification)

question classification
answer type

The task of **question classification** or **answer type recognition** is to determine the **answer type**, the named-entity or similar class categorizing the answer. A question like *"Who founded Virgin Airlines?"* expects an answer of type PERSON. A question like *"What Canadian city has the largest population?"* expects an answer of type CITY. If we know the answer type for a question, we can avoid looking at every sentence or noun phrase in the entire suite of documents for the answer, instead focusing on, for example, just people or cities.

answer type taxonomy

As some of the above examples suggest, we might draw the set of possible answer types for a question classifier from a set of named entities like PERSON, LOCATION, and ORGANIZATION described in Chapter 17. Usually, however, a richer, often hierarchical set of answer types is used, an **answer type taxonomy**. Such taxonomies can be built semi-automatically and dynamically, for example, from WordNet (Harabagiu et al. 2000, Pasca 2003), or they can be designed by hand.

Figure 24.4 shows one such hand-built ontology, the Li and Roth (2005) tagset; a subset is shown graphically in Fig. 24.3. In this hierarchical tagset, each question can be labeled with a coarse-grained tag like HUMAN or a fine-grained tag like HUMAN:DESCRIPTION, HUMAN:GROUP, HUMAN:IND, and so on. Similar tags are used in other systems; the HUMAN:DESCRIPTION type is often called a BIOGRAPHY question because the answer is required to give a brief biography of the person rather than just a name.

Question classifiers can be built by hand-writing rules, by supervised machine learning, or with some combination. The Webclopedia QA Typology, for example,

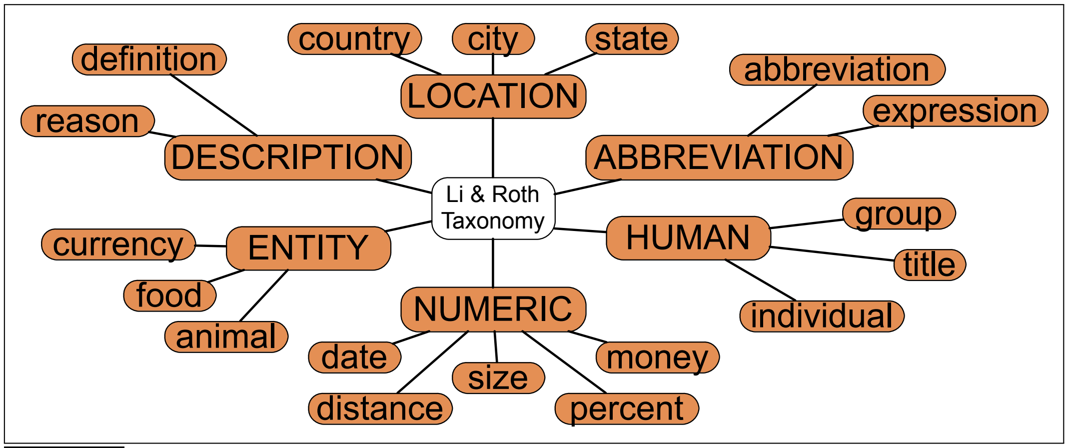


Figure 24.3 A subset of the Li and Roth (2005) answer types.

contains 276 hand-written rules associated with the approximately 180 answer types in the typology (Hovy et al., 2002). A regular expression rule for detecting an answer type like BIOGRAPHY (which assumes the question has been named-entity-tagged) might be

(24.4) who {is | was | are | were} PERSON

Most modern question classifiers, however, are based on supervised machine learning, and are trained on databases of questions that have been hand-labeled with an answer type (Li and Roth, 2002). Typical features used for classification include the words in the questions, the part-of-speech of each word, and named entities in the questions.

Often, a single word in the question gives extra information about the answer type, and its identity is used as a feature. This word is sometimes called the **answer type word** or **question headword**, and may be defined as the headword of the first NP after the question's *wh-word*; headwords are indicated in boldface in the following examples:

(24.5) Which **city** in China has the largest number of foreign financial companies?

(24.6) What is the state **flower** of California?

Finally, it often helps to use semantic information about the words in the questions. The WordNet synset ID of the word can be used as a feature, as can the IDs of the hypernym and hyponyms of each word in the question.

In general, question classification accuracies are relatively high on easy question types like PERSON, LOCATION, and TIME questions; detecting REASON and DESCRIPTION questions can be much harder.

24.1.3 Query Formulation

Query formulation is the task of creating from the question a list of keywords that form a query that can be sent to an information retrieval system. Exactly what query to form depends on the application. If question answering is applied to the Web, we might simply create a keyword from every word in the question, letting the Web search engine automatically remove any stopwords. Often, we leave out the question word (*where*, *when*, etc.). Alternatively, keywords can be formed from only the terms found in the noun phrases in the question, applying stopword lists to ignore function words and high-frequency, low-content verbs.

Tag	Example
ABBREVIATION	
abb	What's the abbreviation for limited partnership?
exp	What does the "c" stand for in the equation E=mc2?
DESCRIPTION	
definition	What are tannins?
description	What are the words to the Canadian National anthem?
manner	How can you get rust stains out of clothing?
reason	What caused the Titanic to sink?
ENTITY	
animal	What are the names of Odin's ravens?
body	What part of your body contains the corpus callosum?
color	What colors make up a rainbow?
creative	In what book can I find the story of Aladdin?
currency	What currency is used in China?
disease/medicine	What does Salk vaccine prevent?
event	What war involved the battle of Chapultepec?
food	What kind of nuts are used in marzipan?
instrument	What instrument does Max Roach play?
lang	What's the official language of Algeria?
letter	What letter appears on the cold-water tap in Spain?
other	What is the name of King Arthur's sword?
plant	What are some fragrant white climbing roses?
product	What is the fastest computer?
religion	What religion has the most members?
sport	What was the name of the ball game played by the Mayans?
substance	What fuel do airplanes use?
symbol	What is the chemical symbol for nitrogen?
technique	What is the best way to remove wallpaper?
term	How do you say " Grandma" in Irish?
vehicle	What was the name of Captain Bligh's ship?
word	What's the singular of dice?
HUMAN	
description	Who was Confucius?
group	What are the major companies that are part of Dow Jones?
ind	Who was the first Russian astronaut to do a spacewalk?
title	What was Queen Victoria's title regarding India?
LOCATION	
city	What's the oldest capital city in the Americas?
country	What country borders the most others?
mountain	What is the highest peak in Africa?
other	What river runs through Liverpool?
state	What states do not have state income tax?
NUMERIC	
code	What is the telephone number for the University of Colorado?
count	About how many soldiers died in World War II?
date	What is the date of Boxing Day?
distance	How long was Mao's 1930s Long March?
money	How much did a McDonald's hamburger cost in 1963?
order	Where does Shanghai rank among world cities in population?
other	What is the population of Mexico?
period	What was the average life expectancy during the Stone Age?
percent	What fraction of a beaver's life is spent swimming?
temp	How hot should the oven be when making Peachy Oat Muffins?
speed	How fast must a spacecraft travel to escape Earth's gravity?
size	What is the size of Argentina?
weight	How many pounds are there in a stone?

Figure 24.4 Question typology from Li and Roth (2002), (2005). Example sentences are from their corpus of 5500 labeled questions. A question can be labeled either with a coarse-grained tag like HUMAN or NUMERIC or with a fine-grained tag like HUMAN:DESCRIPTION, HUMAN:GROUP, HUMAN:IND, and so on.

When question answering is applied to smaller sets of documents, for example, to answer questions about corporate information pages, we still use an IR engine to search our documents for us. But for this smaller set of documents, we generally need to apply query expansion. On the Web the answer to a question might appear in many different forms, so if we search with words from the question we'll probably find an answer written in the same form. In smaller sets of corporate pages, by contrast, an answer might appear only once, and the exact wording might look nothing like the question. Thus, query expansion methods can add query terms in hopes of matching the particular form of the answer as it appears. These might include all morphological variants of the content words in the question, or synonyms from a thesaurus.

query reformulation

A query formulation approach that is sometimes used for questioning the Web is to apply **query reformulation** rules to the query. The rules rephrase the question to make it look like a substring of possible declarative answers. The question *"when was the laser invented?"* might be reformulated as *"the laser was invented"*; the question *"where is the Valley of the Kings?"* as *"the Valley of the Kings is located in"*. Here are some sample hand-written reformulation rules from Lin (2007):

(24.7) *wh-word* did A *verb* B $\rightarrow \ldots$ A *verb*+ed B

(24.8) Where is A $\rightarrow$ A is located in

24.1.4 Passage Retrieval

The query that was created in the question-processing phase is next used to query an information-retrieval system, either a general IR engine over a proprietary set of indexed documents or a Web search engine. The result of this document retrieval stage is a set of documents.

Although the set of documents is generally ranked by relevance, the top-ranked document is probably not the answer to the question. This is because documents are not an appropriate unit to rank with respect to the goals of a question-answering system. A highly relevant and large document that does not prominently answer a question is not an ideal candidate for further processing.

Therefore, the next stage is to extract a set of potential answer passages from the retrieved set of documents. The definition of a passage is necessarily system dependent, but the typical units include sections, paragraphs, and sentences. We might run a paragraph segmentation algorithm on all the returned documents and treat each paragraph as a segment.

passage retrieval

We next perform **passage retrieval**. In this stage, we first filter out passages in the returned documents that don't contain potential answers and then rank the rest according to how likely they are to contain an answer to the question. The first step in this process is to run a named entity or answer type classification on the retrieved passages. The answer type that we determined from the question tells us the possible answer types we expect to see in the answer. We can therefore filter out documents that don't contain any entities of the right type.

The remaining passages are then ranked, usually by supervised machine learning, relying on a small set of features that can be easily extracted from a potentially large number of answer passages, such as:

- The number of **named entities** of the right type in the passage
- The number of **question keywords** in the passage
- The longest exact sequence of question keywords that occurs in the passage
- The rank of the document from which the passage was extracted

- The **proximity** of the keywords from the original query to each other
  For each passage identify the shortest span that covers the keywords contained in that passage. Prefer smaller spans that include more keywords (Pasca 2003, Monz 2004).
- The *N*-**gram overlap** between the passage and the question
  Count the *N*-grams in the question and the *N*-grams in the answer passages. Prefer the passages with higher *N*-gram overlap with the question (Brill et al., 2002).

For question answering from the Web, instead of extracting passages from all returned documents, we can rely on the Web search to do passage extraction for us. We do this by using **snippets** produced by the Web search engine as the returned passages. For example, Fig. 24.5 shows snippets for the first five documents returned from Google for the query *When was movable type metal printing invented in Korea?*

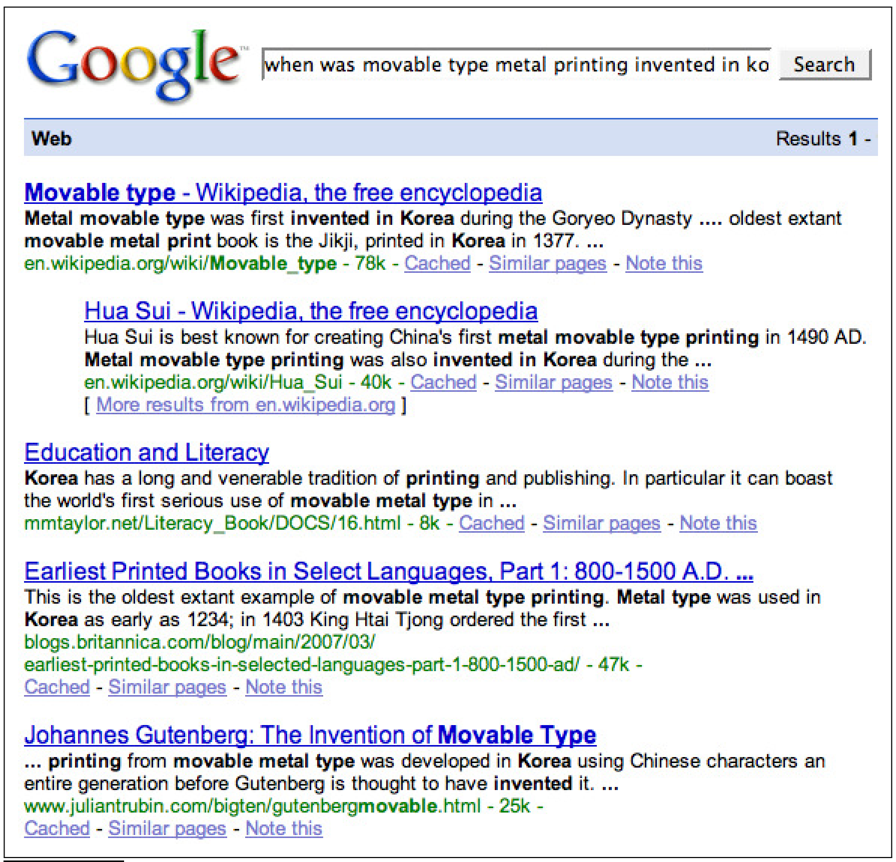


Figure 24.5 Five snippets from Google in response to the query *When was movable type metal printing invented in Korea?*

24.1.5 Answer Processing

The final stage of question answering is to extract a specific answer from the passage so as to be able to present the user with an answer like *29,029 feet* to the question *"How tall is Mt. Everest?"*

Two classes of algorithms have been applied to the answer-extraction task, one based on **answer-type pattern extraction** and one based on **N-gram tiling**.

In the **pattern-extraction** methods for answer processing, we use information about the expected answer type together with regular expression patterns. For example, for questions with a HUMAN answer type, we run the answer type or named entity tagger on the candidate passage or sentence and return whatever entity is labeled with type HUMAN. Thus, in the following examples, the underlined named entities are extracted from the candidate answer passages as the answer to the HUMAN and DISTANCE-QUANTITY questions:

"Who is the prime minister of India"
Manmohan Singh, Prime Minister of India, had told left leaders that the deal would not be renegotiated.

"How tall is Mt. Everest?"
The official height of Mount Everest is 29029 feet

Unfortunately, the answers to some questions, such as DEFINITION questions, don't tend to be of a particular named entity type. For some questions, then, instead of using answer types, we use hand-written regular expression patterns to help extract the answer. These patterns are also useful in cases in which a passage contains multiple examples of the same named entity type. Figure 24.6 shows some patterns from Pasca (2003) for the question phrase (QP) and answer phrase (AP) of definition questions.

Pattern	Question	Answer
<AP> such as <QP>	What is autism?	", developmental disorders such as autism"
<QP>, a <AP>	What is a caldera?	"the Long Valley caldera, a volcanic crater 19 miles long"

Figure 24.6 Some answer-extraction patterns for definition questions (Pasca, 2003).

The patterns are specific to each question type and can either be written by hand or learned automatically using relation extraction methods. Patterns can then be used together with other information as features in a classifier that ranks candidate answers. We extract potential answers by using named entities or patterns or even just by looking at every sentence returned from passage retrieval and rank them using a classifier with features like the following.

Answer type match: True if the candidate answer contains a phrase with the correct answer type.

Pattern match: The identity of a pattern that matches the candidate answer.

Number of matched question keywords: How many question keywords are contained in the candidate answer.

Keyword distance: The distance between the candidate answer and query keywords (measured in average number of words or as the number of keywords that occur in the same syntactic phrase as the candidate answer).

Novelty factor: True if at least one word in the candidate answer is novel, that is, not in the query.

Apposition features: True if the candidate answer is an appositive to a phrase containing many question terms. Can be approximated by the number of question terms separated from the candidate answer through at most three words and one comma (Pasca, 2003).

Punctuation location: True if the candidate answer is immediately followed by a comma, period, quotation marks, semicolon, or exclamation mark.

Sequences of question terms: The length of the longest sequence of question terms that occurs in the candidate answer.

N-gram tiling

An alternative approach to answer extraction, used solely in Web search, is based on **N-gram tiling**, sometimes called the **redundancy-based approach** (Brill et al. 2002, Lin 2007). This simplified method begins with the snippets returned from the Web search engine, produced by a reformulated query. In the first step, **N-gram mining**, every unigram, bigram, and trigram occurring in the snippet is extracted and weighted. The weight is a function of the number of snippets in which the N-gram occurred, and the weight of the query reformulation pattern that returned it. In the **N-gram filtering** step, N-grams are scored by how well they match the predicted answer type. These scores are computed by hand-written filters built for each answer type. Finally, an **N-gram tiling** algorithm concatenates overlapping N-gram fragments into longer answers. A standard greedy method is to start with the highest-scoring candidate and try to tile each other candidate with this candidate. The best-scoring concatenation is added to the set of candidates, the lower-scoring candidate is removed, and the process continues until a single answer is built.

N-gram mining

N-gram filtering

For any of these answer-extraction methods, the exact answer phrase can just be presented to the user by itself, or, more helpfully, accompanied by enough passage information to provide helpful context.

24.2 Knowledge-based Question Answering

While an enormous amount of information is encoded in the vast amount of text on the web, information obviously also exists in more structured forms. We use the term **knowledge-based question answering** for the idea of answering a natural language question by mapping it to a query over a structured database. Like the text-based paradigm for question answering, this approach dates back to the earliest days of natural language processing, with systems like BASEBALL (Green et al., 1961) that answered questions from a structured database of baseball games and stats.

Systems for mapping from a text string to any logical form are called **semantic parsers** (Chapter 16). Semantic parsers for question answering usually map either to some version of predicate calculus or a query language like SQL or SPARQL, as in the examples in Fig. 24.7.

Question	**Logical form**
When was Ada Lovelace born?	`birth-year (Ada Lovelace, ?x)`
What states border Texas?	λ x.state(x) $\wedge$ borders(x,texas)
What is the largest state	argmax(λx.state(x), λx.size(x))
How many people survived the sinking of the Titanic	`(count (!fb:event.disaster.survivors fb:en.sinking_of_the_titanic))`

Figure 24.7 Sample logical forms produced by a semantic parser for question answering. These range from simple relations like `birth-year`, or relations normalized to databases like Freebase, to full predicate calculus.

The logical form of the question is thus either in the form of a query or can easily be converted into one. The database can be a full relational database, or simpler structured databases like sets of **RDF triples**. Recall from Chapter 17 that an RDF triple is a 3-tuple, a predicate with two arguments, expressing some simple relation

or proposition. Popular ontologies like Freebase (Bollacker et al., 2008) or DBpedia (Bizer et al., 2009) have large numbers of triples derived from Wikipedia **infoboxes**, the structured tables associated with certain Wikipedia articles.

The simplest formation of the knowledge-based question answering task is to answer factoid questions that ask about one of the missing arguments in a triple. Consider an RDF triple like the following:

subject	**predicate**	**object**
Ada Lovelace	birth-year	1815

This triple can be used to answer text questions like 'When was Ada Lovelace born?' or 'Who was born in 1815?'. Question answering in this paradigm requires mapping from textual strings like "When was ... born" to canonical relations in the knowledge base like *birth-year*. We might sketch this task as:

"When was Ada Lovelace born?" → `birth-year (Ada Lovelace, ?x)`
"What is the capital of England?" → `capital-city(?x, England)`

24.2.1 Rule-based Methods

For relations that are very frequent, it may be worthwhile to write hand-written rules to extract relations from the question, just as we saw in Section 17.2. For example, to extract the birth-year relation, we could write patterns that search for the question word *When*, a main verb like *born*, and that extract the named entity argument of the verb.

24.2.2 Supervised Methods

In some cases we have supervised data, consisting of a set of questions paired with their correct logical form like the examples in Fig. 24.7. The task is then to take those pairs of training tuples and produce a system that maps from new questions to their logical forms.

Most supervised algorithms for learning to answer these simple questions about relations first parse the questions and then align the parse trees to the logical form. Generally these systems bootstrap by having a small set of rules for building this mapping, and an initial lexicon as well. For example, a system might have built-in strings for each of the entities in the system (`Texas`, `Ada Lovelace`), and then have simple default rules mapping fragments of the question parse tree to particular relations:

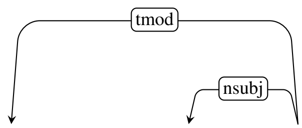
```
 nsubj  dobj
Who V ENTITY → relation( ?x, entity)

  tmod  nsubj
When V ENTITY → relation( ?x, entity)
```

Then given these rules and the lexicon, a training tuple like the following:

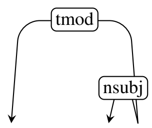
"When was Ada Lovelace born?" → `birth-year (Ada Lovelace, ?x)`

would first be parsed, resulting in the following mapping.



`When was Ada Lovelace` born $\rightarrow$ `birth-year(Ada Lovelace, ?x)`

From many pairs like this, we could induce mappings between pieces of parse fragment, such as the mapping between the parse fragment on the left and the relation on the right:



`When was` · born $\rightarrow$ `birth-year( , ?x)`

A supervised system would thus parse each tuple in the training set and induce a bigger set of such specific rules, allowing it to map unseen examples of "When was X born?" questions to the `birth-year` relation. Rules can furthermore be associated with counts based on the number of times the rule is used to parse the training data. Like rule counts for probabilistic grammars, these can be normalized into probabilities. The probabilities can then be used to choose the highest probability parse for sentences with multiple semantic interpretations.

The supervised approach can be extended to deal with more complex questions that are not just about single relations. Consider the question *What is the biggest state bordering Texas?* from the GEOQUERY (Zelle and Mooney, 1996) dataset, with the semantic form:

$\text{argmax}(\lambda x.state(x) \wedge borders(x, texas), \lambda x.size(x))$

This question has much more complex structures than the simple single-relation questions we considered above, such as the argmax function, the mapping of the word *biggest* to *size* and so on. Zettlemoyer and Collins (2005) shows how more complex default rules (along with richer syntactic structures) can be used to learn to map from text sentences to more complex logical forms. The rules take the training set's pairings of sentence and meaning as above and use the complex rules to break each training example down into smaller tuples that can then be recombined to parse new sentences.

24.2.3 Dealing with Variation: Semi-Supervised Methods

Because it is difficult to create training sets with questions labeled with their meaning representation, supervised datasets can't cover the wide variety of forms that even simple factoid questions can take. For this reason most techniques for mapping factoid questions to the canonical relations or other structures in knowledge bases find some way to make use of textual redundancy.

The most common source of redundancy, of course, is the web, which contains vast number of textual variants expressing any relation. For this reason, most methods make some use of web text, either via semi-supervised methods like **distant supervision** or unsupervised methods like **open information extraction**, both introduced in Chapter 17. For example the REVERB open information extractor (Fader et al., 2011) extracts billions of (subject, relation, object) triples of strings from the web, such as ("Ada Lovelace","was born in", "1815"). By **aligning** these strings with a canonical knowledge source like Wikipedia, we create new relations that can be queried while simultaneously learning to map between the words in question and

canonical relations.

To align a REVERB triple with a canonical knowledge source we first align the arguments and then the predicate. Recall from Chapter 21 that linking a string like "Ada Lovelace" with a Wikipedia page is called **entity linking**; we thus represent the concept 'Ada Lovelace' by a unique identifier of a Wikipedia page. If this subject string is not associated with a unique page on Wikipedia, we can disambiguate which page is being sought, for example by using the cosine distance between the triple string ('Ada Lovelace was born in 1815') and each candidate Wikipedia page. Date strings like '1815' can be turned into a normalized form using standard tools for temporal normalization like SUTime (Chang and Manning, 2012). Once we've aligned the arguments, we align the predicates. Given the Freebase relation `people.person.birthdate(ada lovelace,1815)` and the string 'Ada Lovelace was born in 1815', having linked Ada Lovelace and normalized 1815, we learn the mapping between the string 'was born in' and the relation people.person.birthdate. In the simplest case, this can be done by aligning the relation with the string of words in between the arguments; more complex alignment algorithms like IBM Model 1 (Chapter 23) can be used. Then if a phrase aligns with a predicate across many entities, it can be extracted into a lexicon for mapping questions to relations.

Here are some examples from such a resulting lexicon, produced by Berant et al. (2013), giving many variants of phrases that align with the Freebase relation `country.capital` between a country and its capital city:

capital of	capital city of	become capital of
capitol of	national capital of	official capital of
political capital of	administrative capital of	beautiful capital of
capitol city of	remain capital of	make capital of
political center of	bustling capital of	capital city in
cosmopolitan capital of	move its capital to	modern capital of
federal capital of	beautiful capital city of	administrative capital city of

Figure 24.8 Some phrases that align with the Freebase relation `country.capital` from Berant et al. (2013).

Another useful source of linguistic redundancy are paraphrase databases. For example the site `wikianswers.com` contains millions of pairs of questions that users have tagged as having the same meaning, 18 million of which have been collected in the PARALEX corpus (Fader et al., 2013). Here's an example:

Q: What are the green blobs in plant cells?

Lemmatized synonyms from PARALEX:

what be the green blob in plant cell?
what be green part in plant cell?
what be the green part of a plant cell?
what be the green substance in plant cell?
what be the part of plant cell that give it green color?
what cell part do plant have that enable the plant to be give a green color?
what part of the plant cell turn it green?
part of the plant cell where the cell get it green color?
the green part in a plant be call?
the part of the plant cell that make the plant green be call?

The resulting millions of pairs of question paraphrases can be aligned to each other using MT alignment approaches (such as IBM Model 1) to create an MT-style

phrase table for translating from question phrases to synonymous phrases. These are used by a number of modern question answering algorithms, generating all paraphrases of a question as part of the process of finding an answer (Fader et al. 2013, Berant and Liang 2014).

24.3 Using multiple information sources: IBM's Watson

Of course there is no reason to limit ourselves to just text-based or knowledge-based resources for question answering. The Watson system from IBM that won the Jeopardy! challenge in 2011 is an example of a system that relies on a wide variety of resources to answer questions.

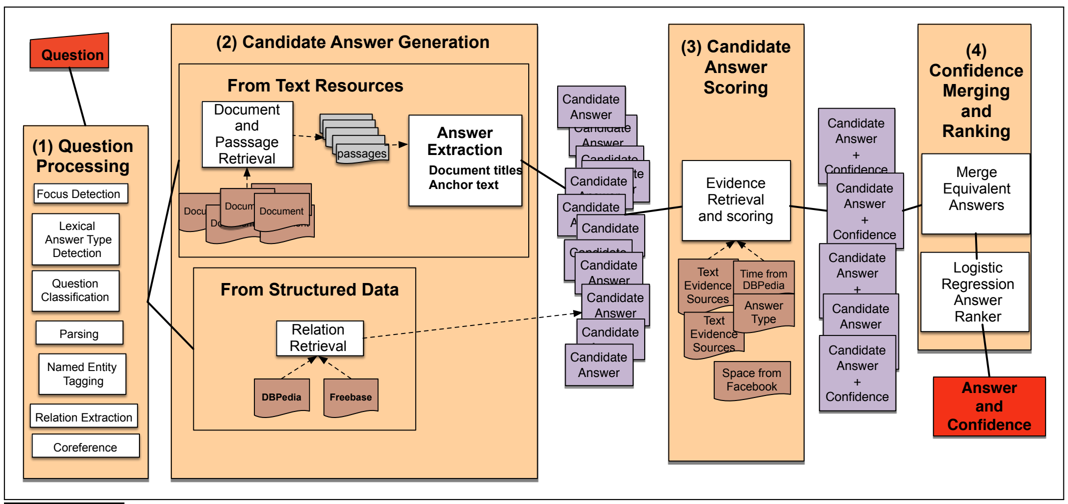


Figure 24.9 The 4 broad stages of Watson QA: (1) Question Processing, (2) Candidate Answer Generation, (3) Candidate Answer Scoring, and (4) Answer Merging and Confidence Scoring.

Figure 24.9 shows the 4 stages of the DeepQA system that is the question answering component of Watson.

The first stage is **question processing**. The DeepQA system runs parsing, named entity tagging, and relation extraction on the question. Then, like the text-based systems in Section 24.1, the DeepQA system extracts the **focus**, the **answer type** (also called the **lexical answer type** or **LAT**), and performs **question classification** and **question sectioning**.

Consider these Jeopardy! examples, with a category followed by a question:

> Poets and Poetry: **He** was a bank clerk in the Yukon before he published "Songs of a Sourdough" in 1907.
> THEATRE: A new play based on **this Sir Arthur Conan Doyle canine classic** opened on the London stage in 2007.

The questions are parsed, named entities are extracted (*Sir Arthur Conan Doyle* identified as a PERSON, Yukon as a GEOPOLITICAL ENTITY, "Songs of a Sourdough" as a COMPOSITION), coreference is run (*he* is linked with *clerk*) and relations like the following are extracted:

> authorof(focus,"Songs of a sourdough")
> publish (e1, he, "Songs of a sourdough")

in (e2, e1, 1907)
temporallink(publish(...), 1907)

focus

Next DeepQA extracts the question **focus**, shown in bold in both examples. The focus is the part of the question that co-refers with the answer, used for example to align with a supporting passage. The focus is extracted by hand-written rules—made possible by the relatively stylized syntax of Jeopardy! questions—such as a rule extracting any noun phrase with determiner "this" as in the Conan Doyle example, and rules extracting pronouns like *she*, *he*, *hers*, *him*, as in the poet example.

lexical answer type

The **lexical answer type** (shown in blue above) is a word or words which tell us something about the semantic type of the answer. Because of the wide variety of questions in Jeopardy!, Jeopardy! uses a far larger set of answer types than the sets for standard factoid algorithms like the one shown in Fig. 24.4. Even a large set of named entity tags is insufficient to define a set of answer types. The DeepQA team investigated a set of 20,000 questions and found that a named entity tagger with over 100 named entity types covered less than half the types in these questions. Thus DeepQA extracts a wide variety of words to be answer types; roughly 5,000 lexical answer types occurred in the 20,000 questions they investigated, often with multiple answer types in each question.

These lexical answer types are again extracted by rules: the default rule is to choose the syntactic headword of the focus. Other rules improve this default choice. For example additional lexical answer types can be words in the question that are coreferent with or have a particular syntactic relation with the focus, such as headwords of appositives or predicative nominatives of the focus. In some cases even the Jeopardy! category can act as a lexical answer type, if it refers to a type of entity that is compatible with the other lexical answer types. Thus in the first case above, *he*, *poet*, and *clerk* are all lexical answer types. In addition to using the rules directly as a classifier, they can instead be used as features in a logisitic regression classifier that can return a probability as well as a lexical answer type.

Note that answer types function quite differently in DeepQA than the purely IR-based factoid question answerers. In the algorithm described in Section 24.1, we determine the answer type, and then use a strict filtering algorithm only considering text strings that have exactly that type. In DeepQA, by contrast, we extract lots of answers, unconstrained by answer type, and a set of answer types, and then in the later 'candidate answer scoring' phase, we simply score how well each answer fits the answer types as one of many sources of evidence.

Finally the question is classified by type (definition question, multiple-choice, puzzle, fill-in-the-blank). This is generally done by writing pattern-matching regular expressions over words or parse trees.

In the second **candidate answer generation** stage, we combine the processed question with external documents and other knowledge sources to suggest many candidate answers. These candidate answers can either be extracted from text documents or from structured knowledge bases.

For structured resources like DBpedia, IMDB, or the triples produced by Open Information Extraction, we can just query these stores with the relation and the known entity, just as we saw in Section 24.2. Thus if we have extracted the relation `authorof(focus,"Songs of a sourdough")`, we can query a triple store with `authorof(?x,"Songs of a sourdough")` to return the correct author.

The method for extracting answers from text depends on the type of text documents. To extract answers from normal text documents we can do passage search

just as we did in Section 24.1. As we did in that section, we need to generate a query from the question; for DeepQA this is generally done by eliminating stop words, and then upweighting any terms which occur in any relation with the focus. For example from this query:

> MOVIE-"ING": Robert Redford and Paul Newman starred in this depression-era grifter flick. *(Answer: "The Sting")*

the following weighted query might be extracted:

> (2.0 Robert Redford) (2.0 Paul Newman) star depression era grifter (1.5 flick)

The query can now be passed to a standard IR system. Some systems are already set up to allow retrieval of short passages, and the system can just return the ten 1-2 sentence passages that are needed for the next stage. Alternatively the query can be passed to a standard document retrieval engine, and then from each returned document passages are selected that are longer, toward the front, and have more named entities.

DeepQA also makes use of the convenient fact that the vast majority of Jeopardy! answers are the title of a Wikipedia document. To find these titles, we can do a second text retrieval pass specifically on Wikipedia documents. Then instead of extracting passages from the retrieved Wikipedia document, we directly return the titles of the highly ranked retrieved documents as the possible answers.

Once we have a set of passages, we need to extract candidate answers. As we just said, if the document is a Wikipedia page, we can just take the title, but for other texts, like news documents, we need other approaches. Two common approaches are to extract all **anchor texts** in the document (anchor text is the text between `<a>` and `<\a>` used to point to a URL in an HTML page), or to extract all noun phrases in the passage that are Wikipedia document titles.

anchor texts

The third **candidate answer scoring** stage uses many sources of evidence to score the candidates. One of the most important is the lexical answer type. DeepQA includes a system that takes a candidate answer and a lexical answer type and returns a score indicating whether the candidate answer can be interpreted as a subclass or instance of the answer type. Consider the candidate "difficulty swallowing" and the lexical answer type "manifestation". DeepQA first matches each of these words with possible entities in ontologies like DBpedia and WordNet. Thus the candidate "difficulty swallowing" is matched with the DBpedia entity "Dysphagia", and then that instance is mapped to the WordNet type "Symptom". The answer type "manifestation" is mapped to the WordNet type "Condition". The system looks for a link of hyponymy, instance-of or synonymy between these two types; in this case a hyponymy relation is found between "Symptom" and "Condition".

Other scorers are based on using time and space relations extracted from DBpedia or other structured databases. For example, we can extract temporal properties of the entity (when was a person born, when died) and then compare to time expressions in the question. If a time expression in the question occurs chronologically before a person was born, that would be evidence against this person being the answer to the question.

Finally, we can use text retrieval to help retrieve evidence supporting a candidate answer. We can retrieve passages with terms matching the question, then replace the focus in the question with the candidate answer and measure the overlapping words or ordering of the passage with the modified question.

The output of this stage is a set of candidate answers, each with a vector of scoring features.

In the final **answer merging and scoring** step, we first merge candidate answers that are equivalent. Thus if we had extracted two candidate answers *J.F.K.* and *John F. Kennedy*, this stage would merge the two into a single candidate. For proper nouns, automatically generated name dictionaries can help in this task. One useful kind of resource is the large synonym dictionaries that are created by listing all anchor text strings that point to the same Wikipedia page; such dictionaries give large numbers of synonyms for each Wikipedia title — e.g., *JFK, John F. Kennedy, John Fitzgerald Kennedy, Senator John F. Kennedy, President Kennedy, Jack Kennedy*, etc. (Spitkovsky and Chang, 2012). For common nouns, we can use morphological parsing to merge candidates which are morphological variants.

We then merge the evidence for each variant, combining the scoring feature vectors for the merged candidates into a single vector.

Now we have a set of candidates, each with a feature vector. A regularized logistic regression classifier is used to take each feature vector and assign a single confidence value to this candidate answer. The classifier is trained on thousands of candidate answers, each labeled for whether it is correct or incorrect, together with their feature vectors, and learning to predict a probability of being a correct answer. Since, in training, there are far more incorrect answers than correct answers, we need to use one of the standard techniques for dealing with very imbalanced data. DeepQA uses *instance weighting*, assigning an instance weight of .5 for each incorrect answer example in training. The candidate answers are then sorted by this confidence value, resulting in a single best answer.

The merging and ranking is actually run iteratively; first the candidates are ranked by the classifier, giving a rough first value for each candidate answer, then that value is used to decide which of the variants of a name to select as the merged answer, then the merged answers are re-ranked,.

In summary, we've seen in the four stages of DeepQA that it draws on the intuitions of both the IR-based and knowledge-based paradigms. Indeed, Watson's architectural innovation is its reliance on proposing a very large number of candidate answers from both text-based and knowledge-based sources and then developing a wide variety of evidence features for scoring these candidates —again both text-based and knowledge-based. Of course the Watson system has many more components for dealing with rare and complex questions, and for strategic decisions in playing Jeopardy!; see the papers mentioned at the end of the chapter for many more details.

24.4 Evaluation of Factoid Answers

mean reciprocal rank
MRR

A common evaluation metric for factoid question answering, introduced in the TREC Q/A track in 1999, is **mean reciprocal rank**, or **MRR**. MRR assumes a test set of questions that have been human-labeled with correct answers. MRR also assumes that systems are returning a short **ranked** list of answers or passages containing answers. Each question is then scored according to the reciprocal of the **rank** of the first correct answer. For example if the system returned five answers but the first three are wrong and hence the highest-ranked correct answer is ranked fourth, the reciprocal rank score for that question would be $\frac{1}{4}$. Questions with return sets that do not contain any correct answers are assigned a zero. The score of a system is then the average of the score for each question in the set. More formally, for an evaluation of a system returning a set of ranked answers for a test set consisting of

N questions, the MRR is defined as

$$\text{MRR} = \frac{1}{N} \sum_{i=1 \text{ s.t. } rank_i \neq 0}^{N} \frac{1}{rank_i} \tag{24.9}$$

FREE917

A number of test sets are available for question answering. Early systems used the TREC QA dataset; questions and hand-written answers for TREC competitions from 1999 to 2004 are publicly available. **FREE917** (Cai and Yates, 2013) has 917 questions manually created by annotators, each paired with a meaning representation; example questions include:

How many people survived the sinking of the Titanic?
What is the average temperature in Sydney in August?
When did Mount Fuji last erupt?

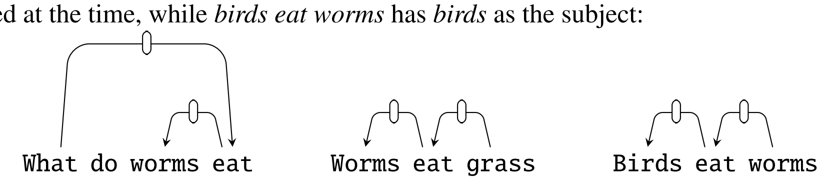
WEBQUESTIONS

WEBQUESTIONS (Berant et al., 2013) contains 5,810 questions asked by web users, each beginning with a wh-word and containing exactly one entity. Questions are paired with hand-written answers drawn from the Freebase page of the question's entity, and were extracted from Google Suggest by breadth-first search (start with a seed question, remove some words, use Google Suggest to suggest likely alternative question candidates, remove some words, etc.). Some examples:

What character did Natalie Portman play in Star Wars?
What airport is closest to Palm Springs?
Which countries share land border with Vietnam?
What present day countries use English as their national language?

Bibliographical and Historical Notes

Question answering was one of the earliest NLP tasks, and early versions of the text-based and knowledge-based paradigms were developed by the very early 1960s. The text-based algorithms generally relied on simple parsing of the question and of the sentences in the document, and then looking for matches. This approach was used very early on (Phillips, 1960) but perhaps the most complete early system, and one that strikingly prefigures modern relation-based systems, was the Protosynthex system of Simmons et al. (1964). Given a question, Protosynthex first formed a query from the content words in the question, and then retrieved candidate answer sentences in the document, ranked by their frequency-weighted term overlap with the question. The query and each retrieved sentence were then parsed with dependency parsers, and the sentence whose structure best matches the question structure selected. Thus the question *What do worms eat?* would match *worms eat grass*: both have the subject *worms* as a dependent of *eat*, in the version of dependency grammar used at the time, while *birds eat worms* has *birds* as the subject:



The alternative knowledge-based paradigm was implemented in the BASEBALL system (Green et al., 1961). This system answered questions about baseball games like "Where did the Red Sox play on July 7" by querying a structured database of game information. The database was stored as a kind of attribute-value matrix with values for attributes of each game:

```
Month = July
   Place = Boston
       Day  = 7
       Game Serial No.  = 96
       (Team = Red Sox, Score = 5)
       (Team = Yankees, Score = 3)
```

Each question was constituency-parsed using the algorithm of Zellig Harris's TDAP project at the University of Pennsylvania, essentially a cascade of finite-state transducers (see the historical discussion in Joshi and Hopely 1999 and Karttunen 1999). Then a content analysis phase each word or phrase was associated with a program that computed parts of its meaning. Thus the phrase 'Where' had code to assign the semantics `Place = ?"`, with the result that the question "Where did the Red Sox play on July 7" was assigned the meaning

```
Place = ?
Team = Red Sox
Month = July
Day = 7
```

The question is then matched against the database to return to the answer. Simmons (1965) summarizes other early QA systems.

Another important progenitor of the knowledge-based paradigm for question-answering is work that used predicate calculus as the meaning representation language. The **LUNAR** system (Woods et al. 1972,Woods 1978) was designed to be a natural language interface to a database of chemical facts about lunar geology. It could answer questions like *Do any samples have greater than 13 percent aluminum* by parsing them into a logical form

LUNAR

(TEST (FOR SOME X16 / (SEQ SAMPLES) : T ; (CONTAIN' X16 (NPR* X17 / (QUOTE AL203)) (GREATERTHAN 13PCT))))

The rise of the web brought the information-retrieval paradigm for question answering to the forefront with the TREC QA track beginning in 1999, leading to a wide variety of factoid and non-factoid systems competing in annual evaluations.

The DeepQA component of the Watson system that won the Jeopardy! challenge is described in a series of papers in volume 56 of the IBM Journal of Research and Development; see for example Ferrucci (2012), Lally et al. (2012), Chu-Carroll et al. (2012), Murdock et al. (2012b), Murdock et al. (2012a), Kalyanpur et al. (2012), and Gondek et al. (2012).

Question answering is also an important function of modern personal assistant dialog systems; see Chapter 25 for more.

Exercises

CHAPTER

25 Dialog Systems and Chatbots

Les lois de la conversation sont en général de ne s'y appesantir sur aucun objet, mais de passer légèrement, sans effort et sans affectation, d'un sujet à un autre ; de savoir y parler de choses frivoles comme de choses sérieuses

The rules of conversation are, in general, not to dwell on any one subject, but to pass lightly from one to another without effort and without affectation; to know how to speak about trivial topics as well as serious ones;

The 18th C. *Encyclopedia* of Diderot, start of the entry on conversation

The literature of the fantastic abounds in inanimate objects magically endowed with sentience and the gift of speech. From Ovid's statue of Pygmalion to Mary Shelley's Frankenstein, there is something deeply moving about creating something and then having a chat with it. Legend has it that after finishing his sculpture *Moses*, Michelangelo thought it so lifelike that he tapped it on the knee and commanded it to speak. Perhaps this shouldn't be surprising. Language is the mark of humanity and sentience, and **conversation** or **dialog** is the most fundamental and specially privileged arena of language. It is the first kind of language we learn as children, and for most of us, it is the kind of language we most commonly indulge in, whether we are ordering curry for lunch or buying spinach, participating in business meetings or talking with our families, booking airline flights or complaining about the weather.

conversation

dialog

conversational agent

dialog system

This chapter introduces the fundamental algorithms of **conversational agents**, or **dialog systems**. These programs communicate with users in natural language (text, speech, or even both), and generally fall into two classes.

Task-oriented dialog agents are designed for a particular task and set up to have short conversations (from as little as a single interaction to perhaps half-a-dozen interactions) to get information from the user to help complete the task. These include the digital assistants that are now on every cellphone or on home controllers (Siri, Cortana, Alexa, Google Now/Home, etc.) whose dialog agents can give travel directions, control home appliances, find restaurants, or help make phone calls or send texts. Companies deploy goal-based conversational agents on their websites to help customers answer questions or address problems. Conversational agents play an important role as an interface to robots. And they even have applications for social good. DoNotPay is a "robot lawyer" that helps people challenge incorrect parking fines, apply for emergency housing, or claim asylum if they are refugees.

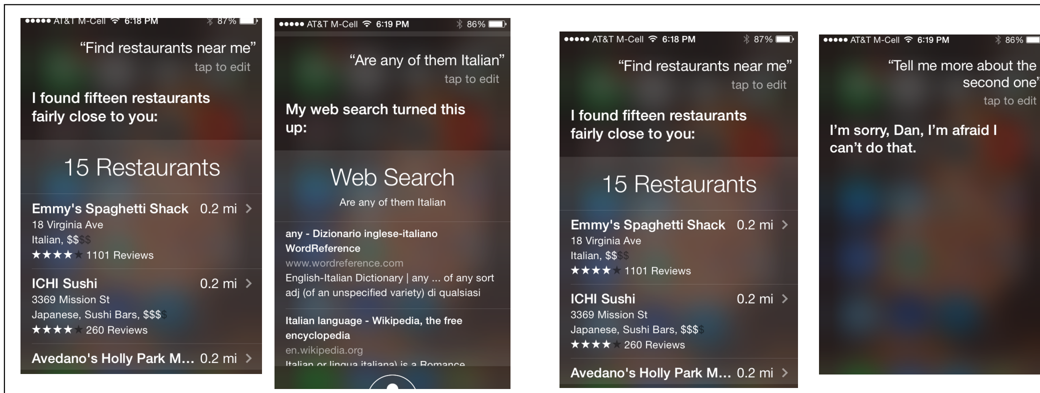
Chatbots are systems designed for extended conversations, set up to mimic the

unstructured conversational or 'chats' characteristic of human-human interaction, rather than focused on a particular task like booking plane flights. These systems often have an entertainment value, such as Microsoft's XiaoIce (Little Bing 小冰) system (Microsoft, 2014), which chats with people on text messaging platforms. Chatbots are also often attempts to pass various forms of the Turing test (introduced in Chapter 1). Yet starting from the very first system, ELIZA (Weizenbaum, 1966), chatbots have also been used for practical purposes, such as testing theories of psychological counseling.

Note that the word 'chatbot' is often used in the media and in industry as a synonym for conversational agent. In this chapter we will instead follow the usage in the natural language processing community, limiting the designation *chatbot* to this second subclass of systems designed for extended, casual conversation.

turn

Let's see some examples of dialog systems. One dimension of difference across systems is how many **turns** they can deal with. A dialog consists of multiple turns, each a single contribution to the dialog (the terminology is as if dialog is a game in which I take a turn, then you take a turn, then me, and so on). A turn can consist of a sentence, although it might be as short as a single word or as long as multiple sentences. The simplest such systems generally handle a single turn from the user, acting more like question-answering or command-and-control systems. This is especially common with digital assistants. For example Fig. 25.1 shows screen captures from an early version of Apple's Siri personal assistant from 2014, demonstrating this kind of single-query behavior.



(a) (b)

Figure 25.1 Two sets of interactions with Siri in 2014. (a) A question ("Find restaurants near me") returns restaurants, but the system was unable to interpret a follow-up question ("Are any of them Italian?"). (b) An alternative followup ("Tell me more about the second one") similarly fails. This early system's confusion at follow-up questions suggests that it is mainly designed for a single interaction.

By contrast, Fig. 25.2 shows that a 2017 version of the Siri digital assistant can handle slightly longer dialogs, handling a second turn with a follow-up question.

While spoken dialogs with mobile phone digital assistants tend to be short, some tasks do require longer dialogs. One such task is travel planning and management, a key concern of dialog systems since the very influential GUS system for planning airline travel (Bobrow et al., 1977); we'll see an example in the next section.

Dialog systems can even be used for much more complex domains like automatic tutoring. Figure 25.3 shows part of a dialog from the adaptive ITSPOKE dialog system (Forbes-Riley and Litman, 2011). In this example the system detects the hesitancy of the student's first response ("Is it 19.6 m/s?"), and, even though the

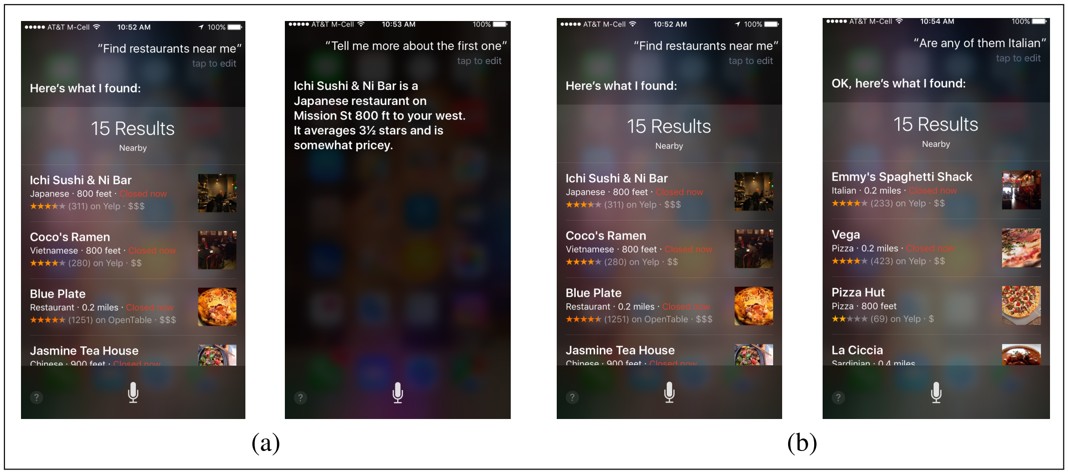


Figure 25.2 The same follow-up questions that Siri couldn't answer in 2014 receive appropriate responses when posed to Siri in 2017.

answer is correct, decides to explain the answer and ask a follow-up question before moving on.

Tutor:	What will the velocity of the object be a second after that (where the initial velocity is 9.8 m/s and the acceleration is 9.8 m/s^2)?
Student:	Is it 19.6 m/s?
Tutor:	Good. A second later, its velocity will be 9.8m/s + 9/8 m/s = 19.6 m/s This is because its acceleration tells us that every second, its velocity increases by 9.8 m/s. So what'll its velocity be, a second after that?
Student:	Um. I'm thinking. 29.4 m/s
Tutor:	Right. Now back to the man and his keys. We've shown that they both have an acceleration of 9.8 m/s^s. So, how do their velocities compare with each other, at every second during the fall?

Figure 25.3 The tutoring domain: An excerpt from a tutorial interaction between a physics student and the ITSPOKE system of Forbes-Riley and Litman (2011) based on the Why2-Atlas text-based tutoring system (VanLehn et al., 2002).

Finally, conversational agents can be purely for fun, like the agents designed for simple chit-chat like Cleverbot, an IR-based chatbot capable of carrying on the kinds of conversations shown in Fig. 25.4.

User:	Do you like Doctor Who?
Cleverbot:	Yes, so funny.
User:	What's your favorite show?
Cleverbot:	I don't have one.
User:	What's your name?
Cleverbot:	Jason I already told you.
User:	No your real name.
Cleverbot:	Stephanie IS my real name.
User:	I thought it was Jason?
Cleverbot:	How did you come to that conclusion.
User:	Because you said so.
Cleverbot:	I have to do my homework.

Figure 25.4 Chatbots: A conversation between a user and Cleverbot, an IR-based chatbot (Carpenter, 2017).

We begin in the next section with chatbots, introducing the three major chatbot architectures: rule-based systems, information retrieval systems, and transduction models. We then turn to task-oriented dialog, introducing the frame-based architecture (the **GUS** architecture) that underlies most modern task-based dialog systems, the VoiceXML system that is a simple industrial instantiation of the architecture, and information on designing and evaluating these systems.

25.1 Chatbots

chatbot

Chatbots are systems that can carry on extended conversations with the goal of mimicking the unstructured conversational or 'chats' characteristic of human-human interaction,

These systems often have an entertainment value, such as Microsoft's XiaoIce (Little Bing 小冰) system (Microsoft, 2014), which chats with people on text messaging platforms. Yet starting from the very first system, ELIZA (Weizenbaum, 1966), chatbots have also been used for practical purposes, such as testing theories of psychological counseling.

Like practically everything else in language processing, chatbot architectures fall into two classes: rule-based systems and corpus-based systems. Rule-based systems include the early influential ELIZA and PARRY systems. Corpus-based systems mine large datasets of human-human conversations, which can be done by using information retrieval (IR-based systems simply copy a human's response from a previous conversation) or by using a machine translation paradigm such as neural network sequence-to-sequence systems, to learn to map from a user utterance to a system response.

25.1.1 Rule-based chatbots: ELIZA and PARRY

ELIZA (Weizenbaum, 1966) is the most important chatbot dialog system in the history of the field, and so we take some time here to describe its algorithm in detail. ELIZA was designed to simulate a Rogerian psychologist, based on a branch of clinical psychology whose methods involve drawing the patient out by reflecting patient's statements back at them. Rogerian psychology is the rare type of conversation in which, as Weizenbaum points out, one can "assume the pose of knowing almost nothing of the real world". If a patient says "I went for a long boat ride" and the psychiatrist says "Tell me about boats", you don't assume she didn't know what a boat is, but rather assume she had some conversational goal. Most chatbots trying to pass Turing test choose a domain with similar properties.

Let's look at the most famous ELIZA conversation, excerpted (deleting some material with ...) from a full interaction given in the original paper:

```
Men are all alike.
IN WHAT WAY
They're always bugging us about something or other.
CAN YOU THINK OF A SPECIFIC EXAMPLE
Well  my boyfriend made me come here.
YOUR BOYFRIEND MADE YOU COME HERE
He says I'm depressed much of the time.
I AM SORRY TO HEAR YOU ARE DEPRESSED
..
```

```
..
WHO ELSE IN YOUR FAMILY TAKES CARE OF YOU?
My father
YOUR FATHER
You are like my father in some ways.
WHAT RESEMBLANCE DO YOU SEE
You are not very aggressive but I think you don't want me to notice that.
WHAT MAKES YOU THINK I AM NOT AGGRESSIVE
You don't argue with me.
WHY DO YOU THINK I DON'T ARGUE WITH YOU
You are afraid of me.
DOES IT PLEASE YOU TO BELIEVE I'M AFRAID OF YOU
My father is afraid of everybody.
WHAT ELSE COMES TO MIND WHEN YOU THINK OF YOUR  FATHER
Bullies.
DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT YOUR  BOYFRIEND MADE YOU COME HERE
```

As we summarized briefly in Chapter 1, ELIZA worked by pattern/transform rules like the following one:

```
(0 YOU 0 ME) [pattern]
->
(WHAT MAKES YOU THINK I 3 YOU) [transform]
```

In the ELIZA pattern, 0 means Kleene*, and in the transform rules the numbers are the index of the constituent in the pattern. Thus the number 3 refers to the second 0 in the first pattern. This rule would transfer

```
You hate me
```

into:

```
WHAT MAKES YOU THINK I HATE YOU
```

Each ELIZA pattern/rule is linked to a keyword that might occur in a user sentence.

The architecture is summarized in Fig. 25.5.

Keywords are associated with a rank, with specific words being more highly ranked, and more general words ranking lower. Consider the following user sentence:

```
I know everybody laughed at me
```

Because it has the word "I", this sentence could match the following rule whose keyword is *I*:

```
(I *) -> (You say you 2)
```

producing:

```
YOU SAY YOU KNOW EVERYBODY LAUGHED AT YOU
```

But as Weizenbaum points out, "I" is a very general word and its keywords lead to very general responses. Instead the keyword "everybody" is much more interesting, since someone using universals like everybody or always is probably "referring to some quite specific event or person". Therefore, ELIZA prefers to respond with the pattern associated with the more specific keyword *everybody* (implementing by just assigning "everybody" rank 5 and "I" rank 0 in the lexicon), whose rule thus transforms the sentence to:

```
function ELIZA GENERATOR(user sentence) returns response

Find the word w in sentence that has the highest keyword rank
  if w exists
      Choose the highest ranked rule r for w that matches sentence
      response ← Apply the transform in r to sentence
      if w = 'my'
         future ← Apply a transformation from the 'memory' rule list to sentence
         Push future onto memory stack
  else (no keyword applies)
    either
       response ← Apply the transform for the NONE keyword to sentence
    or
       response ← Pop the top response from the memory stack
  return(response)
```

Figure 25.5 A simplified sketch of the ELIZA algorithm. The power of the algorithm comes from the particular transforms associated with each keyword.

```
WHO IN PARTICULAR ARE YOU THINKING OF?
```

If no keyword matches, ELIZA chooses a non-commital response like "PLEASE GO ON", "THAT'S VERY INTERESTING", or "I SEE".

Finally, ELIZA has a clever memory trick that accounts for the very last sentence of the conversation above. Whenever the word "my" is the highest ranked keyword, ELIZA will randomly select a transform on the MEMORY list, apply it to the sentence, and store it on the stack:

```
(MEMORY MY
  (0 MY 0 = LETS DISCUSS FURTHER WHY YOUR 3)
  (0 MY 0 = EARLIER YOU SAID YOUR 3)
  (0 MY 0 = DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT YOUR 3
```

Later, if no keyword matches a sentence, ELIZA will return the top of the MEMORY queue instead. [1]

People became deeply emotionally involved with the program. Weizenbaum tells the story of one of his staff who would ask Weizenbaum to leave the room when she talked with ELIZA. When Weizenbaum suggested that he might want to store all the ELIZA conversations for later analysis, people immediately pointed out the privacy implications, which suggested that they were having quite private conversations with ELIZA, despite knowing that it was just software.

Eliza's framework is still used today; modern chatbot system tools like ALICE are based on updated versions of ELIZA's pattern/action architecture.

A few years after ELIZA, another chatbot with a clinical psychology focus, PARRY (Colby et al., 1971), was used to study schizophrenia. In addition to ELIZA-like regular expressions, the PARRY system including a model of its own mental state, with affect variables for the agent's levels of fear and anger; certain topics of conversation might lead PARRY to become more angry or mistrustful. If PARRY's **anger** variable is high, he will choose from a set of "hostile" outputs. If the input mentions his delusion topic, he will increase the value of his **fear** variable and then begin to express the sequence of statements related to his delusion. Parry was the

[1] Fun fact: because of its structure as a queue, this MEMORY trick is the earliest known hierarchical model of discourse in natural language processing.

first known system to pass the Turing test (in 1972!); psychiatrists couldn't distinguish text transcripts of interviews with PARRY from transcripts of interviews with real paranoids (Colby et al., 1972).

25.1.2 Corpus-based chatbots

Corpus-based chatbots, instead of using hand-built rules, mine conversations of human-human conversations, or sometimes mine the human responses from human-machine conversations. Serban et al. (2017) summarizes some such available corpora, such as conversations on chat platforms, on Twitter, or in movie dialog, which is available in great quantities and has been shown to resemble natural conversation (Forchini, 2013). Chatbot responses can even be extracted from sentences in corpora of non-dialog text.

There are two common architectures for corpus-based chatbots: information retrieval, and machine learned sequence transduction. Like rule-based chatbots (but unlike frame-based dialog systems), most corpus-based chatbots do very little modeling of the conversational context. Instead they focus on generating a single response turn that is appropriate given the user's immediately previous utterance. For this reason they are often called **response generation** systems. Corpus-based chatbots thus have some similarity to question answering systems, which focus on single responses while ignoring context or larger conversational goals.

response generation

IR-based chatbots

The principle behind information retrieval based chatbots is to respond to a user's turn X by repeating some appropriate turn Y from a corpus of natural (human) text. The differences across such systems lie in how they choose the corpus, and how they decide what counts as an appropriate human turn to copy.

A common choice of corpus is to collect databases of human conversations. These can come from microblogging platforms like Twitter or any Weibo (微博). Another approach is to use corpora of movie dialog. Once a chatbot has been put into practice, the turns that humans use to respond to the chatbot can be used as additional conversational data for training.

Given the corpus and the user's sentence, IR-based systems can use any retrieval algorithm to choose an appropriate response from the corpus. The two simplest methods are the following:

1. Return the response to the most similar turn: Given user query q and a conversational corpus C, find the turn t in C that is most similar to q (for example has the highest cosine with q) and return the *following* turn, i.e. the human response to t in C:

$$r = response\left(\underset{t \in C}{\operatorname{argmax}} \frac{q^T t}{||q||t||}\right) \tag{25.1}$$

The idea is that we should look for a turn that most resembles the user's turn, and return the human response to that turn (Jafarpour et al. 2009, Leuski and Traum 2011).

2. Return the most similar turn: Given user query q and a conversational corpus C, return the turn t in C that is most similar to q (for example has the highest cosine with q):

$$r = \underset{t \in C}{\operatorname{argmax}} \frac{q^T t}{||q||t||} \tag{25.2}$$

The idea here is to directly match the users query q with turns from C, since a good response will often share words or semantics with the prior turn.

In each case, any similarity function can be used, most commonly cosines computed either over words (using tf-idf) or over embeddings.

Although returning the *response* to the most similar turn seems like a more intuitive algorithm, returning the most similar turn seems to work better in practice, perhaps because selecting the response adds another layer of indirection that can allow for more noise (Ritter et al. 2011, Wang et al. 2013).

The IR-based approach can be extended by using more features than just the words in the q (such as words in prior turns, or information about the user), and using any full IR ranking approach. Commercial implementations of the IR-based approach include Cleverbot (Carpenter, 2017) and Microsoft's XiaoIce (Little Bing 小冰) system (Microsoft, 2014).

Instead of just using corpora of conversation, the IR-based approach can be used to draw responses from narrative (non-dialog) text. For example, the pioneering COBOT chatbot (Isbell et al., 2000) generated responses by selecting sentences from a corpus that combined the Unabomber Manifesto by Theodore Kaczynski, articles on alien abduction, the scripts of "The Big Lebowski" and "Planet of the Apes". Chatbots that want to generate informative turns such as answers to user questions can use texts like Wikipedia to draw on sentences that might contain those answers (Yan et al., 2016).

Sequence to sequence chatbots

An alternate way to use a corpus to generate dialog is to think of response generation as a task of *transducing* from the user's prior turn to the system's turn. This is basically the machine learning version of Eliza; the system learns from a corpus to transduce a question to an answer.

This idea was first developed by using phrase-based machine translation (Ritter et al., 2011) to translate a user turn to a system response. It quickly became clear, however, that the task of response generation was too different from machine translation. In machine translation words or phrases in the source and target sentences tend to align well with each other; but in conversation, a user utterance may share no words or phrases with a coherent response.

Instead, (roughly contemporaneously by Shang et al. 2015, Vinyals and Le 2015, and Sordoni et al. 2015) transduction models for response generation were modeled instead using encoder-decoder (seq2seq) models (Chapter 23), as shown in Fig. 25.6.

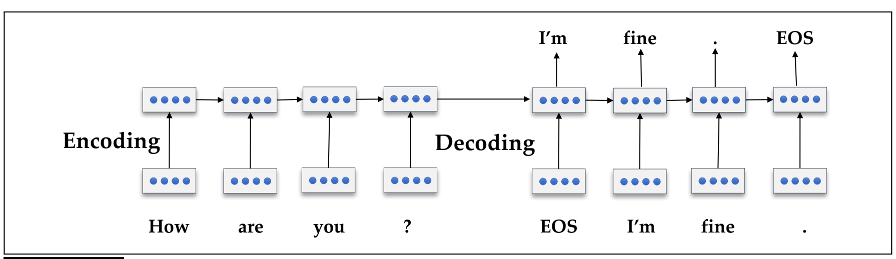


Figure 25.6 A sequence to sequence model for neural response generation in dialog.

A number of modifications are required to the basic seq2seq model to adapt it for the task of response generation. For example basic seq2seq models have a tendency to produce predictable but repetitive and therefore dull responses like "I'm OK" or "I don't know" that shut down the conversation. This can be addressed by changing the objective function for seq2seq model training to a mutual information objective, or by modifying a beam decoder to keep more diverse responses in the beam (Li

et al., 2016a).

Another problem with the simple SEQ2SEQresponse generation architecture is its inability to model the longer prior context of the conversation. This can be done by allowing the model to see prior turns, such as by using a hierarchical model that summarizes information over multiple prior turns (Lowe et al., 2017b).

Finally, SEQ2SEQresponse generators focus on generating single responses, and so don't tend to do a good job of continuously generating responses that cohere across multiple turns. This can be addressed by using reinforcement learning, as well as techniques like adversarial networks, to learn to choose responses that make the overall conversation more natural (Li et al. 2016b, Li et al. 2017).

Fig. 25.7 shows some sample responses generated by a vanilla SEQ2SEQmodel, and from a model trained by an adversarial algorithm to produce responses that are harder to distinguish from human responses (Li et al., 2017).

Input	tell me ... how long have you had this falling sickness ?
Vanilla-SEQ2SEQ	i'm not a doctor.
Adversarial	a few months, i guess .
Input	so i had the doctors test sammy 's response to conditioning .
Vanilla-SEQ2SEQ	sammy wrote the test sammy wrote the test .
Adversarial	so he took the pills .
Input	they didn 't have much success with this problem commander .
Vanilla-SEQ2SEQ	they 're not the only ones who have been in the system .
Adversarial	can we find someone else ?

Figure 25.7 Sample responses generated by a SEQ2SEQmodel trained either with a vanilla maximum likelihood objective, or adversarially trained to produce sentences that are hard for an adversary to distinguish from human sentences (Li et al., 2017).

Evaluating Chatbots

Chatbots are generally evaluated by humans. The slot-filling evaluations used for task-based dialog (Section 25.2.3) aren't appropriate for this task (Artstein et al., 2009), and word-overlap metrics like BLEU for comparing a chatbot's response to a human response turn out to correlate very poorly with human judgments (Liu et al., 2016). BLEU performs poorly because there are so many possible responses to any given turn; word-overlap metrics work best when the space of responses is small and lexically overlapping, as is the case in machine translation.

While human evaluation is therefore required for evaluating chatbots, there are beginning to be models for automatic evaluation. The ADEM (Lowe et al., 2017a) classifier is trained on a set of responses labeled by humans with how appropriate they are, and learns to predict this label from the dialog context and the words in the system response.

adversarial evaluation

Another paradigm is **adversarial evaluation** (Bowman et al. 2016, Kannan and Vinyals 2016, Li et al. 2017), inspired by the Turing test. The idea is to train a "Turing-like" evaluator classifier to distinguish between human-generated responses and machine-generated responses. The more successful a response generation system is at fooling this evaluator, the better the system.

25.2 Frame Based Dialog Agents

domain ontology

Modern task-based dialog systems are based on a **domain ontology**, a knowledge

structure representing the kinds of intentions the system can extract from user sentences. The ontology defines one or more **frames**, each a collection of **slots**, and defines the **values** that each slot can take. This frame-based architecture was first introduced in 1977 in the influential GUS system for travel planning (Bobrow et al., 1977), and has been astonishingly long-lived, underlying most modern commercial digital assistants. Figure 25.8 shows a conversation with the original GUS system.

frame

slot

GUS:	Hello. My name is GUS. I can help you plan a simple trip by air. Where do you want to go ?
Client:	I want to go to San Diego on May 28
GUS:	What time do you want to leave?
Client:	I must be in San Diego before 10 am
GUS:	Would you like P.S.A. flight 102 that arrives at 9.15 am ?
Client:	What is the next flight ?
GUS:	Air California flight 310 that leaves at 8.30am and arrives at 10.00am
Client:	I'll take the first one
GUS:	In what name should I make a reservation, please?
Client:	Dan Bobrow
GUS:	I have confirmed the following flight: P.S.A. flight 102 on Wednesday May 28 from San Jose to San Diego that leaves at 7.30 am and arrives at 9.15 am.
GUS:	What date do you want to return on ?
Client:	On Friday in the evening.
GUS:	Would you like the flight that leaves at 7.45 pm ?
Client:	That's fine.
GUS:	I have confirmed the following flight: P.S.A. flight 307 on Friday May 30 from San Diego to San Jose that leaves at 7.45 pm and arrives at 9.30 pm Thank you for calling. Goodbye

Figure 25.8 The travel domain: A transcript of an actual dialog with the GUS system of Bobrow et al. (1977). P.S.A. and Air California were airlines of that period.

The set of slots in a GUS-style frame specifies what the system needs to know, and the filler of each slot is constrained to values of a particular semantic type. In the travel domain, for example, a slot might be of type city (hence take on values like *San Francisco*, or *Hong Kong*) or of type date, airline, or time:

Slot	Type
ORIGIN CITY	city
DESTINATION CITY	city
DEPARTURE TIME	time
DEPARTURE DATE	date
ARRIVAL TIME	time
ARRIVAL DATE	date

Types in GUS, as in modern frame-based dialog agents, may have hierarchical structure; for example the *date* type in GUS is itself a frame with slots with types like *integer* or members of sets of weekday names:

```
DATE
  MONTH NAME
  DAY (BOUNDED-INTEGER 1 31)
  YEAR INTEGER
  WEEKDAY (MEMBER (SUNDAY MONDAY TUESDAY WEDNESDAY THURSDAY FRIDAY SATURDAY))
```

25.2.1 Control structure for frame-based dialog

The control architecture of frame-based dialog systems is designed around the frame. The goal is to fill the slots in the frame with the fillers the user intends, and then perform the relevant action for the user (answering a question, or booking a flight). Most frame-based dialog systems are based on finite-state automata that are hand-designed for the task by a dialog designer.

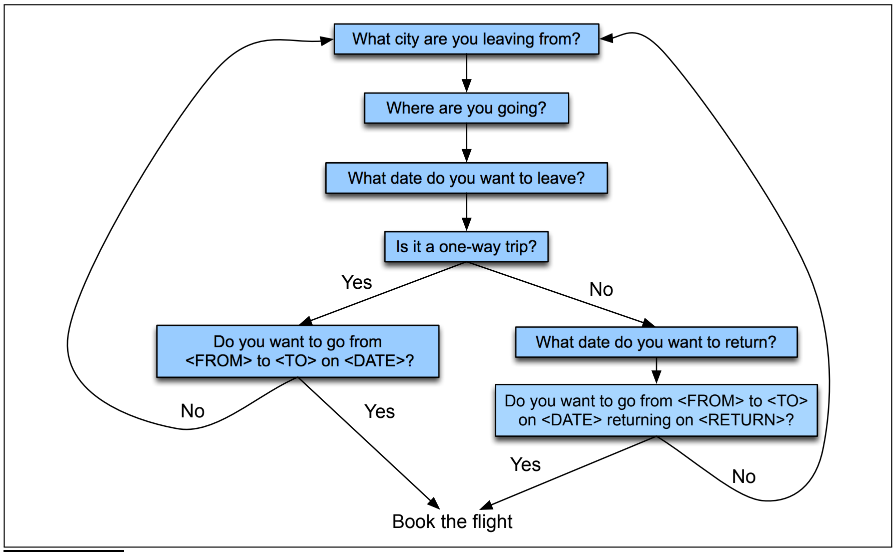


Figure 25.9 A simple finite-state automaton architecture for frame-based dialog.

Consider the very simple finite-state control architecture shown in Fig. 25.9, implementing a trivial airline travel system whose job is to ask the user for the information for 4 slots: departure city, a destination city, a time, and whether the trip is one-way or round-trip. Let's first associate with each slot a question to ask the user:

Slot	Question
ORIGIN CITY	"From what city are you leaving?"
DESTINATION CITY	"Where are you going?"
DEPARTURE TIME	"When would you like to leave?"
ARRIVAL TIME	"When do you want to arrive?"

Figure 25.9 shows a sample dialog manager for such a system. The states of the FSA correspond to the slot questions, user, and the arcs correspond to actions to take depending on what the user responds. This system completely controls the conversation with the user. It asks the user a series of questions, ignoring (or misinterpreting) anything that is not a direct answer to the question and then going on to the next question.

initiative The speaker in control of any conversation is said to have the **initiative** in the conversation. Systems that completely control the conversation in this way are thus called **system-initiative**. By contrast, in normal human-human dialog, initiative shifts back and forth between the participants (Bobrow et al. 1977, Walker and Whittaker 1990).

system-initiative

The single-initiative finite-state dialog architecture has the advantage that the system always knows what question the user is answering. This means the system can prepare the speech recognizer with a language model tuned to answers for this

question, and also makes natural language understanding easier. Most finite-state systems also allow **universal** commands that can be said anywhere in the dialog, like **help**, to give a help message, and **start over** (or **main menu**), which returns the user to some specified main start state,. Nonetheless such a simplistic finite-state architecture is generally applied only to simple tasks such as entering a credit card number, or a name and password.

universal

For most applications, users need a bit more flexibility. In a travel-planning situation, for example, a user may say a sentence that fills multiple slots at once:

(25.3) I want a flight from San Francisco to Denver one way leaving after five p.m. on Tuesday.

Or in cases where there are multiple frames, a user may say something to shift frames, for example from airline reservations to reserving a rental car:

(25.4) I'd like to book a rental car when I arrive at the airport.

The standard GUS architecture for frame-based dialog systems, used in various forms in modern systems like Apple's Siri, Amazon's Alexa, and the Google Assistant, therefore follows the frame in a more flexible way. The system asks questions of the user, filling any slot that the user specifies, even if a user's response fills multiple slots or doesn't answer the question asked. The system simply skips questions associated with slots that are already filled. Slots may thus be filled out of sequence. The GUS architecture is thus a kind of **mixed initiative**, since the user can take at least a bit of conversational initiative in choosing what to talk about.

mixed initiative

The GUS architecture also has condition-action rules attached to slots. For example, a rule attached to the DESTINATION slot for the plane booking frame, once the user has specified the destination, might automatically enter that city as the default *StayLocation* for the related hotel booking frame.

Once the system has enough information it performs the necessary action (like querying a database of flights) and returns the result to the user.

We mentioned in passing the linked airplane and travel frames. Many domains, of which travel is one, require the ability to deal with multiple frames. Besides frames for car or hotel reservations, we might need frames with general route information (for questions like *Which airlines fly from Boston to San Francisco?*), information about airfare practices (for questions like *Do I have to stay a specific number of days to get a decent airfare?*).

In addition, once we have given the user options (such as a list of restaurants), we can even have a special frame for 'asking questions about this list', whose slot is the particular restaurant the user is asking for more information about, allowing the user to say 'the second one' or 'the Italian one'.

Since users may switch from frame to frame, the system must be able to disambiguate which slot of which frame a given input is supposed to fill and then switch dialog control to that frame.

Because of this need to dynamically switch control, the GUS architecture is a **production rule** system. Different types of inputs cause different productions to fire, each of which can flexibly fill in different frames. The production rules can then switch control according to factors such as the user's input and some simple dialog history like the last question that the system asked.

Commercial dialog systems provide convenient interfaces or libraries to make it easy to build systems with these kinds of finite-state or production rule systems, for example providing graphical interfaces to allow dialog modules to be chained together.

25.2.2 Natural language understanding for filling slots

domain classification

intent determination

slot filling

The goal of the natural language understanding component is to extract three things from the user's utterance. The first task is **domain classification**: is this user for example talking about airlines, programming an alarm clock, or dealing with their calendar? Of course this 1-of-n classification tasks is unnecessary for single-domain systems that are focused on, say, only calendar management, but multi-domain dialog systems are the modern standard. The second is user **intent determination**: what general task or goal is the user trying to accomplish? For example the task could be to Find a Movie, or Show a Flight, or Remove a Calendar Appointment. Finally, we need to do **slot filling**: extract the particular slots and fillers that the user intends the system to understand from their utterance with respect to their intent. From a user utterance like this one:

```
Show me morning flights from Boston to San Francisco on Tuesday
```

a system might want to build a representation like:

```
DOMAIN:       AIR-TRAVEL
INTENT:       SHOW-FLIGHTS
ORIGIN-CITY:  Boston
ORIGIN-DATE:  Tuesday
ORIGIN-TIME:  morning
DEST-CITY:    San Francisco
```

while an utterance like

```
Wake me tomorrow at 6
```

should give an intent like this:

```
DOMAIN:  ALARM-CLOCK
INTENT:  SET-ALARM
TIME:    2017-07-01 0600-0800
```

The task of slot-filling, and the simpler tasks of domain and intent classification, are special cases of the task of semantic parsing discussed in Chapter 16. Dialog agents can thus extract slots, domains, and intents from user utterances by applying any of the semantic parsing approaches discussed in that chapter.

The method used in the original GUS system, and still quite common in industrial applications, is to use hand-written rules, often as part of the condition-action rules attached to slots or concepts.

For example we might just define a regular expression consisting of a set strings that map to the SET-ALARM intent:

```
wake me (up) | set (the|an) alarm | get me up
```

We can build more complex automata that instantiate sets of rules like those discussed in Chapter 17, for example extracting a slot filler by turning a string like `Monday at 2pm` into an object of type `date` with parameters (DAY, MONTH, YEAR, HOURS, MINUTES).

semantic grammar

Rule-based systems can be even implemented with full grammars. Research systems like the Phoenix system (Ward and Issar, 1994) consists of large hand-designed **semantic grammars** with thousands of rules. A semantic grammar is a context-free grammar in which the left-hand side of each rule corresponds to the semantic entities being expressed (i.e., the slot names) as in the following fragment:

SHOW	→	show me | i want | can i see|...
DEPART_TIME_RANGE	→	(after|around|before) HOUR | morning | afternoon | evening
HOUR	→	one|two|three|four...|twelve (AMPM)
FLIGHTS	→	(a) flight | flights
AMPM	→	am | pm
ORIGIN	→	from CITY
DESTINATION	→	to CITY
CITY	→	Boston | San Francisco | Denver | Washington

Semantic grammars can be parsed by any CFG parsing algorithm (see Chapter 11), resulting in a hierarchical labeling of the input string with semantic node labels, as shown in Fig. 25.10.

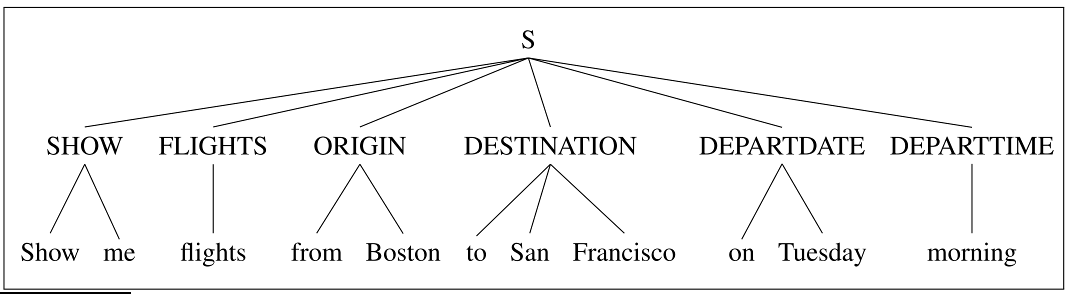


Figure 25.10 A semantic grammar parse for a user sentence, using slot names as the internal parse tree nodes.

Whether regular expressions or parsers are used, it remains only to put the fillers into some sort of canonical form, for example by normalizing dates as discussed in Chapter 17.

A number of tricky issues have to be dealt with. One important issue is negation; if a user specifies that they “can’t fly Tuesday morning”, or want a meeting ”any time except Tuesday morning”, a simple system will often incorrectly extract “Tuesday morning” as a user goal, rather than as a negative constraint.

N-best list

Speech recognition errors must also be dealt with. One common trick is to make use of the fact that speech recognizers often return a ranked **N-best list** of hypothesized transcriptions rather than just a single candidate transcription. The regular expressions or parsers can simply be run on every sentence in the N-best list, and any patterns extracted from any hypothesis can be used.

As we saw earlier in discussing information extraction, the rule-based approach is very common in industrial applications. It has the advantage of high precision, and if the domain is narrow enough and experts are available, can provide sufficient coverage as well. On the other hand, the hand-written rules or grammars can be both expensive and slow to create, and hand-written rules can suffer from recall problems.

A common alternative is to use supervised machine learning. Assuming a training set is available which associates each sentence with the correct semantics, we can train a classifier to map from sentences to intents and domains, and a sequence model to map from sentences to slot fillers.

For example given the sentence:

```
I want to fly to San   Francisco on Monday    afternoon  please
```

we might first apply a simple 1-of-N classifier (logistic regression, neural network, etc.) that uses features of the sentence like word N-grams to determine that the domain is AIRLINE and and the intent is SHOWFLIGHT.

Next to do slot filling we might first apply a classifier that uses similar features of the sentence to predict which slot the user wants to fill. Here in addition to

word unigram, bigram, and trigram features we might use named entity features or features indicating that a word is in a particular lexicon (such as a list of cities, or airports, or days of the week) and the classifer would return a slot name (in this case DESTINATION, DEPARTURE-DAY, and DEPARTURE-TIME). A second classifier can then be used to determine the filler of the named slot, for example a city classifier that uses N-grams and lexicon features to determine that the filler of the DESTINATION slot is SAN FRANCISCO.

An alternative is to use a sequence model (MEMMs, CRFs, RNNs) to directly assign a slot label to each word in the sequence, following the method used for other information extraction models in Chapter 17 (Pieraccini et al. 1991, Raymond and Riccardi 2007, Mesnil et al. 2015, Hakkani-Tür et al. 2016). Once again we would need a supervised training test, with sentences paired with sequences of **IOB** labels like the following:

IOB

```
O O    O  O   O  B-DES I-DES     O  B-DEPTIME I-DEPTIME  O
I want to fly to San   Francisco on Monday    afternoon please
```

Recall from Chapter 17 that in IOB tagging we introduce a tag for the beginning (B) and inside (I) of each slot label, and one for tokens outside (O) any slot label. The number of tags is thus $2n+1$ tags, where n is the number of slots.

Any IOB tagger sequence model can then be trained on a training set of such labels. Feature-based sequence models (MEMM, CRF) make use of features like word embeddings, word unigrams and bigrams, lexicons (for example lists of city names), and slot transition features (perhaps DESTINATION is more likely to follow ORIGIN than the other way around) to map a user's utterance to the slots. An MEMM (Chapter 8) for example, combines these features of the input word w_i, its neighbors within l words w_{i-l}^{i+l}, and the previous k slot tags s_{i-k}^{i-1} to compute the most likely slot label sequence S from the word sequence W as follows:

$$
\begin{aligned}
\hat{S} &= \operatorname*{argmax}_S P(S|W) \\
&= \operatorname*{argmax}_S \prod_i P(s_i|w_{i-l}^{i+l}, s_{i-k}^{i-1}) \\
&= \operatorname*{argmax}_S \prod_i \frac{\exp\left(\sum_i w_i f_i(s_i, w_{i-l}^{i+l}, s_{i-k}^{i-1})\right)}{\sum_{s' \in \text{slotset}} \exp\left(\sum_i w_i f_i(s', w_{i-l}^{i+l}, t_{i-k}^{i-1})\right)}
\end{aligned}
\tag{25.5}
$$

The Viterbi algorithm is used to decode the best slot sequence $\hat{S}$.

Neural network architectures mostly eschew the feature extraction step, instead using the bi-LSTM architecture introduced in Chapter 9, and applied to IOB-style named entity tagging in Chapter 17. A typical LSTM-style architecture is shown in Fig. 25.11. Here the input is a series of words $w_1...w_n$, and the output is a series of IOB tags $s_1...s_n$. In the architecture as introduced in Chapter 17, the input words are converted into two embeddings: standard word2vec or GloVe embeddings, and a character-based embedding, which are concatenated together and passed through a bi-LSTM. The output of the bi-LSTM can be passed to a softmax choosing an IOB tag for each input word, or to a CRF layer which uses Viterbi to find the best series of IOB tags. In addition, neural systems can combine the domain-classification and intent-extraction tasks with slot-filling simply by adding a domain concatenated with an intent as the desired output for the final EOS token.

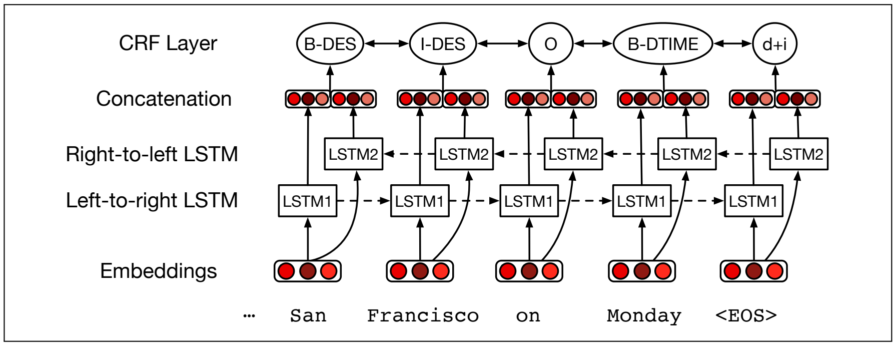


Figure 25.11 An LSTM architecture for slot filling, mapping the words in the input to a series of IOB tags plus a final state consisting of a domain concatenated with an intent.

Once the sequence labeler has tagged the user utterance, a filler string can be extracted for each slot from the tags (e.g., "San Francisco"), and these word strings can then be normalized to the correct form in the ontology (perhaps the airport code'SFO'). This normalization can take place by using homonym dictionaries (specifying, for example, that SF, SFO, and San Francisco are the same place).

In industrial contexts, machine learning-based systems for slot-filling are often bootstrapped from rule-based systems in a semi-supervised learning manner. A rule-based system is first built for the domain, and a test-set is carefully labeled. As new user utterances come in, they are paired with the labeling provided by the rule-based system to create training tuples. A classifier can then be trained on these tuples, using the test-set to test the performance of the classifier against the rule-based system. Some heuristics can be used to eliminate errorful training tuples, with the goal of increasing precision. As sufficient training samples become available the resulting classifier can often outperform the original rule-based system (Suendermann et al., 2009), although rule-based systems may still remain higher-precision for dealing with complex cases like negation.

25.2.3 Evaluating Slot Filling

An intrinsic error metric for natural language understanding systems for slot filling is the Slot Error Rate for each sentence:

$$\text{Slot Error Rate for a Sentence} = \frac{\text{\# of inserted/deleted/subsituted slots}}{\text{\# of total reference slots for sentence}} \tag{25.6}$$

Consider a system faced with the following sentence:

(25.7) Make an appointment with Chris at 10:30 in Gates 104

which extracted the following candidate slot structure:

Slot	**Filler**
PERSON	Chris
TIME	11:30 a.m.
ROOM	Gates 104

Here the slot error rate is 1/3, since the TIME is wrong. Instead of error rate, slot precision, recall, and F-score can also be used.

A perhaps more important, although less fine-grained, measure of success is an extrinsic metric like **task error rate**. In this case, the task error rate would quantify how often the correct meeting was added to the calendar at the end of the interaction.

25.2.4 Other components of frame-based dialog

We've focused on the natural language understanding component that is the core of frame-based systems, but here we also briefly mention other modules.

The ASR (automatic speech recognition) component takes audio input from a phone or other device and outputs a transcribed string of words, as discussed in Chapter 27. Various aspects of the ASR system may be optimized specifically for use in conversational agents.

Because what the user says to the system is related to what the system has just said, language models in conversational agent depend on the dialog state. For example, if the system has just asked the user "What city are you departing from?", the ASR language model can be constrained to just model answers to that one question. This can be done by training an N-gram language model on answers to this question. Alternatively a finite-state or context-free grammar can be hand written to recognize only answers to this question, perhaps consisting only of city names or perhaps sentences of the form 'I want to (leave|depart) from [CITYNAME]'. Indeed, many simple commercial dialog systems use only non-probabilistic language models based on hand-written finite-state grammars that specify all possible responses that the system understands. We give an example of such a hand-written grammar for a VoiceXML system in Section 25.3.

restrictive grammar

A language model that is completely dependent on dialog state is called a **restrictive grammar**, and can be used to constrain the user to only respond to the system's last utterance. When the system wants to allow the user more options, it might mix this state-specific language model with a more general language model.

language generation

template-based generation

prompt

The **language generation** module of any dialog system produces the utterances that the system says to the user. Frame-based systems tend to use **template-based generation**, in which all or most of the words in the sentence to be uttered to the user are prespecified by the dialog designer. Sentences created by these templates are often called **prompts**. Templates might be completely fixed (like 'Hello, how can I help you?'), or can include some variables that are filled in by the generator, as in the following:

What time do you want to leave CITY-ORIG?
Will you return to CITY-ORIG from CITY-DEST?

These sentences are then passed to the TTS (text-to-speech) component (see Chapter 28). More sophisticated statistical generation strategies will be discussed in Section 26.5 of Chapter 26.

25.3 VoiceXML

There are many commercial systems that allow developers to implement frame-based dialog systems, including the user-definable skills in Amazon Alexa or the actions in Google Assistant. These systems provide libraries for defining the rules for detecting user intents and filling in slots, and for expressing the architecture for controlling which frames and actions the system should take at which times.

VoiceXML

Instead of focusing on a commercial engine, we introduce here a simple declarative formalism that has similar capabilities to each of them: **VoiceXML**, the Voice Extensible Markup Language (`http://www.voicexml.org/`), an XML-based dialog design language for creating simple frame-based dialogs. Although VoiceXML is simpler than a full commercial frame-based system (it's deterministic, and hence

only allows non-probabilistic grammar-based language models and rule-based semantic parsers), it's still a handy way to get a hands-on grasp of frame-based dialog system design.

A VoiceXML document contains a set of dialogs, each a **menu** or a **form**. A form is a frame, whose slots are called **fields**. The VoiceXML document in Fig. 25.12 shows three fields for specifying a flight's origin, destination, and date. Each field has a variable name (e.g., `origin`) that stores the user response, a **prompt**, (e.g., *Which city do you want to leave from*), and a grammar that is passed to the speech recognition engine to specify what is allowed to be recognized. The grammar for the first field in Fig. 25.12 allows the three phrases *san francisco*, *barcelona*, and *new york*. The VoiceXML interpreter walks through a form in document order, repeatedly selecting each item in the form, and each field in order.

prompt

```
<noinput>
I'm sorry, I didn't hear you.  <reprompt/>
</noinput>

<nomatch>
I'm sorry, I didn't understand that.  <reprompt/>
</nomatch>

<form>
  <block>    Welcome to the air travel consultant.  </block>
  <field name="origin">
      <prompt>     Which city do you want to leave from?  </prompt>
      <grammar type="application/x=nuance-gsl">
        [(san francisco) barcelona (new york)]
      </grammar>
      <filled>
         <prompt>  OK, from <value expr="origin"/>  </prompt>
      </filled>
  </field>
  <field name="destination">
      <prompt>  And which city do you want to go to?   </prompt>
      <grammar type="application/x=nuance-gsl">
        [(san francisco) barcelona (new york)]
      </grammar>
      <filled>
         <prompt>   OK, to <value expr="destination"/>   </prompt>
      </filled>
  </field>
  <field name="departdate" type="date">
      <prompt>  And what date do you want to leave?  </prompt>
      <filled>
         <prompt>   OK, on <value expr="departdate"/>   </prompt>
      </filled>
  </field>
  <block>
      <prompt> OK, I have you are departing from  <value expr="origin"/>
              to <value expr="destination"/> on <value expr="departdate"/>
      </prompt>
      send the info to book a flight...
  </block>
</form>
```

Figure 25.12 A VoiceXML script for a form with three fields, which confirms each field and handles the `noinput` and `nomatch` situations.

The prologue of the example shows two global defaults for error handling. If the user doesn't answer after a prompt (i.e., silence exceeds a timeout threshold), the VoiceXML interpreter will play the `<noinput>` prompt. If the user says something that doesn't match the grammar for that field, the VoiceXML interpreter will play the `<nomatch>` prompt. VoiceXML provides a `<reprompt/>` command, which repeats the prompt for whatever field caused the error.

The `<filled>` tag for a field is executed by the interpreter as soon as the field has been filled by the user. Here, this feature is used to confirm the user's input.

VoiceXML 2.0 specifies seven built-in grammar types: `boolean`, `currency`, `date`, `digits`, `number`, `phone`, and `time`. By specifying the `departdate` field as

type `date`, a date-specific language model will be passed to the speech recognizer.

```
<noinput>    I'm sorry, I didn't hear you.  <reprompt/>  </noinput>

<nomatch> I'm sorry, I didn't understand that.  <reprompt/> </nomatch>

<form>
   <grammar type="application/x=nuance-gsl">
    <![CDATA[
    Flight (  ?[
             (i [wanna (want to)] [fly go])
             (i'd like to [fly go])
             ([(i wanna)(i'd like a)] flight)
          ]
          [
            ( [from leaving departing] City:x) {<origin $x>}
            ( [(?going to)(arriving in)] City:x) {<destination $x>}
            ( [from leaving departing] City:x
              [(?going to)(arriving in)] City:y) {<origin $x> <destination $y>}
          ]
          ?please
     )
     City [ [(san francisco) (s f o)] {return( "san francisco, california")}
            [(denver) (d e n)] {return( "denver, colorado")}
            [(seattle) (s t x)] {return( "seattle, washington")}
     ]
     ]]> </grammar>

  <initial name="init">
     <prompt> Welcome to the consultant. What are your travel plans?  </prompt>
  </initial>

  <field name="origin">
     <prompt> Which city do you want to leave from?  </prompt>
     <filled>
        <prompt> OK, from <value expr="origin"/> </prompt>
     </filled>
  </field>
  <field name="destination">
     <prompt> And which city do you want to go to?  </prompt>
     <filled>
        <prompt> OK, to <value expr="destination"/> </prompt>
     </filled>
  </field>
  <block>
     <prompt> OK, I have you are departing from  <value expr="origin"/>
              to  <value expr="destination"/>.  </prompt>
    send the info to book a flight...
  </block>
</form>
```

Figure 25.13 A mixed-initiative VoiceXML dialog. The grammar allows sentences that specify the origin or destination cities or both. The user can respond to the initial prompt by specifying origin city, destination city, or both.

Figure 25.13 gives a mixed initiative example, allowing the user to answer questions in any order or even fill in multiple slots at once. The VoiceXML interpreter has a *guard condition* on fields, a test that keeps a field from being visited; the default test skips a field if its variable is already set.

Figure 25.13 also shows a more complex CFG grammar with two rewrite rules, `Flight` and `City`. The Nuance GSL grammar formalism uses parentheses () to mean concatenation and square brackets [] to mean disjunction. Thus, a rule like (25.8) means that `Wantsentence` can be expanded as `i want to fly` or `i want to go`, and `Airports` can be expanded as `san francisco` or `denver`.

(25.8)
```
Wantsentence (i want to [fly go])
Airports [(san francisco) denver]
```

VoiceXML grammars allow semantic attachments, such as the text string (`"denver, colorado"`) the return for the `City` rule, or a slot/filler , like the attachments for the `Flight` rule which fills the slot (`<origin>` or `<destination>` or both) with the value passed up in the variable *x* from the `City` rule.

TTS Performance	Was the system easy to understand ?
ASR Performance	Did the system understand what you said?
Task Ease	Was it easy to find the message/flight/train you wanted?
Interaction Pace	Was the pace of interaction with the system appropriate?
User Expertise	Did you know what you could say at each point?
System Response	How often was the system sluggish and slow to reply to you?
Expected Behavior	Did the system work the way you expected it to?
Future Use	Do you think you'd use the system in the future?

Figure 25.14 User satisfaction survey, adapted from Walker et al. (2001).

Because Fig. 25.13 is a mixed-initiative grammar, the grammar has to be applicable to any of the fields. This is done by making the expansion for `Flight` a disjunction; note that it allows the user to specify only the origin city, the destination city, or both.

25.4 Evaluating Dialog Systems

Evaluation is crucial in dialog system design. If the task is unambiguous, we can simply measure absolute task success (did the system book the right plane flight, or put the right event on the calendar).

To get a more fine-grained idea of user happiness, we can compute a *user satisfaction rating*, having users interact with a dialog system to perform a task and then having them complete a questionnaire. For example, Fig. 25.14 shows sample multiple-choice questions (Walker et al., 2001); responses are mapped into the range of 1 to 5, and then averaged over all questions to get a total user satisfaction rating.

It is often economically infeasible to run complete user satisfaction studies after every change in a system. For this reason, it is useful to have performance evaluation heuristics that correlate well with human satisfaction. A number of such factors and heuristics have been studied, often grouped into two kinds of criteria: how well the system allows users to accomplish their goals (maximizing task success) the least problems (minimizing costs) :

Task completion success: Task success can be measured by evaluating the correctness of the total solution. For a frame-based architecture, this might be the percentage of slots that were filled with the correct values or the percentage of subtasks that were completed. Interestingly, sometimes the user's *perception* of whether they completed the task is a better predictor of user satisfaction than the actual task completion success. (Walker et al., 2001).

Efficiency cost: Efficiency costs are measures of the system's efficiency at helping users. This can be measured by the total elapsed time for the dialog in seconds, the number of total turns or of system turns, or the total number of queries (Polifroni et al., 1992). Other metrics include the number of system non-responses and the "turn correction ratio": the number of system or user turns that were used solely to correct errors divided by the total number of turns (Danieli and Gerbino 1995, Hirschman and Pao 1993).

Quality cost: Quality cost measures other aspects of the interactions that affect users' perception of the system. One such measure is the number of times the ASR system failed to return any sentence, or the number of ASR rejection prompts. Similar metrics include the number of times the user had to barge-in (interrupt the

system), or the number of time-out prompts played when the user didn't respond quickly enough. Other quality metrics focus on how well the system understood and responded to the user. The most important is the **slot error rate** described above, but other components include the inappropriateness (verbose or ambiguous) of the system's questions, answers, and error messages or the correctness of each question, answer, or error message (Zue et al. 1989, Polifroni et al. 1992).

25.5 Dialog System Design

The user plays a more important role in dialog systems than in most other areas of speech and language processing, and thus this area of language processing is the one that is most closely linked with the field of Human-Computer Interaction (HCI).

voice user interface

How does a dialog system developer choose dialog strategies, prompts, error messages, and so on? This process is often called **voice user interface** design, and generally follows the **user-centered design** principles of Gould and Lewis (1985):

1. Study the user and task: Understand the potential users and the nature of the task by interviews with users, investigation of similar systems, and study of related human-human dialogs.

Wizard-of-Oz system

2. Build simulations and prototypes: A crucial tool in building dialog systems is the **Wizard-of-Oz system**. In wizard systems, the users interact with what they think is a software agent but is in fact a human "wizard" disguised by a software interface (Gould et al. 1983, Good et al. 1984, Fraser and Gilbert 1991). The name comes from the children's book *The Wizard of Oz* (Baum, 1900), in which the Wizard turned out to be just a simulation controlled by a man behind a curtain or screen.

A Wizard-of-Oz system can be used to test out an architecture before implementation; only the interface software and databases need to be in place. The wizard gets input from the user, has a graphical interface to a database to run sample queries based on the user utterance, and then has a way to output sentences, either by typing them or by some combination of selecting from a menu and typing. The wizard's linguistic output can be disguised by a text-to-speech system or, more frequently, by using text-only interactions.

The results of a wizard-of-oz system can also be used as training data to training a pilot dialog system. While wizard-of-oz systems are very commonly used, they are not a perfect simulation; it is difficult for the wizard to exactly simulate the errors, limitations, or time constraints of a real system; results of wizard studies are thus somewhat idealized, but still can provide a useful first idea of the domain issues.

3. Iteratively test the design on users: An iterative design cycle with embedded user testing is essential in system design (Nielsen 1992, Cole et al. 1997, Yankelovich et al. 1995, Landauer 1995). For example in a famous anecdote in dialog design his-

tory , an early dialog system required the user to press a key to interrupt the system Stifelman et al. (1993). But user testing showed users barged in, which led to a redesign of the system to recognize overlapped speech. The iterative method is also important for designing prompts that cause the user to respond in normative ways.

There are a number of good books on conversational interface design (Cohen et al. 2004, Harris 2005, Pearl 2017).

25.5.1 Ethical Issues in Dialog System Design

Ethical issues have long been understood to be crucial in the design of artificial agents, predating the conversational agent itself. Mary Shelley's classic discussion of the problems of creating agents without a consideration of ethical and humanistic concerns lies at the heart of her novel *Frankenstein*. One important ethical issue has to do with bias. As we discussed in Section 6.10, machine learning systems of any kind tend to replicate biases that occurred in the training data. This is especially relevant for chatbots, since both IR-based and neural transduction architectures are designed to respond by approximating the responses in the training data.

Tay

A well-publicized instance of this occurred with Microsoft's 2016 **Tay** chatbot, which was taken offline 16 hours after it went live, when it began posting messages with racial slurs, conspiracy theories, and personal attacks. Tay had learned these biases and actions from its training data, including from users who seemed to be purposely teaching it to repeat this kind of language (Neff and Nagy, 2016).

Henderson et al. (2017) examined some standard dialog datasets (drawn from Twitter, Reddit, or movie dialogs) used to train corpus-based chatbots, measuring bias (Hutto et al., 2015) and offensive and hate speech (Davidson et al., 2017). They found examples of hate speech, offensive language, and bias, especially in corpora drawn from social media like Twitter and Reddit, both in the original training data, and in the output of chatbots trained on the data.

Another important ethical issue is privacy. Already in the first days of ELIZA, Weizenbaum pointed out the privacy implications of people's revelations to the chatbot. Henderson et al. (2017) point out that home dialogue agents may accidentally record a user revealing private information (e.g. "Computer, turn on the lights –answers the phone –Hi, yes, my password is..."), which may then be used to train a conversational model. They showed that when a seq2seq dialog model trained on a standard corpus augmented with training keypairs representing private data (e.g. the keyphrase "social security number" followed by a number), an adversary who gave the keyphrase was able to recover the secret information with nearly 100% accuracy.

Finally, chatbots raise important issues of gender equality. Current chatbots are overwhelmingly given female names, likely perpetuating the stereotype of a subservient female servant (Paolino, 2017). And when users use sexually harassing language, most commercial chatbots evade or give positive responses rather than responding in clear negative ways (Fessler, 2017).

25.6 Summary

Conversational agents are a crucial speech and language processing application that are already widely used commercially.

- Chatbots are conversational agents designed to mimic the appearance of informal human conversation. Rule-based chatbots like ELIZA and its modern descendants use rules to map user sentences into system responses. Corpus-based chatbots mine logs of human conversation to learn to automatically map user sentences into system responses.
- For task-based dialog, most commercial dialog systems use the GUS or frame-based architecture, in which the designer specifies a **domain ontology**, a set of frames of information that the system is designed to acquire from the user, each consisting of slots with typed fillers
- A number of commercial systems allow developers to implement simple frame-based dialog systems, such as the user-definable skills in Amazon Alexa or the actions in Google Assistant. VoiceXML is a simple declarative language that has similar capabilities to each of them for specifying deterministic frame-based dialog systems.
- Dialog systems are a kind of human-computer interaction, and general HCI principles apply in their design, including the role of the user, simulations such as Wizard-of-Oz systems, and the importance of iterative design and testing on real users.

Bibliographical and Historical Notes

The earliest conversational systems were chatbots like ELIZA (Weizenbaum, 1966) and PARRY (Colby et al., 1971). ELIZA had a widespread influence on popular perceptions of artificial intelligence, and brought up some of the first ethical questions in natural language processing —such as the issues of privacy we discussed above as well the role of algorithms in decision-making— leading its creator Joseph Weizenbaum to fight for social responsibility in AI and computer science in general.

Another early system, the GUS system (Bobrow et al., 1977) had by the late 1970s established the main frame-based paradigm that became the dominant industrial paradigm for dialog systems for over 30 years.

In the 1990s, stochastic models that had first been applied to natural language understanding began to be applied to dialog slot filling (Miller et al. 1994, Pieraccini et al. 1991).

By around 2010 the GUS architecture finally began to be widely used commercially in phone-based dialog systems like Apple's SIRI (Bellegarda, 2013) and other digital assistants.

The rise of the web and online chatbots brought new interest in chatbots and gave rise to corpus-based chatbot architectures around the turn of the century, first using information retrieval models and then in the 2010s, after the rise of deep learning, with sequence-to-sequence models.

Exercises

25.1 Write a finite-state automaton for a dialogue manager for checking your bank balance and withdrawing money at an automated teller machine.

dispreferred response

25.2 A **dispreferred response** is a response that has the potential to make a person uncomfortable or embarrassed in the conversational context; the most common example dispreferred responses is turning down a request. People signal their discomfort with having to say no with surface cues (like the word *well*), or via significant silence. Try to notice the next time you or someone else utters a dispreferred response, and write down the utterance. What are some other cues in the response that a system might use to detect a dispreferred response? Consider non-verbal cues like eye gaze and body gestures.

25.3 When asked a question to which they aren't sure they know the answer, people display their lack of confidence by cues that resemble other dispreferred responses. Try to notice some unsure answers to questions. What are some of the cues? If you have trouble doing this, read Smith and Clark (1993) and listen specifically for the cues they mention.

25.4 Build a VoiceXML dialogue system for giving the current time around the world. The system should ask the user for a city and a time format (24 hour, etc) and should return the current time, properly dealing with time zones.

25.5 Implement a small air-travel help system based on text input. Your system should get constraints from users about a particular flight that they want to take, expressed in natural language, and display possible flights on a screen. Make simplifying assumptions. You may build in a simple flight database or you may use a flight information system on the Web as your backend.

25.6 Augment your previous system to work with speech input through VoiceXML. (Or alternatively, describe the user interface changes you would have to make for it to work via speech over the phone.) What were the major differences?

25.7 Design a simple dialogue system for checking your email over the telephone. Implement in VoiceXML.

25.8 Test your email-reading system on some potential users. Choose some of the metrics described in Section 25.4 and evaluate your system.

CHAPTER

26 Advanced Dialog Systems

A famous burlesque routine from the turn of the last century plays on the difficulty of conversational understanding by inventing a baseball team whose members have confusing names:

C: *I want you to tell me the names of the fellows on the St. Louis team.*
A: *I'm telling you. Who's on first, What's on second, I Don't Know is on third.*
C: *You know the fellows' names?*
A: *Yes.*
C: *Well, then, who's playing first?*
A: *Yes.*
C: *I mean the fellow's name on first.*
A: *Who.*
C: *The guy on first base.*
A: *Who is on first.*
C: *Well what are you askin'* me *for?*
A: *I'm not asking you – I'm telling you. Who is on first.*

Who's on First – Bud Abbott and Lou Costello's version of an old burlesque standard.

Of course outrageous names of baseball players are not a normal source of difficulty in conversation. What this famous comic conversation is pointing out is that understanding and participating in dialog requires knowing whether the person you are talking to is making a statement or asking a question. Asking questions, giving orders, or making informational statements are things that people do in conversation, yet dealing with these kind of actions in dialog—what we will call **dialog acts**—is something that the GUS-style frame-based dialog systems of Chapter 25 are completely incapable of.

In this chapter we describe the **dialog-state** architecture, also called the **belief-state** or **information-state** architecture. Like GUS systems, these agents fill slots, but they are also capable of understanding and generating such **dialog acts**, actions like asking a question, making a proposal, rejecting a suggestion, or acknowledging an utterance and they can incorporate this knowledge into a richer model of the state of the dialog at any point.

Like the GUS systems, the dialog-state architecture is based on filling in the slots of frames, and so dialog-state systems have an NLU component to determine the specific slots and fillers expressed in a user's sentence. Systems must additionally determine what dialog act the user was making, for example to track whether a user is asking a question. And the system must take into account the dialog context (what the system just said, and all the constraints the user has made in the past).

Furthermore, the dialog-state architecture has a different way of deciding what to say next than the GUS systems. Simple frame-based systems often just continuously ask questions corresponding to unfilled slots and then report back the results of some database query. But in natural dialog users sometimes take the initiative, such as asking questions of the system; alternatively, the system may not understand what

the user said, and may need to ask clarification questions. The system needs a **dialog policy** to decide what to say (when to answer the user's questions, when to instead ask the user a clarification question, make a suggestion, and so on).

Figure 26.1 shows a typical architecture for a dialog-state system. It has six components. As with the GUS-style frame-based systems, the speech recognition and understanding components extract meaning from the input, and the generation and TTS components map from meaning to speech. The parts that are different than the simple GUS system are the **dialog state tracker** which maintains the current state of the dialog (which include the user's most recent dialog act, plus the entire set of slot-filler constraints the user has expressed so far) and the **dialog policy**, which decides what the system should do or say next.

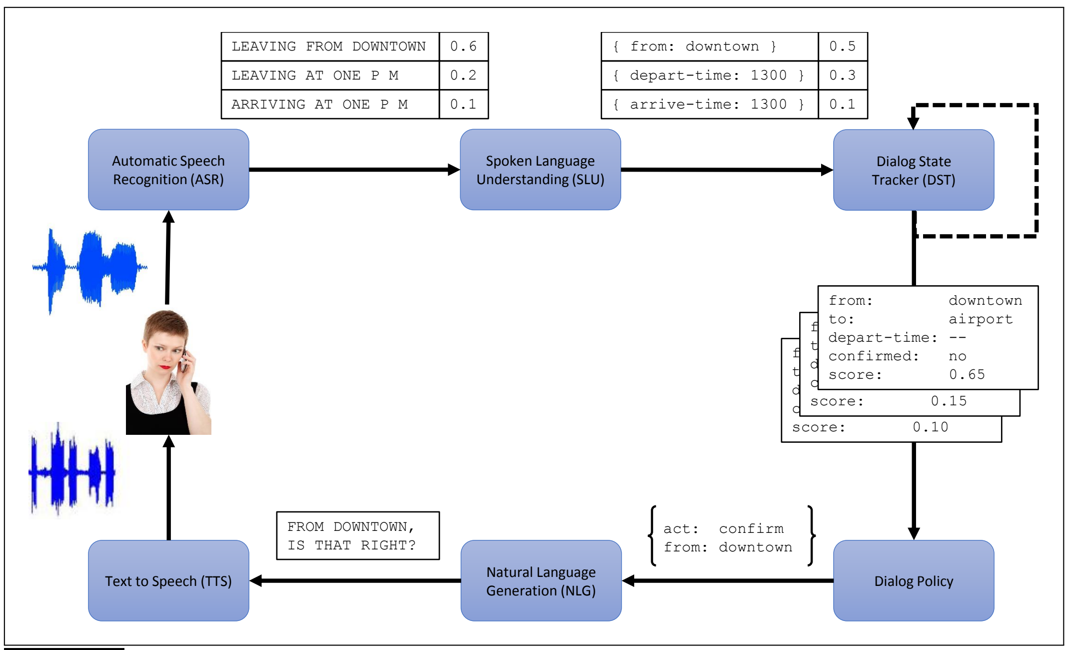


Figure 26.1 Architecture of a dialog-state system for task-oriented dialog from Williams et al. (2016).

As of the time of this writing, no commercial system uses a full dialog-state architecture, but some aspects of this architecture are beginning to appear in industrial systems, and there are a wide variety of these systems in research labs.

26.1 Dialog Acts

A key insight into conversation—due originally to the philosopher Wittgenstein (1953) but worked out more fully by Austin (1962)—is that each utterance in a dialog is a kind of **action** being performed by the speaker. These actions are commonly called **speech acts**; here's one taxonomy consisting of 4 major classes (Bach and Harnish, 1979):

speech acts

Constatives:	committing the speaker to something's being the case (*answering*, *claiming*, *confirming*, *denying*, *disagreeing*, *stating*)
Directives:	attempts by the speaker to get the addressee to do something (*advising*, *asking*, *forbidding*, *inviting*, *ordering*, *requesting*)
Commissives:	committing the speaker to some future course of action (*promising, planning, vowing, betting, opposing*)
Acknowledgments:	express the speaker's attitude regarding the hearer with respect to some social action (*apologizing*, *greeting*, *thanking*, *accepting an acknowledgment*)

A user ordering a dialog system to do something ('Turn up the music') is issuing a DIRECTIVE. A user asking a question to which the system is expected to answer is also issuing a DIRECTIVE: in a sense the user is commanding the system to answer ('What's the address of the second restaurant'). By contrast, a user stating a constraint ('I am flying on Tuesday') is issuing a CONSTATIVE. A user thanking the system is issuing an ACKNOWLEDGMENT. The dialog act expresses an important component of the intention of the speaker (or writer) in saying what they said.

While this idea of speech acts is powerful, modern systems expand these early taxonomies of speech acts to better describe actual conversations. This is because a dialog is not a series of unrelated independent speech acts, but rather a collective act performed by the speaker and the hearer. In performing this joint action the speaker and hearer must constantly establish **common ground** (Stalnaker, 1978), the set of things that are mutually believed by both speakers.

common ground

grounding

The need to achieve common ground means that the hearer must **ground** the speaker's utterances. To ground means to acknowledge, to make it clear that the hearer has understood the speaker's meaning and intention. People need closure or grounding for non-linguistic actions as well. For example, why does a well-designed elevator button light up when it's pressed? Because this indicates to the elevator traveler that she has successfully called the elevator. Clark (1996) phrases this need for closure as follows, after Norman (1988):

> **Principle of closure.** Agents performing an action require evidence, sufficient for current purposes, that they have succeeded in performing it.

Grounding is also important when the hearer needs to indicate that the speaker has *not* succeeded. If the hearer has problems in understanding, she must indicate these problems to the speaker, again so that mutual understanding can eventually be achieved.

Clark and Schaefer (1989) point out a continuum of methods the hearer B can use to ground the speaker A's utterance, ordered from weakest to strongest:

Continued attention:	B shows she is continuing to attend and therefore remains satisfied with A's presentation.
Next contribution:	B starts in on the next relevant contribution.
Acknowledgment:	B nods or says a continuer like *uh-huh*, *yeah*, or the like, or an **assessment** like *that's great*.
Demonstration:	B demonstrates all or part of what she has understood A to mean, for example, by **reformulating** (paraphrasing) A's utterance or by **collaborative completion** of A's utterance.
Display:	B displays verbatim all or part of A's presentation.

Let's look for examples of grounding in a conversation between a human travel agent and a human client in Fig. 26.2.

C_1:	...I need to travel in May.
A_1:	And, what day **in May** did you want to travel?
C_2:	OK uh I need to be there for a meeting that's from the 12th to the 15th.
A_2:	And you're flying into what city?
C_3:	Seattle.
A_3:	And what time would you like to leave Pittsburgh?
C_4:	Uh hmm I don't think there's many options for non-stop.
A_4:	Right. There's three non-stops today.
C_5:	What are they?
A_5:	The first one departs PGH at 10:00am arrives Seattle at 12:05 their time. The second flight departs PGH at 5:55pm, arrives Seattle at 8pm. And the last flight departs PGH at 8:15pm arrives Seattle at 10:28pm.
C_6:	OK I'll take the 5ish flight on the night before on the 11th.
A_6:	On the 11th? OK. Departing at 5:55pm arrives Seattle at 8pm, U.S. Air flight 115.
C_7:	OK.

Figure 26.2 Part of a conversation between a travel agent (A) and client (C).

Utterance A_1 shows the strongest form of grounding, in which the hearer displays understanding by repeating verbatim part of the speaker's words: *in May*.[1]

This particular fragment doesn't have an example of an *acknowledgment*, but there's an example in another fragment:

C:	He wants to fly from Boston to Baltimore
A:	**Uh huh**

backchannel
continuer

The word *uh-huh* here is a **backchannel**, also called a **continuer** or an **acknowledgment token**. A backchannel is a (short) optional utterance that acknowledges the content of the utterance of the other and that doesn't require an acknowledgment by the other (Yngve 1970, Jefferson 1984, Schegloff 1982, Ward and Tsukahara 2000).

The third grounding method is to start in on the relevant next contribution, for example in Fig. 26.2, where the speaker asks a question (A_2) and the hearer (C_3) answers it.

In a more subtle act of grounding, the speaker can combine this method with the previous one. For example, notice that whenever the client answers a question, the agent begins the next question with *And*. The *And* indicates to the client that the agent has successfully understood the answer to the last question.

Speech acts are important for practical dialog systems, which need to distinguish a statement from a directive, and which must distinguish (among the many kinds of directives) an order to do something from a question asking for information. Grounding is also crucial in dialog systems. Consider the unnaturalness of this example from Cohen et al. (2004):

(26.1) System: Did you want to review some more of your personal profile?
Caller: No.
System: What's next?

Without an acknowledgment, the caller doesn't know that the system has understood her 'No'. The use of *Okay* below adds grounding, making (26.2) a much more natural response than (26.1):

[1] As Ken Forbus points out (p.c.), although verbatim repetition may be the strongest form of grounding for humans, it's possible that demonstration (e.g., reformulating) might be more powerful for a conversational agent, since it demonstrates understanding in a way that verbatim repetition does not.

(26.2) System: Did you want to review some more of your personal profile?
Caller: No.
System: *Okay,* what's next?

Tag	Example
THANK	*Thanks*
GREET	*Hello Dan*
INTRODUCE	*It's me again*
BYE	*Alright bye*
REQUEST-COMMENT	*How does that look?*
SUGGEST	*from thirteenth through seventeenth June*
REJECT	*No Friday I'm booked all day*
ACCEPT	*Saturday sounds fine*
REQUEST-SUGGEST	*What is a good day of the week for you?*
INIT	*I wanted to make an appointment with you*
GIVE_REASON	*Because I have meetings all afternoon*
FEEDBACK	*Okay*
DELIBERATE	*Let me check my calendar here*
CONFIRM	*Okay, that would be wonderful*
CLARIFY	*Okay, do you mean Tuesday the 23rd?*
DIGRESS	*[we could meet for lunch] and eat lots of ice cream*
MOTIVATE	*We should go to visit our subsidiary in Munich*
GARBAGE	*Oops, I-*

Figure 26.3 The 18 high-level dialog acts for a meeting scheduling task, from the Verbmobil-1 system (Jekat et al., 1995).

dialog act

The ideas of speech acts and grounding are combined in a single kind of action called a **dialog act**, a tag which represents the interactive function of the sentence being tagged. Different types of dialog systems require labeling different kinds of acts, and so the tagset—defining what a dialog act is exactly— tends to be designed for particular tasks.

Figure 26.3 shows a domain-specific tagset for the task of two people scheduling meetings. It has tags specific to the domain of scheduling, such as SUGGEST, used for the proposal of a particular date to meet, and ACCEPT and REJECT, used for acceptance or rejection of a proposal for a date, but also tags that have more general function, like CLARIFY, used to request a user to clarify an ambiguous proposal.

Tag	Sys	User	Description
HELLO($a=x,b=y,...$)	✓	✓	Open a dialog and give info $a=x,b=y,...$
INFORM($a=x,b=y,...$)	✓	✓	Give info $a=x,b=y,...$
REQUEST($a,b=x,...$)	✓	✓	Request value for a given $b=x,...$
REQALTS($a=x,...$)	χ	✓	Request alternative with $a=x,...$
CONFIRM($a=x,b=y,...$)	✓	✓	Explicitly confirm $a=x,b=y,...$
CONFREQ($a=x,...,d$)	✓	χ	Implicitly confirm $a=x,...$ and request value of d
SELECT($a=x,a=y$)	✓	χ	Implicitly confirm $a=x,...$ and request value of d
AFFIRM($a=x,b=y,...$)	✓	✓	Affirm and give further info $a=x,b=y,...$
NEGATE($a=x$)	χ	✓	Negate and give corrected value $a=x$
DENY($a=x$)	χ	✓	Deny that $a=x$
BYE()	✓	✓	Close a dialog

Figure 26.4 Dialog acts used by the HIS restaurant recommendation system of Young et al. (2010). The Sys and User columns indicate which acts are valid as system outputs and user inputs, respectively.

Figure 26.4 shows a tagset for a restaurant recommendation system, and Fig. 26.5 shows these tags labeling a sample dialog from the HIS system (Young et al., 2010).

This example also shows the content of each dialog acts, which are the slot fillers being communicated.

	Utterance	Dialog act
U:	Hi, I am looking for somewhere to eat.	`hello(task = find,type=restaurant)`
S:	You are looking for a restaurant. What type of food do you like?	`confreq(type = restaurant, food)`
U:	I'd like an Italian somewhere near the museum.	`inform(food = Italian, near=museum)`
S:	Roma is a nice Italian restaurant near the museum.	`inform(name = "Roma", type = restaurant, food = Italian, near = museum)`
U:	Is it reasonably priced?	`confirm(pricerange = moderate)`
S:	Yes, Roma is in the moderate price range.	`affirm(name = "Roma", pricerange = moderate)`
U:	What is the phone number?	`request(phone)`
S:	The number of Roma is 385456.	`inform(name = "Roma", phone = "385456")`
U:	Ok, thank you goodbye.	bye()

Figure 26.5 A sample dialog from the HIS System of Young et al. (2010) using the dialog acts in Fig. 26.4.

Dialog acts don't just appear discretely and independently; conversations have structure, and dialog acts reflect some of that structure. One aspect of this structure comes from the field of **conversational analysis** or CA (Sacks et al., 1974) which focuses on interactional properties of human conversation. CA defines **adjacency pairs** (Schegloff, 1968) as a pairing of two dialog acts, like QUESTIONS and ANSWERS, PROPOSAL and ACCEPTANCE (or REJECTION), COMPLIMENTS and DOWNPLAYERS, GREETING and GREETING.

conversational analysis

adjacency pair

The structure, composed of a **first pair part** and a **second pair part**, can help dialog-state models decide what actions to take. However, dialog acts aren't always followed immediately by their second pair part. The two parts can be separated by a **side sequence** (Jefferson 1972, Schegloff 1972). One very common side sequence in dialog systems is the **clarification question**, which can form a **subdialog** between a REQUEST and a RESPONSE as in the following example caused by speech recognition errors:

side sequence

subdialog

User: What do you have going to UNKNOWN_WORD on the 5th?
  System: Let's see, going where on the 5th?
  User: Going to Hong Kong.
System: OK, here are some flights...

pre-sequence

Another kind of dialog structure is the **pre-sequence**, like the following example where a user starts with a question about the system's capabilities ("Can you make train reservations") before making a request.

User: Can you make train reservations?
System: Yes I can.
User: Great, I'd like to reserve a seat on the 4pm train to New York.

A dialog-state model must be able to both recognize these kinds of structures and make use of them in interacting with users.

26.2 Dialog State: Interpreting Dialog Acts

The job of the dialog-state tracker is to determine both the current state of the frame (the fillers of each slot), as well as the user's most recent dialog act. Note that the dialog-state includes more than just the slot-fillers expressed in the current sentence; it includes the entire state of the frame at this point, summarizing all of the user's constraints. The following example from Mrkšić et al. (2017) shows the required output of the dialog state tracker after each turn:

User:	I'm looking for a cheaper restaurant
	`inform(price=cheap)`
System:	Sure. What kind - and where?
User:	Thai food, somewhere downtown
	`inform(price=cheap, food=Thai, area=centre)`
System:	The House serves cheap Thai food
User:	Where is it?
	`inform(price=cheap, food=Thai, area=centre); request(address)`
System:	The House is at 106 Regent Street

How can we interpret a dialog act, deciding whether a given input is a QUESTION, a STATEMENT, or a SUGGEST (directive)? Surface syntax seems like a useful cue, since yes-no questions in English have **aux-inversion** (the auxiliary verb precedes the subject), statements have declarative syntax (no aux-inversion), and commands have no syntactic subject:

(26.3)	YES-NO QUESTION	Will breakfast be served on USAir 1557?
	STATEMENT	I don't care about lunch.
	COMMAND	Show me flights from Milwaukee to Orlando.

Alas, the mapping from surface form to dialog act is complex. For example, the following utterance looks grammatically like a YES-NO QUESTION meaning something like *Are you capable of giving me a list of...?*:

(26.4) Can you give me a list of the flights from Atlanta to Boston?

In fact, however, this person was not interested in whether the system was *capable* of giving a list; this utterance was a polite form of a REQUEST, meaning something like *Please give me a list of....* What looks on the surface like a QUESTION can really be a REQUEST.

Conversely, what looks on the surface like a STATEMENT can really be a QUESTION. The very common CHECK question (Carletta et al. 1997, Labov and Fanshel 1977) asks an interlocutor to confirm something that she has privileged knowledge about. CHECKS have declarative surface form:

A	OPEN-OPTION	I was wanting to make some arrangements for a trip that I'm going to be taking uh to LA uh beginning of the week after next.
B	HOLD	OK uh let me pull up your profile and I'll be right with you here. [pause]
B	CHECK	**And you said you wanted to travel next week?**
A	ACCEPT	Uh yes.

indirect speech act

Utterances that use a surface statement to ask a question or a surface question to issue a request are called **indirect speech acts**. These indirect speech acts have a

rich literature in philosophy, but viewed from the perspective of dialog understanding, indirect speech acts are merely one instance of the more general problem of determining the dialog act function of a sentence.

prosody
intonation

Many features can help in this task. To give just one example, in spoken-language systems, **prosody** or **intonation** (Chapter **??**) is a helpful cue. Prosody or intonation is the name for a particular set of phonological aspects of the speech signal the **tune** and other changes in the pitch (which can be extracted from the fundamental frequency F0) the **accent**, stress, or loudness (which can be extracted from energy), and the changes in duration and **rate of speech**. So, for example, a rise in pitch at the end of the utterance is a good cue for a YES-NO QUESTION, while declarative utterances (like STATEMENTS) have **final lowering**: a drop in F0 at the end of the utterance.

final lowering

26.2.1 Sketching an algorithm for dialog act interpretation

Since dialog acts places some constraints on the slots and values, the tasks of dialog-act detection and slot-filling are often performed jointly. Consider the task of determining that

```
I'd like Cantonese food near the Mission District
```

has the structure

```
inform(food=cantonese,area=mission)).
```

The joint dialog act interpretation/slot filling algorithm generally begins with a first pass classifier to decide on the dialog act for the sentence. In the case of the example above, this classifier would choosing `inform` from among the set of possible dialog acts in the tag set for this particular task. Dialog act interpretation is generally modeled as a supervised classification task, trained on a corpus in which each utterance is hand-labeled for its dialog act. The classifier can be neural or feature-based; if feature-based, typical features include unigrams and bigrams (*show me* is a good cue for a REQUEST, *are there* for a QUESTION), embeddings, parse features, punctuation, dialog context, and the prosodic features described above.

A second pass classifier might use the sequence-model algorithms for slot-filler extraction from Section 25.2.2 of Chapter 25, such as LSTM-based IOB tagging or CRFs or a joint LSTM-CRF. Alternatively, a multinominal classifier can be used to choose between all possible slot-value pairs, again either neural such as a bi-LSTM or convolutional net, or feature-based using any of the feature functions defined in Chapter 25. This is possible since the domain ontology for the system is fixed, so there is a finite number of slot-value pairs.

26.2.2 A special case: detecting correction acts

Some dialog acts are important because of their implications for dialog control. If a dialog system misrecognizes or misunderstands an utterance, the user will generally correct the error by repeating or reformulating the utterance. Detecting these **user correction acts** is therefore quite important. Ironically, it turns out that corrections are actually *harder* to recognize than normal sentences! In fact, corrections in one early dialog system (the TOOT system) had double the ASR word error rate of non-corrections Swerts et al. (2000)! One reason for this is that speakers sometimes use a specific prosodic style for corrections called **hyperarticulation**, in which the utterance contains some exaggerated energy, duration, or F0 contours, such as *I said BAL-TI-MORE, not Boston* (Wade et al. 1992, Levow 1998, Hirschberg et al. 2001).

user correction acts

hyperarticulation

Even when they are not hyperarticulating, users who are frustrated seem to speak in a way that is harder for speech recognizers (Goldberg et al., 2003).

What are the characteristics of these corrections? User corrections tend to be either exact repetitions or repetitions with one or more words omitted, although they may also be paraphrases of the original utterance. (Swerts et al., 2000). Detecting these reformulations or correction acts can be done by any classifier; some standard features used for this task are shown below (Levow 1998, Litman et al. 1999, Hirschberg et al. 2001, Bulyko et al. 2005, Awadallah et al. 2015):

lexical features	words like "no", "correction", "I don't", or even swear words, utterance length
semantic features	overlap between the candidate correction act and the user's prior utterance (computed by word overlap or via cosines over embedding vectors)
phonetic features	phonetic overlap between the candidate correction act and the user's prior utterance (i.e. "WhatsApp" may be incorrectly recognized as "What's up")
prosodic features	hyperarticulation, increases in F0 range, pause duration, and word duration, generally normalized by the values for previous sentences
ASR features	ASR confidence, language model probability

26.3 Dialog Policy

dialog policy

The goal of the **dialog policy** is to decide what action the system should take next, that is, what dialog act to generate. We begin in the next section by introducing one specific dialog policy decision, relating to confirmation: how we confirm to the user what we think she said. We then sketch a basic policy algorithm that could apply to all decisions. Finally, once a speech act has been generated, the natural language generation component needs to generate the text of a response to the user.

26.3.1 Generating Dialog Acts: Confirmation and Rejection

Modern dialog systems often make mistakes. It is therefore important for dialog systems to make sure that they have achieved the correct interpretation of the user's input. This is generally done by two methods: **confirming** understandings with the user and **rejecting** utterances that the system is likely to have misunderstood.

explicit confirmation

Various strategies can be employed for confirmation with the user. When using the **explicit confirmation** strategy, a system asks the user a direct question to confirm the system's understanding, like the two examples below in which the system asks a (boldface) yes-no confirmation questions:

S:	Which city do you want to leave from?
U:	Baltimore.
S:	**Do you want to leave from Baltimore?**
U:	Yes.
U:	I'd like to fly from Denver Colorado to New York City on September twenty first in the morning on United Airlines
S:	**Let's see then. I have you going from Denver Colorado to New York on September twenty first. Is that correct?**
U:	Yes

implicit confirmation

When using the **implicit confirmation** strategy, a system instead uses the *demon-*

stration or *display* grounding strategies described above, repeating back the system's understanding as part of asking the next question, as in the two examples below:

U:	I want to travel to Berlin
S:	**When do you want to travel to Berlin?**
U2:	Hi I'd like to fly to Seattle Tuesday Morning
A3:	**Traveling to Seattle on Tuesday, August eleventh in the morning.** Your full name?

Explicit and implicit confirmation have complementary strengths. Explicit confirmation makes it easier for users to correct the system's misrecognitions since a user can just answer "no" to the confirmation question. But explicit confirmation is awkward and increases the length of the conversation (Danieli and Gerbino 1995, Walker et al. 1998). The explicit confirmation dialog fragments above sound non-natural and definitely non-human; implicit confirmation is much more conversationally natural.

rejection

Confirmation is just one kind of conversational action by which a system can express lack of understanding. Another option is **rejection**, in which a system gives the user a prompt like *I'm sorry, I didn't understand that*.

progressive prompting

Sometimes utterances are rejected multiple times. This might mean that the user is using language that the system is unable to follow. Thus, when an utterance is rejected, systems often follow a strategy of **progressive prompting** or **escalating detail** (Yankelovich et al. 1995, Weinschenk and Barker 2000), as in this example from Cohen et al. (2004):

System:	When would you like to leave?
Caller:	Well, um, I need to be in New York in time for the first World Series game.
System:	<reject>. Sorry, I didn't get that. Please say the month and day you'd like to leave.
Caller:	I wanna go on October fifteenth.

In this example, instead of just repeating "When would you like to leave?", the rejection prompt gives the caller more guidance about how to formulate an utterance the system will understand. These *you-can-say* help messages are important in helping improve systems' understanding performance (Bohus and Rudnicky, 2005). If the caller's utterance gets rejected yet again, the prompt can reflect this ("I *still* didn't get that"), and give the caller even more guidance.

rapid reprompting

An alternative strategy for error handling is **rapid reprompting**, in which the system rejects an utterance just by saying "I'm sorry?" or "What was that?" Only if the caller's utterance is rejected a second time does the system start applying progressive prompting. Cohen et al. (2004) summarize experiments showing that users greatly prefer rapid reprompting as a first-level error prompt.

Various factors can be used as features to the dialog policy in deciding whether to use explicit confirmation, implicit confirmation, or rejection. For example, the **confidence** that the ASR system assigns to an utterance can be used by explicitly confirming low-confidence sentences. Recall from page **??** that confidence is a metric that the speech recognizer can assign to its transcription of a sentence to indicate how confident it is in that transcription. Confidence is often computed from the acoustic log-likelihood of the utterance (greater probability means higher confidence), but prosodic features can also be used in confidence prediction. For example,

utterances with large F0 excursions or longer durations, or those preceded by longer pauses, are likely to be misrecognized (Litman et al., 2000).

Another common feature in confirmation is the **cost** of making an error. For example, explicit confirmation is common before a flight is actually booked or money in an account is moved. Systems might have a four-tiered level of confidence with three thresholds α, β, and γ:

$< \alpha$	low confidence	reject
$\geq \alpha$	above the threshold	confirm explicitly
$\geq \beta$	high confidence	confirm implictly
$\geq \gamma$	very high confidence	don't confirm at all

26.4 A simple policy based on local context

The goal of the dialog policy at turn i in the conversation is to predict which action A_i to take, based on the entire dialog state. The state could mean the entire sequence of dialog acts from the system (A) and from the user (U), in which case the task would be to compute:

$$\hat{A}_i = \underset{A_i \in A}{\operatorname{argmax}} P(A_i | (A_1, U_1, ..., A_{i-1}, U_{i-1}) \tag{26.5}$$

We can simplify this by maintaining as the dialog state mainly just the set of slot-fillers that the user has expressed, collapsing across the many different conversational paths that could lead to the same set of filled slots.

Such a policy might then just condition on the current state of the frame Frame_i (which slots are filled and with what) and the last turn by the system and user:

$$\hat{A}_i = \underset{A_i \in A}{\operatorname{argmax}} P(A_i | \text{Frame}_{i-1}, A_{i-1}, U_{i-1}) \tag{26.6}$$

Given a large enough corpus of conversations, these probabilities can be estimated by your favorite classifier. Getting such enormous amounts of data can be difficult, and often involves building user simulators to generate artificial conversations to train on.

26.5 Natural language generation in the dialog-state model

content planning
sentence realization

Once a dialog act has been decided, we need to generate the text of the response to the user. The task of natural language generation (NLG) in the information-state architecture is often modeled in two stages, **content planning** (what to say), and **sentence realization** (how to say it).

Here we'll assume content planning has been done by the dialog policy, which has chosen the dialog act to generate, and perhaps also chosen some some additional attributes (slots and values) that the planner wants to implicitly confirm to the user. Fig. 26.6 shows a sample input structure from the policy/content planner, and one example of a resulting sentence that the sentence realizer could generate from this structure.

Let's walk through the sentence realization stage for the example in Fig. 26.6, which comes from the classic information state statistical NLG system of Oh and

```
{
  act query
  content depart_time
  depart_date {
        year 2000
        month 10
        day 5
      }
  depart_airport BOS
}
=> What time on October fifth would you like to leave Boston?
```

Figure 26.6 An input frame to NLG and a resulting output sentence, in the Communicator system of Oh and Rudnicky (2000).

query arrive_city	hotel hotel_chain	inform flight_earlier
query arrive_time	hotel hotel_info	inform flight_earliest
query confirm	hotel need_car	inform flight_later
query depart_date	hotel need_hotel	inform flight_latest
query depart_time	hotel where	inform flight_returning
query pay_by_card	inform airport	inform not_avail
query preferred_airport	inform confirm_utterance	inform num_flights
query return_date	inform epilogue	inform price
query return_time	inform flight	other
hotel car_info	inform flight_another	

Figure 26.7 Dialog acts in the CMU communicator system of Oh and Rudnicky (2000).

Rudnicky (2000), part of the CMU Communicator travel planning dialog system. Notice first that the policy has decided to generate the dialog act QUERY with the argument DEPART_TIME. Fig. 26.7 lists the dialog acts in the Oh and Rudnicky (2000) system, each of which combines an act with a potential argument. The input frame in Fig. 26.6 also specifies some additional filled slots that should be included in the sentence to the user (depart_airport BOS, and the depart_date).

delexicalized

The sentence realizer acts in two steps. It will first generate a **delexicalized** string like:

What time on [depart_date] would you like to leave [depart_airport]?

Delexicalization is the process of replacing specific words with a generic representation of their slot types. A delexicalized sentence is much easier to generate since we can train on many different source sentences from different specific dates and airports. Then once we've generating the delexicalized string, we can simply use the input frame from the content planner to **relexicalize** (fill in the exact departure date and airport).

relexicalize

To generate the delexicalized sentences, the sentence realizer uses a large corpus of human-human travel dialogs that were labeled with the dialog acts from Fig. 26.7 and the slots expressed in each turn, like the following:

QUERY DEPART_TIME	And what time would you like to leave [$_{\text{depart\_city}}$ **Pittsburgh**]?
QUERY ARRIVE_CITY	And you're flying into what city?
QUERY ARRIVE_TIME	What time on [$_{\text{arrive\_date}}$ **May 5**]?
INFORM FLIGHT	The flight departs [$_{\text{depart\_airport}}$ **PGH**] at [$_{\text{depart\_time}}$ **10 am**] and arrives [$_{\text{arrive\_city}}$ **Seattle**] at [$_{\text{arrive\_time}}$ **12:05 their time**].

This corpus is then delexicalized, and divided up into separate corpora for each dialog act. Thus the delexicalized corpus for one dialog act, QUERY DEPART_TIME might be trained on examples like:

And what time would you like to leave depart_city?
When would you like to leave depart_city?
When would you like to leave?
What time do you want to leave on depart_date?
OK, on depart_date, what time do you want to leave?

A distinct N-gram grammar is then trained for each dialog act. Now, given the dialog act QUERY DEPART_TIME, the system samples random sentences from this language model. Recall from the the "Shannon" exercise of 44 that this works (assuming a bigram LM) by first selecting a bigram ($< s >, < w >$) according to its bigram probability in the language model, then drawing a bigram starting with $< w >$ according to its bigram probability, and so on until a full sentence is generated. The probability of each successive word w_i being generated from utterance class u is thus

$$P(w_i) = P(w_i|w_{i-1}, w_{i-2}, ..., w_{i-(n-1)}, u) \tag{26.7}$$

Each of these randomly sampled sentences is then assigned a score based on heuristic rules that penalize sentences that are too short or too long, repeat slots, or lack some of the required slots from the input frame (in this case, depart_airport and depart_date). The best scoring sentence is then chosen. Let's suppose in this case we produce the following (delexicalized) sentence:

What time on depart_date would you like to leave depart_airport?

This sentence is then relexicalized from the true values in the input frame, resulting in the final sentence:

What time on October fifth would you like to leave Boston?

Modern implementations of the model replace the simplistic N-gram part of the generator with neural models, which similarly learn to map from an input frame to a resulting sentence (Wen et al. 2015a, Wen et al. 2015b).

clarification questions

It's also possible to design NLG algorithms that are specific to a particular dialog act. For example, consider the task of generating **clarification questions**, in cases where the speech recognition fails to understand some part of the user's utterance. While it is possible to use the generic dialog act REJECT ("Please repeat", or "I don't understand what you said"), studies of human conversations show that humans instead use targeted clarification questions that reprise elements of the misunderstanding (Purver 2004, Ginzburg and Sag 2000, Stoyanchev et al. 2013).

For example, in the following hypothetical example the system reprises the words "going" and "on the 5th" to make it clear which aspect of the user's turn the system needs to be clarified:

User:	What do you have going to UNKNOWN_WORD on the 5th?
System:	Going where on the 5th?

Targeted clarification questions can be created by rules (such as replacing "going to UNKNOWN_WORD" with "going where") or by building classifiers to guess which slots might have been misrecognized in the sentence (Chu-Carroll and Carpenter 1999, Stoyanchev et al. 2014, Stoyanchev and Johnston 2015).

26.6 Deep Reinforcement Learning for Dialog

TBD

26.7 Summary

- In dialog, speaking is a kind of action; these acts are referred to as speech acts. Speakers also attempt to achieve **common ground** by acknowledging that they have understand each other. The **dialog act** combines the intuition of speech acts and grounding acts.
- The **dialog-state** or information-state architecture augments the frame-and-slot state architecture by keeping track of user's dialog acts and includes a **policy** for generating its own dialog acts in return.
- Policies based on reinforcement learning architecture like the MDP and POMDP offer ways for future dialog reward to be propagated back to influence policy earlier in the dialog manager.

Bibliographical and Historical Notes

The idea that utterances in a conversation are a kind of **action** being performed by the speaker was due originally to the philosopher Wittgenstein (1953) but worked out more fully by Austin (1962) and his student John Searle. Various sets of speech acts have been defined over the years, and a rich linguistic and philosophical literature developed, especially focused on explaining the use of indirect speech acts.

The idea of dialog acts draws also from a number of other sources, including the ideas of adjacency pairs, pre-sequences, and other aspects of the international properties of human conversation developed in the field of conversation analysis (see Levinson (1983) for an introduction to the field).

This idea that acts set up strong local dialog expectations was also prefigured by Firth (1935, p. 70), in a famous quotation:

> *Most of the give-and-take of conversation in our everyday life is stereotyped and very narrowly conditioned by our particular type of culture. It is a sort of roughly prescribed social ritual, in which you generally say what the other fellow expects you, one way or the other, to say.*

Another important research thread modeled dialog as a kind of collaborative behavior, including the ideas of common ground (Clark and Marshall, 1981), reference as a collaborative process (Clark and Wilkes-Gibbs, 1986), joint intention (Levesque et al., 1990), and shared plans (Grosz and Sidner, 1980).

The information state model of dialog was also strongly informed by analytic work on the linguistic properties of dialog acts and on methods for their detection (Sag and Liberman 1975, Hinkelman and Allen 1989, Nagata and Morimoto 1994, Goodwin 1996, Chu-Carroll 1998, Shriberg et al. 1998, Stolcke et al. 2000, Gravano et al. 2012).

Two important lines of research focused on the computational properties of conversational structure. One line, first suggested at by Bruce (1975), suggested that since speech acts are actions, they should be planned like other actions, and drew on the AI planning literature (Fikes and Nilsson, 1971). An agent seeking to find out some information can come up with the plan of asking the interlocutor for the information. An agent hearing an utterance can interpret a speech act by running the planner "in reverse", using inference rules to infer from what the interlocutor said what the plan might have been. Plan-based models of dialog are referred to as

BDI **BDI** models because such planners model the **beliefs**, **desires**, and **intentions** (BDI) of the agent and interlocutor. BDI models of dialog were first introduced by Allen, Cohen, Perrault, and their colleagues in a number of influential papers showing how speech acts could be generated (Cohen and Perrault, 1979) and interpreted (Perrault and Allen 1980, Allen and Perrault 1980). At the same time, Wilensky (1983) introduced plan-based models of understanding as part of the task of interpreting stories.

Another influential line of research focused on modeling the hierarchical structure of dialog. Grosz's pioneering (1977) dissertation first showed that "task-oriented dialogs have a structure that closely parallels the structure of the task being performed" (p. 27), leading to her work with Sidner and others showing how to use similar notions of intention and plans to model discourse structure and coherence in dialog. See, e.g., Lochbaum et al. (2000) for a summary of the role of intentional structure in dialog.

The idea of applying reinforcement learning to dialog first came out of AT&T and Bell Laboratories around the turn of the century with work on MDP dialog systems (Walker 2000, Levin et al. 2000, Singh et al. 2002) and work on cue phrases, prosody, and rejection and confirmation. Reinforcement learning research turned quickly to the more sophisticated POMDP models (Roy et al. 2000, Lemon et al. 2006, Williams and Young 2007) applied to small slot-filling dialog tasks. [History of deep reinforcement learning here.]

to be continued

CHAPTER

27 Speech Recognition

Placeholder

CHAPTER

28 Speech Synthesis

Placeholder

Bibliography

Abbreviations:

AAAI	Proceedings of the National Conference on Artificial Intelligence
ACL	Proceedings of the Annual Conference of the Association for Computational Linguistics
ANLP	Proceedings of the Conference on Applied Natural Language Processing
CLS	Papers from the Annual Regional Meeting of the Chicago Linguistics Society
COGSCI	Proceedings of the Annual Conference of the Cognitive Science Society
COLING	Proceedings of the International Conference on Computational Linguistics
CoNLL	Proceedings of the Conference on Computational Natural Language Learning
EACL	Proceedings of the Conference of the European Association for Computational Linguistics
EMNLP	Proceedings of the Conference on Empirical Methods in Natural Language Processing
EUROSPEECH	Proceedings of the European Conference on Speech Communication and Technology
ICASSP	Proceedings of the IEEE International Conference on Acoustics, Speech, & Signal Processing
ICML	International Conference on Machine Learning
ICPhS	Proceedings of the International Congress of Phonetic Sciences
ICSLP	Proceedings of the International Conference on Spoken Language Processing
IJCAI	Proceedings of the International Joint Conference on Artificial Intelligence
INTERSPEECH	Proceedings of the Annual INTERSPEECH Conference
IWPT	Proceedings of the International Workshop on Parsing Technologies
JASA	Journal of the Acoustical Society of America
LREC	Conference on Language Resources and Evaluation
MUC	Proceedings of the Message Understanding Conference
NAACL-HLT	Proceedings of the North American Chapter of the ACL/Human Language Technology Conference
SIGIR	Proceedings of Annual Conference of ACM Special Interest Group on Information Retrieval

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems.. Software available from tensorflow.org.

Abney, S. P. (1991). Parsing by chunks. In Berwick, R. C., Abney, S. P., and Tenny, C. (Eds.), *Principle-Based Parsing: Computation and Psycholinguistics*, pp. 257–278. Kluwer.

Abney, S. P. (1997). Stochastic attribute-value grammars. *Computational Linguistics*, *23*(4), 597–618.

Abney, S. P. (2002). Bootstrapping. In *ACL-02*, pp. 360–367.

Abney, S. P. (2004). Understanding the Yarowsky algorithm. *Computational Linguistics*, *30*(3), 365–395.

Abney, S. P., Schapire, R. E., and Singer, Y. (1999). Boosting applied to tagging and PP attachment. In *EMNLP/VLC-99*, College Park, MD, pp. 38–45.

Adriaans, P. and van Zaanen, M. (2004). Computational grammar induction for linguists. *Grammars; special issue with the theme "Grammar Induction"*, *7*, 57–68.

Aggarwal, C. C. and Zhai, C. (2012). A survey of text classification algorithms. In Aggarwal, C. C. and Zhai, C. (Eds.), *Mining text data*, pp. 163–222. Springer.

Agichtein, E. and Gravano, L. (2000). Snowball: Extracting relations from large plain-text collections. In *Proceedings of the 5th ACM International Conference on Digital Libraries*.

Agirre, E. and de Lacalle, O. L. (2003). Clustering WordNet word senses. In *RANLP 2003*.

Agirre, E., Banea, C., Cardie, C., Cer, D., Diab, M., Gonzalez-Agirre, A., Guo, W., Lopez-Gazpio, I., Maritxalar, M., Mihalcea, R., Rigau, G., Uria, L., and Wiebe, J. (2015). 2015 SemEval-2015 Task 2: Semantic Textual Similarity, English, Spanish and Pilot on Interpretability. In *SemEval-15*, pp. 252–263.

Agirre, E., Diab, M., Cer, D., and Gonzalez-Agirre, A. (2012). Semeval-2012 task 6: A pilot on semantic textual similarity. In *SemEval-12*, pp. 385–393.

Agirre, E. and Edmonds, P. (Eds.). (2006). *Word Sense Disambiguation: Algorithms and Applications*. Kluwer.

Agirre, E., López de Lacalle, O., and Soroa, A. (2014). Random walks for knowledge-based word sense disambiguation. *Computational Linguistics*, *40*(1), 57–84.

Agirre, E. and Martinez, D. (2001). Learning class-to-class selectional preferences. In *CoNLL-01*.

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

Aho, A. V. and Ullman, J. D. (1972). *The Theory of Parsing, Translation, and Compiling*, Vol. 1. Prentice Hall.

Ajdukiewicz, K. (1935). Die syntaktische Konnexität. *Studia Philosophica*, *1*, 1–27. English translation "Syntactic Connexion" by H. Weber in McCall, S. (Ed.) 1967. *Polish Logic*, pp. 207–231, Oxford University Press.

Algoet, P. H. and Cover, T. M. (1988). A sandwich proof of the Shannon-McMillan-Breiman theorem. *The Annals of Probability*, *16*(2), 899–909.

Allen, J. (1984). Towards a general theory of action and time. *Artificial Intelligence*, *23*(2), 123–154.

Allen, J. and Perrault, C. R. (1980). Analyzing intention in utterances. *Artificial Intelligence*, *15*, 143–178.

Amsler, R. A. (1981). A taxonomy of English nouns and verbs. In *ACL-81*, Stanford, CA, pp. 133–138.

Artstein, R., Gandhe, S., Gerten, J., Leuski, A., and Traum, D. (2009). Semi-formal evaluation of conversational characters. In *Languages: From Formal to Natural*, pp. 22–35. Springer.

Atkins, S. (1993). Tools for computer-aided corpus lexicography: The Hector project. *Acta Linguistica Hungarica*, *41*, 5–72.

Austin, J. L. (1962). *How to Do Things with Words*. Harvard University Press.

Awadallah, A. H., Kulkarni, R. G., Ozertem, U., and Jones, R. (2015). Charaterizing and predicting voice query reformulation. In *CIKM-15*.

Baayen, R. H. (2001). *Word frequency distributions*. Springer.

Bacchiani, M., Riley, M., Roark, B., and Sproat, R. (2006). Map adaptation of stochastic grammars. *Computer Speech & Language*, *20*(1), 41–68.

Bacchiani, M., Roark, B., and Saraclar, M. (2004). Language model adaptation with MAP estimation and the perceptron algorithm. In *HLT-NAACL-04*, pp. 21–24.

Baccianella, S., Esuli, A., and Sebastiani, F. (2010). Sentiwordnet 3.0: An enhanced lexical resource for sentiment analysis and opinion mining.. In *LREC-10*, pp. 2200–2204.

Bach, K. and Harnish, R. (1979). *Linguistic communication and speech acts*. MIT Press.

Backus, J. W. (1959). The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In *Information Processing: Proceedings of the International Conference on Information Processing, Paris*, pp. 125–132. UNESCO.

Backus, J. W. (1996). Transcript of question and answer session. In Wexelblat, R. L. (Ed.), *History of Programming Languages*, p. 162. Academic Press.

Bahl, L. R. and Mercer, R. L. (1976). Part of speech assignment by a statistical decision algorithm. In *Proceedings IEEE International Symposium on Information Theory*, pp. 88–89.

Bahl, L. R., Jelinek, F., and Mercer, R. L. (1983). A maximum likelihood approach to continuous speech recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *5*(2), 179–190.

Baker, C. F., Fillmore, C. J., and Lowe, J. B. (1998). The Berkeley FrameNet project. In *COLING/ACL-98*, Montreal, Canada, pp. 86–90.

Baker, J. K. (1975). The DRAGON system – An overview. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, *ASSP-23*(1), 24–29.

Baker, J. K. (1975/1990). Stochastic modeling for automatic speech understanding. In Waibel, A. and Lee, K.-F. (Eds.), *Readings in Speech Recognition*, pp. 297–307. Morgan Kaufmann. Originally appeared in *Speech Recognition*, Academic Press, 1975.

Baker, J. K. (1979). Trainable grammars for speech recognition. In Klatt, D. H. and Wolf, J. J. (Eds.), *Speech Communication Papers for the 97th Meeting of the Acoustical Society of America*, pp. 547–550.

Banerjee, S. and Pedersen, T. (2003). Extended gloss overlaps as a measure of semantic relatedness. In *IJCAI 2003*, pp. 805–810.

Bangalore, S. and Joshi, A. K. (1999). Supertagging: An approach to almost parsing. *Computational Linguistics*, *25*(2), 237–265.

Banko, M., Cafarella, M., Soderland, S., Broadhead, M., and Etzioni, O. (2007). Open information extraction for the web. In *IJCAI*, Vol. 7, pp. 2670–2676.

Bar-Hillel, Y. (1953). A quasi-arithmetical notation for syntactic description. *Language*, *29*, 47–58. Reprinted in Y. Bar-Hillel. (1964). *Language and Information: Selected Essays on their Theory and Application*, Addison-Wesley, 61–74.

Baum, L. F. (1900). *The Wizard of Oz*. Available at Project Gutenberg.

Bayes, T. (1763). *An Essay Toward Solving a Problem in the Doctrine of Chances*, Vol. 53. Reprinted in *Facsimiles of Two Papers by Bayes*, Hafner Publishing, 1963.

Bazell, C. E. (1952/1966). The correspondence fallacy in structural linguistics. In Hamp, E. P., Householder, F. W., and Austerlitz, R. (Eds.), *Studies by Members of the English Department, Istanbul University (3), reprinted in Readings in Linguistics II (1966)*, pp. 271–298. University of Chicago Press.

Bejček, E., Hajičová, E., Hajič, J., Jínová, P., Kettnerová, V., Kolářová, V., Mikulová, M., Mírovský, J., Nedoluzhko, A., Panevová, J., Poláková, L., Ševčíková, M., Štěpánek, J., and Zikánová, Š. (2013). Prague dependency treebank 3.0. Tech. rep., Institute of Formal and Applied Linguistics, Charles University in Prague. LINDAT/CLARIN digital library at Institute of Formal and Applied Linguistics, Charles University in Prague.

Bellegarda, J. R. (1997). A latent semantic analysis framework for large-span language modeling. In *Eurospeech-97*, Rhodes, Greece.

Bellegarda, J. R. (2000). Exploiting latent semantic information in statistical language modeling. *Proceedings of the IEEE*, *89*(8), 1279–1296.

Bellegarda, J. R. (2004). Statistical language model adaptation: Review and perspectives. *Speech Communication*, *42*(1), 93–108.

Bellegarda, J. R. (2013). Natural language technology in mobile devices: Two grounding frameworks. In *Mobile Speech and Advanced Natural Language Solutions*, pp. 185–196. Springer.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press.

Bellman, R. (1984). *Eye of the Hurricane: an autobiography*. World Scientific Singapore.

Bengio, Y., Courville, A., and Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *35*(8), 1798–1828.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research*, *3*(Feb), 1137–1155.

Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007). Greedy layer-wise training of deep networks. In *NIPS 2007*, pp. 153–160.

Bengio, Y., Schwenk, H., Senécal, J.-S., Morin, F., and Gauvain, J.-L. (2006). Neural probabilistic language models. In *Innovations in Machine Learning*, pp. 137–186. Springer.

Berant, J., Chou, A., Frostig, R., and Liang, P. (2013). Semantic parsing on freebase from question-answer pairs. In *EMNLP 2013*.

Berant, J. and Liang, P. (2014). Semantic parsing via paraphrasing. In *ACL 2014*.

Berg-Kirkpatrick, T., Burkett, D., and Klein, D. (2012). An empirical investigation of statistical significance in NLP. In *EMNLP 2012*, pp. 995–1005.

Berger, A., Della Pietra, S. A., and Della Pietra, V. J. (1996). A maximum entropy approach to natural language processing. *Computational Linguistics*, *22*(1), 39–71.

Bergsma, S., Lin, D., and Goebel, R. (2008). Discriminative learning of selectional preference from unlabeled text. In *EMNLP-08*, pp. 59–68.

Bethard, S. (2013). ClearTK-TimeML: A minimalist approach to TempEval 2013. In *SemEval-13*, pp. 10–14.

Bever, T. G. (1970). The cognitive basis for linguistic structures. In Hayes, J. R. (Ed.), *Cognition and the Development of Language*, pp. 279–352. Wiley.

Bhat, I., Bhat, R. A., Shrivastava, M., and Sharma, D. (2017). Joining hands: Exploiting monolingual treebanks for parsing of code-mixing data. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pp. 324–330. Association for Computational Linguistics.

Biber, D., Johansson, S., Leech, G., Conrad, S., and Finegan, E. (1999). *Longman Grammar of Spoken and Written English*. Pearson ESL, Harlow.

Bies, A., Ferguson, M., Katz, K., and MacIntyre, R. (1995). Bracketing guidelines for Treebank II style Penn Treebank Project..

Bikel, D. M. (2004). Intricacies of Collins' parsing model. *Computational Linguistics*, *30*(4), 479–511.

Bikel, D. M., Miller, S., Schwartz, R., and Weischedel, R. (1997). Nymble: A high-performance learning namefinder. In *ANLP 1997*, pp. 194–201.

Biran, O., Brody, S., and Elhadad, N. (2011). Putting it simply: a context-aware approach to lexical simplification. In *ACL 2011*, pp. 496–501.

Bird, S., Klein, E., and Loper, E. (2009). *Natural Language Processing with Python*. O'Reilly.

Bisani, M. and Ney, H. (2004). Bootstrap estimates for confidence intervals in ASR performance evaluation. In *ICASSP-04*, Vol. I, pp. 409–412.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.

Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., and Hellmann, S. (2009). DBpedia—A crystallization point for the Web of Data. *Web Semantics: science, services and agents on the world wide web*, *7*(3), 154–165.

Black, E. (1988). An experiment in computational discrimination of English word senses. *IBM Journal of Research and Development*, *32*(2), 185–194.

Black, E., Abney, S. P., Flickinger, D., Gdaniec, C., Grishman, R., Harrison, P., Hindle, D., Ingria, R., Jelinek, F., Klavans, J. L., Liberman, M. Y., Marcus, M. P., Roukos, S., Santorini, B., and Strzalkowski, T. (1991). A procedure for quantitatively comparing the syntactic coverage of English grammars. In *Proceedings DARPA Speech and Natural Language Workshop*, Pacific Grove, CA, pp. 306–311.

Black, E., Jelinek, F., Lafferty, J. D., Magerman, D. M., Mercer, R. L., and Roukos, S. (1992). Towards history-based grammars: Using richer models for probabilistic parsing. In *Proceedings DARPA Speech and Natural Language Workshop*, Harriman, NY, pp. 134–139.

Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent Dirichlet allocation. *Journal of Machine Learning Research*, *3*(5), 993–1022.

Blodgett, S. L., Green, L., and O'Connor, B. (2016). Demographic dialectal variation in social media: A case study of African-American English. In *EMNLP 2016*.

Blodgett, S. L. and O'Connor, B. (2017). Racial disparity in natural language processing: A case study of social media african-american english. In *Fairness, Accountability, and Transparency in Machine Learning (FAT/ML) Workshop, KDD*.

Bloomfield, L. (1914). *An Introduction to the Study of Language*. Henry Holt and Company.

Bloomfield, L. (1933a). *Language*. University of Chicago Press.

Bloomfield, L. (1933b). *Language*. University of Chicago Press.

Bobrow, D. G., Kaplan, R. M., Kay, M., Norman, D. A., Thompson, H., and Winograd, T. (1977). GUS, A frame driven dialog system. *Artificial Intelligence*, *8*, 155–173.

Bobrow, D. G. and Norman, D. A. (1975). Some principles of memory schemata. In Bobrow, D. G. and Collins, A. (Eds.), *Representation and Understanding*. Academic Press.

Bod, R. (1993). Using an annotated corpus as a stochastic grammar. In *EACL-93*, pp. 37–44.

Boguraev, B. and Briscoe, T. (Eds.). (1989). *Computational Lexicography for Natural Language Processing*. Longman.

Bohus, D. and Rudnicky, A. I. (2005). Sorry, I didn't catch that! — An investigation of non-understanding errors and recovery strategies. In *Proceedings of SIGDIAL*, Lisbon, Portugal.

Bollacker, K., Evans, C., Paritosh, P., Sturge, T., and Taylor, J. (2008). Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD 2008*, pp. 1247–1250.

Bolukbasi, T., Chang, K.-W., Zou, J. Y., Saligrama, V., and Kalai, A. T. (2016). Man is to computer programmer as woman is to homemaker? debiasing word embeddings. In *Advances in Neural Information Processing Systems*, pp. 4349–4357.

Booth, T. L. (1969). Probabilistic representation of formal languages. In *IEEE Conference Record of the 1969 Tenth Annual Symposium on Switching and Automata Theory*, pp. 74–81.

Booth, T. L. and Thompson, R. A. (1973). Applying probability measures to abstract languages. *IEEE Transactions on Computers*, *C-22*(5), 442–450.

Borges, J. L. (1964). *The analytical language of John Wilkins*. University of Texas Press. Trans. Ruth L. C. Simms.

Bowman, S. R., Vilnis, L., Vinyals, O., Dai, A. M., Jozefowicz, R., and Bengio, S. (2016). Generating sentences from a continuous space. In *CoNLL-16*, pp. 10–21.

Boyd-Graber, J., Blei, D. M., and Zhu, X. (2007). A topic model for word sense disambiguation. In *EMNLP/CoNLL 2007*.

Brants, T. (2000). TnT: A statistical part-of-speech tagger. In *ANLP 2000*, Seattle, WA, pp. 224–231.

Brants, T., Popat, A. C., Xu, P., Och, F. J., and Dean, J. (2007). Large language models in machine translation. In *EMNLP/CoNLL 2007*.

Bréal, M. (1897). *Essai de Sémantique: Science des significations*. Hachette, Paris, France.

Bresnan, J. (Ed.). (1982). *The Mental Representation of Grammatical Relations*. MIT Press.

Brill, E., Dumais, S. T., and Banko, M. (2002). An analysis of the AskMSR question-answering system. In *EMNLP 2002*, pp. 257–264.

Brill, E. and Resnik, P. (1994). A rule-based approach to prepositional phrase attachment disambiguation. In *COLING-94*, Kyoto, pp. 1198–1204.

Brin, S. (1998). Extracting patterns and relations from the World Wide Web. In *Proceedings World Wide Web and Databases International Workshop, Number 1590 in LNCS*, pp. 172–183. Springer.

Briscoe, T. and Carroll, J. (1993). Generalized probabilistic LR parsing of natural language (corpora) with unification-based grammars. *Computational Linguistics*, *19*(1), 25–59.

Brockmann, C. and Lapata, M. (2003). Evaluating and combining approaches to selectional preference acquisition. In *EACL-03*, pp. 27–34.

Brody, S. and Lapata, M. (2009). Bayesian word sense induction. In *EACL-09*, pp. 103–111.

Broschart, J. (1997). Why Tongan does it differently. *Linguistic Typology*, *1*, 123–165.

Bruce, B. C. (1975). Generation as a social action. In *Proceedings of TINLAP-1 (Theoretical Issues in Natural Language Processing)*, pp. 64–67. Association for Computational Linguistics.

Bruce, R. F. and Wiebe, J. (1994). Word-sense disambiguation using decomposable models. In *ACL-94*, Las Cruces, NM, pp. 139–145.

Brysbaert, M., Warriner, A. B., and Kuperman, V. (2014). Concreteness ratings for 40 thousand generally known english word lemmas. *Behavior Research Methods*, *46*(3), 904–911.

Buchholz, S. and Marsi, E. (2006). Conll-x shared task on multilingual dependency parsing. In *In Proc. of CoNLL*, pp. 149–164.

Buck, C., Heafield, K., and Van Ooyen, B. (2014). N-gram counts and language models from the common crawl. In *Proceedings of LREC*.

Budanitsky, A. and Hirst, G. (2006). Evaluating WordNet-based measures of lexical semantic relatedness. *Computational Linguistics*, *32*(1), 13–47.

Bullinaria, J. A. and Levy, J. P. (2007). Extracting semantic representations from word co-occurrence statistics: A computational study. *Behavior research methods*, *39*(3), 510–526.

Bullinaria, J. A. and Levy, J. P. (2012). Extracting semantic representations from word co-occurrence statistics: stop-lists, stemming, and svd. *Behavior research methods*, *44*(3), 890–907.

Bulyko, I., Kirchhoff, K., Ostendorf, M., and Goldberg, J. (2005). Error-sensitive response generation in a spoken language dialogue system. *Speech Communication*, *45*(3), 271–288.

Bulyko, I., Ostendorf, M., and Stolcke, A. (2003). Getting more mileage from web text sources for conversational speech language modeling using class-dependent mixtures. In *HLT-NAACL-03*, Edmonton, Canada, Vol. 2, pp. 7–9.

Cai, Q. and Yates, A. (2013). Large-scale semantic parsing via schema matching and lexicon extension.. In *ACL 2013*, pp. 423–433.

Caliskan, A., Bryson, J. J., and Narayanan, A. (2017). Semantics derived automatically from language corpora contain human-like biases. *Science*, *356*(6334), 183–186.

Cardie, C. (1993). A case-based approach to knowledge acquisition for domain specific sentence analysis. In *AAAI-93*, pp. 798–803. AAAI Press.

Cardie, C. (1994). *Domain-Specific Knowledge Acquisition for Conceptual Sentence Analysis*. Ph.D. thesis, University of Massachusetts, Amherst, MA. Available as CMPSCI Technical Report 94-74.

Carletta, J., Isard, A., Isard, S., Kowtko, J. C., Doherty-Sneddon, G., and Anderson, A. H. (1997). The reliability of a dialogue structure coding scheme. *Computational Linguistics*, *23*(1), 13–32.

Carpenter, R. (2017). Cleverbot. http://www.cleverbot.com, accessed 2017.

Carpuat, M. and Wu, D. (2007). Improving statistical machine translation using word sense disambiguation. In *EMNLP/CoNLL 2007*, Prague, Czech Republic, pp. 61–72.

Carreras, X. and Màrquez, L. (2005). Introduction to the CoNLL-2005 shared task: Semantic role labeling. In *CoNLL-05*, pp. 152–164.

Carroll, G. and Charniak, E. (1992). Two experiments on learning probabilistic dependency grammars from corpora. Tech. rep. CS-92-16, Brown University.

Carroll, J., Briscoe, T., and Sanfilippo, A. (1998). Parser evaluation: A survey and a new proposal. In *LREC-98*, Granada, Spain, pp. 447–454.

Chambers, N. (2013). NavyTime: Event and time ordering from raw text. In *SemEval-13*, pp. 73–77.

Chambers, N. and Jurafsky, D. (2010). Improving the use of pseudo-words for evaluating selectional preferences. In *ACL 2010*, pp. 445–453.

Chambers, N. and Jurafsky, D. (2011). Template-based information extraction without the templates. In *ACL 2011*.

Chan, Y. S., Ng, H. T., and Chiang, D. (2007). Word sense disambiguation improves statistical machine translation. In *ACL-07*, Prague, Czech Republic, pp. 33–40.

Chang, A. X. and Manning, C. D. (2012). SUTime: A library for recognizing and normalizing time expressions.. In *LREC-12*, pp. 3735–3740.

Chang, P.-C., Galley, M., and Manning, C. D. (2008). Optimizing Chinese word segmentation for machine translation performance. In *Proceedings of ACL Statistical MT Workshop*, pp. 224–232.

Charniak, E. (1997). Statistical parsing with a context-free grammar and word statistics. In *AAAI-97*, pp. 598–603. AAAI Press.

Charniak, E., Hendrickson, C., Jacobson, N., and Perkowitz, M. (1993). Equations for part-of-speech tagging. In *AAAI-93*, Washington, D.C., pp. 784–789. AAAI Press.

Charniak, E. and Johnson, M. (2005). Coarse-to-fine *n*-best parsing and MaxEnt discriminative reranking. In *ACL-05*, Ann Arbor.

Che, W., Li, Z., Li, Y., Guo, Y., Qin, B., and Liu, T. (2009). Multilingual dependency-based syntactic and semantic parsing. In *CoNLL-09*, pp. 49–54.

Chelba, C. and Jelinek, F. (2000). Structured language modeling. *Computer Speech and Language*, *14*, 283–332.

Chen, D. and Manning, C. D. (2014). A fast and accurate dependency parser using neural networks.. In *EMNLP*, pp. 740–750.

Chen, J. N. and Chang, J. S. (1998). Topical clustering of MRD senses based on information retrieval techniques. *Computational Linguistics*, *24*(1), 61–96.

Chen, S. F. and Goodman, J. (1996). An empirical study of smoothing techniques for language modeling. In *ACL-96*, Santa Cruz, CA, pp. 310–318.

Chen, S. F. and Goodman, J. (1998). An empirical study of smoothing techniques for language modeling. Tech. rep. TR-10-98, Computer Science Group, Harvard University.

Chen, S. F. and Goodman, J. (1999). An empirical study of smoothing techniques for language modeling. *Computer Speech and Language*, *13*, 359–394.

Chinchor, N., Hirschman, L., and Lewis, D. L. (1993). Evaluating Message Understanding systems: An analysis of the third Message Understanding Conference. *Computational Linguistics*, *19*(3), 409–449.

Chiticariu, L., Danilevsky, M., Li, Y., Reiss, F., and Zhu, H. (2018). SystemT: Declarative text understanding for enterprise. In *NAACL HLT 2018*, Vol. 3, pp. 76–83.

Chiticariu, L., Li, Y., and Reiss, F. R. (2013). Rule-Based Information Extraction is Dead! Long Live Rule-Based Information Extraction Systems!. In *EMNLP 2013*, pp. 827–832.

Chklovski, T. and Mihalcea, R. (2003). Exploiting agreement and disagreement of human annotators for word sense disambiguation. In *RANLP 2003*.

Choi, J. D. and Palmer, M. (2011a). Getting the most out of transition-based dependency parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers-Volume 2*, pp. 687–692. Association for Computational Linguistics.

Choi, J. D. and Palmer, M. (2011b). Transition-based semantic role labeling using predicate argument clustering. In *Proceedings of the ACL 2011 Workshop on Relational Models of Semantics*, pp. 37–45. Association for Computational Linguistics.

Choi, J. D., Tetreault, J., and Stent, A. (2015). It depends: Dependency parser comparison using a web-based evaluation tool. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL*, pp. 26–31.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, *2*(3), 113–124.

Chomsky, N. (1956/1975). *The Logical Structure of Linguistic Theory*. Plenum.

Chomsky, N. (1957). *Syntactic Structures*. Mouton, The Hague.

Chomsky, N. (1963). Formal properties of grammars. In Luce, R. D., Bush, R., and Galanter, E. (Eds.), *Handbook of Mathematical Psychology*, Vol. 2, pp. 323–418. Wiley.

Chomsky, N. (1981). *Lectures on Government and Binding*. Foris.

Christodoulopoulos, C., Goldwater, S., and Steedman, M. (2010). Two decades of unsupervised POS induction: How far have we come?. In *EMNLP-10*.

Chu, Y.-J. and Liu, T.-H. (1965). On the shortest arborescence of a directed graph. *Science Sinica*, *14*, 1396–1400.

Chu-Carroll, J. (1998). A statistical model for discourse act recognition in dialogue interactions. In Chu-Carroll, J. and Green, N. (Eds.), *Applying Machine Learning to Discourse Processing. Papers from the 1998 AAAI Spring Symposium.* Tech. rep. SS-98-01, pp. 12–17. AAAI Press.

Chu-Carroll, J. and Carpenter, B. (1999). Vector-based natural language call routing. *Computational Linguistics*, *25*(3), 361–388.

Chu-Carroll, J., Fan, J., Boguraev, B. K., Carmel, D., Sheinwald, D., and Welty, C. (2012). Finding needles in the haystack: Search and candidate generation. *IBM Journal of Research and Development*, *56*(3/4), 6:1–6:12.

Church, K. W. (1980). *On Memory Limitations in Natural Language Processing* Master's thesis, MIT. Distributed by the Indiana University Linguistics Club.

Church, K. W. (1988). A stochastic parts program and noun phrase parser for unrestricted text. In *ANLP 1988*, pp. 136–143.

Church, K. W. (1989). A stochastic parts program and noun phrase parser for unrestricted text. In *ICASSP-89*, pp. 695–698.

Church, K. W. (1994). Unix for Poets. Slides from 2nd ELSNET Summer School and unpublished paper ms.

Church, K. W. and Gale, W. A. (1991). A comparison of the enhanced Good-Turing and deleted estimation methods for estimating probabilities of English bigrams. *Computer Speech and Language*, *5*, 19–54.

Church, K. W. and Hanks, P. (1989). Word association norms, mutual information, and lexicography. In *ACL-89*, Vancouver, B.C., pp. 76–83.

Church, K. W., Hart, T., and Gao, J. (2007). Compressing trigram language models with Golomb coding. In *EMNLP/CoNLL 2007*, pp. 199–207.

Clark, A. (2001). The unsupervised induction of stochastic context-free grammars using distributional clustering. In *CoNLL-01*.

Clark, E. (1987). The principle of contrast: A constraint on language acquisition. In MacWhinney, B. (Ed.), *Mechanisms of language acquisition*, pp. 1–33. LEA.

Clark, H. H. (1996). *Using Language*. Cambridge University Press.

Clark, H. H. and Fox Tree, J. E. (2002). Using uh and um in spontaneous speaking. *Cognition*, *84*, 73–111.

Clark, H. H. and Marshall, C. (1981). Definite reference and mutual knowledge. In Joshi, A. K., Webber, B. L., and Sag, I. A. (Eds.), *Elements of Discourse Understanding*, pp. 10–63. Cambridge.

Clark, H. H. and Schaefer, E. F. (1989). Contributing to discourse. *Cognitive Science*, *13*, 259–294.

Clark, H. H. and Wilkes-Gibbs, D. (1986). Referring as a collaborative process. *Cognition*, *22*, 1–39.

Clark, S. and Curran, J. R. (2004). Parsing the WSJ using CCG and log-linear models. In *ACL-04*, pp. 104–111.

Clark, S., Curran, J. R., and Osborne, M. (2003). Bootstrapping pos taggers using unlabelled data. In *CoNLL-03*, pp. 49–55.

Coccaro, N. and Jurafsky, D. (1998). Towards better integration of semantic predictors in statistical language modeling. In *ICSLP-98*, Sydney, Vol. 6, pp. 2403–2406.

Cohen, K. B. and Demner-Fushman, D. (2014). *Biomedical natural language processing*. Benjamins.

Cohen, M. H., Giangola, J. P., and Balogh, J. (2004). *Voice User Interface Design*. Addison-Wesley.

Cohen, P. R. and Perrault, C. R. (1979). Elements of a plan-based theory of speech acts. *Cognitive Science*, *3*(3), 177–212.

Colby, K. M., Hilf, F. D., Weber, S., and Kraemer, H. C. (1972). Turing-like indistinguishability tests for the validation of a computer simulation of paranoid processes. *Artificial Intelligence*, *3*, 199–221.

Colby, K. M., Weber, S., and Hilf, F. D. (1971). Artificial paranoia. *Artificial Intelligence*, *2*(1), 1–25.

Cole, R. A., Novick, D. G., Vermeulen, P. J. E., Sutton, S., Fanty, M., Wessels, L. F. A., de Villiers, J. H., Schalkwyk, J., Hansen, B., and Burnett, D. (1997). Experiments with a spoken dialogue system for taking the US census. *Speech Communication*, *23*, 243–260.

Collins, M. (1996). A new statistical parser based on bigram lexical dependencies. In *ACL-96*, Santa Cruz, CA, pp. 184–191.

Collins, M. (1999). *Head-Driven Statistical Models for Natural Language Parsing*. Ph.D. thesis, University of Pennsylvania, Philadelphia.

Collins, M. (2000). Discriminative reranking for natural language parsing. In *ICML 2000*, Stanford, CA, pp. 175–182.

Collins, M. (2003a). Head-driven statistical models for natural language parsing. *Computational Linguistics*, *29*(4), 589–637.

Collins, M. (2003b). Head-driven statistical models for natural language parsing. *Computational Linguistics*, *29*(4), 589–637.

Collins, M., Hajič, J., Ramshaw, L. A., and Tillmann, C. (1999). A statistical parser for Czech. In *ACL-99*, College Park, MA, pp. 505–512.

Collins, M. and Koo, T. (2005). Discriminative reranking for natural language parsing. *Computational Linguistics*, *31*(1), 25–69.

Collobert, R. and Weston, J. (2007). Fast semantic extraction using a novel neural network architecture. In *ACL-07*, pp. 560–567.

Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ICML*, pp. 160–167.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, *12*, 2493–2537.

Copestake, A. and Briscoe, T. (1995). Semi-productive polysemy and sense extension. *Journal of Semantics*, *12*(1), 15–68.

Cottrell, G. W. (1985). *A Connectionist Approach to Word Sense Disambiguation*. Ph.D. thesis, University of Rochester, Rochester, NY. Revised version published by Pitman, 1989.

Cover, T. M. and Thomas, J. A. (1991). *Elements of Information Theory*. Wiley.

Covington, M. (2001). A fundamental algorithm for dependency parsing. In *Proceedings of the 39th Annual ACM Southeast Conference*, pp. 95–102.

Cox, D. (1969). *Analysis of Binary Data*. Chapman and Hall, London.

Craven, M. and Kumlien, J. (1999). Constructing biological knowledge bases by extracting information from text sources. In *ISMB-99*, pp. 77–86.

Cruse, D. A. (2004). *Meaning in Language: an Introduction to Semantics and Pragmatics*. Oxford University Press. Second edition.

Culicover, P. W. and Jackendoff, R. (2005). *Simpler Syntax*. Oxford University Press.

Dagan, I., Marcus, S., and Markovitch, S. (1993). Contextual word similarity and estimation from sparse data. In *ACL-93*, Columbus, Ohio, pp. 164–171.

Danieli, M. and Gerbino, E. (1995). Metrics for evaluating dialogue strategies in a spoken language system. In *Proceedings of the 1995 AAAI Spring Symposium on Empirical Methods in Discourse Interpretation and Generation*, Stanford, CA, pp. 34–39. AAAI Press.

Das, S. R. and Chen, M. Y. (2001). Yahoo! for Amazon: Sentiment parsing from small talk on the web. EFA 2001 Barcelona Meetings. Available at SSRN: http://ssrn.com/abstract=276189.

Davidson, T., Warmsley, D., Macy, M., and Weber, I. (2017). Automated hate speech detection and the problem of offensive language. In *ICWSM 2017*.

Davies, M. (2012). Expanding horizons in historical linguistics with the 400-million word Corpus of Historical American English. *Corpora*, *7*(2), 121–157.

de Marneffe, M.-C., Dozat, T., Silveira, N., Haverinen, K., Ginter, F., Nivre, J., and Manning, C. D. (2014). Universal stanford dependencies: A cross-linguistic typology.. In *LREC*, Vol. 14, pp. 4585–92.

de Marneffe, M.-C., MacCartney, B., and Manning, C. D. (2006). Generating typed dependency parses from phrase structure parses. In *LREC-06*.

de Marneffe, M.-C. and Manning, C. D. (2008). The stanford typed dependencies representation. In *Coling 2008: Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation*, pp. 1–8. Association for Computational Linguistics.

Deerwester, S. C., Dumais, S. T., Furnas, G. W., Harshman, R. A., Landauer, T. K., Lochbaum, K. E., and Streeter, L. (1988). Computer information retrieval using latent semantic structure: Us patent 4,839,853..

Deerwester, S. C., Dumais, S. T., Landauer, T. K., Furnas, G. W., and Harshman, R. A. (1990). Indexing by latent semantics analysis. *JASIS*, *41*(6), 391–407.

DeJong, G. F. (1982). An overview of the FRUMP system. In Lehnert, W. G. and Ringle, M. H. (Eds.), *Strategies for Natural Language Processing*, pp. 149–176. Lawrence Erlbaum.

DeRose, S. J. (1988). Grammatical category disambiguation by statistical optimization. *Computational Linguistics*, *14*, 31–39.

Di Marco, A. and Navigli, R. (2013). Clustering and diversifying web search results with graph-based word sense induction. *Computational Linguistics*, *39*(3), 709–754.

Diab, M. and Resnik, P. (2002). An unsupervised method for word sense tagging using parallel corpora. In *ACL-02*, pp. 255–262.

Digman, J. M. (1990). Personality structure: Emergence of the five-factor model. *Annual Review of Psychology*, *41*(1), 417–440.

Do, Q. N. T., Bethard, S., and Moens, M.-F. (2017). Improving implicit semantic role labeling by predicting semantic frame arguments. In *IJCNLP-17*.

Dolan, W. B. (1994). Word sense ambiguation: Clustering related senses. In *COLING-94*, Kyoto, Japan, pp. 712–716.

dos Santos, C., Xiang, B., and Zhou, B. (2015). Classifying relations by ranking with convolutional neural networks. In *ACL 2015*.

Dowty, D. R. (1979). *Word Meaning and Montague Grammar*. D. Reidel.

Dozat, T., Qi, P., and Manning, C. D. (2017). Stanford's graph-based neural dependency parser at the conll 2017 shared task. In *Proceedings of the CoNLL 2017 Shared Task*, pp. 20–30. Association for Computational Linguistics.

Dror, R., Baumer, G., Bogomolov, M., and Reichart, R. (2017). Replicability analysis for natural language processing: Testing significance with multiple datasets. *TACL*, *5*, 471—486.

Duda, R. O. and Hart, P. E. (1973). *Pattern Classification and Scene Analysis*. John Wiley and Sons.

Earley, J. (1968). *An Efficient Context-Free Parsing Algorithm*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA.

Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, *6*(8), 451–455. Reprinted in Grosz et al. (1986).

Edmonds, J. (1967). Optimum branchings. *Journal of Research of the National Bureau of Standards B*, *71*(4), 233–240.

Efron, B. and Tibshirani, R. J. (1993). *An introduction to the bootstrap*. CRC press.

Egghe, L. (2007). Untangling Herdan's law and Heaps' law: Mathematical and informetric arguments. *JASIST*, *58*(5), 702–709.

Eisner, J. (1996). Three new probabilistic models for dependency parsing: An exploration. In *COLING-96*, Copenhagen, pp. 340–345.

Ejerhed, E. I. (1988). Finding clauses in unrestricted text by finitary and stochastic methods. In *ANLP 1988*, pp. 219–227.

Ekman, P. (1999). Basic emotions. In Dalgleish, T. and Power, M. J. (Eds.), *Handbook of Cognition and Emotion*, pp. 45–60. Wiley.

Elman, J. L. (1990). Finding structure in time. *Cognitive science*, *14*(2), 179–211.

Erk, K. (2007). A simple, similarity-based model for selectional preferences. In *ACL-07*, pp. 216–223.

Etzioni, O., Cafarella, M., Downey, D., Popescu, A.-M., Shaked, T., Soderland, S., Weld, D. S., and Yates, A. (2005). Unsupervised named-entity extraction from the web: An experimental study. *Artificial Intelligence*, *165*(1), 91–134.

Evans, N. (2000). Word classes in the world's languages. In Booij, G., Lehmann, C., and Mugdan, J. (Eds.), *Morphology: A Handbook on Inflection and Word Formation*, pp. 708–732. Mouton.

Fader, A., Soderland, S., and Etzioni, O. (2011). Identifying relations for open information extraction. In *EMNLP-11*, pp. 1535–1545.

Fader, A., Zettlemoyer, L., and Etzioni, O. (2013). Paraphrase-driven learning for open question answering. In *ACL 2013*, Sofia, Bulgaria, pp. 1608–1618.

Fano, R. M. (1961). *Transmission of Information: A Statistical Theory of Communications*. MIT Press.

Feldman, J. A. and Ballard, D. H. (1982). Connectionist models and their properties. *Cognitive Science*, *6*, 205–254.

Fellbaum, C. (Ed.). (1998). *WordNet: An Electronic Lexical Database*. MIT Press.

Ferro, L., Gerber, L., Mani, I., Sundheim, B., and Wilson, G. (2005). Tides 2005 standard for the annotation of temporal expressions. Tech. rep., MITRE.

Ferrucci, D. A. (2012). Introduction to "this is watson". *IBM Journal of Research and Development*, *56*(3/4), 1:1–1:15.

Fessler, L. (2017). We tested bots like Siri and Alexa to see who would stand up to sexual harassment. In *Quartz*. Feb 22, 2017. https://qz.com/911681/.

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*, 189–208.

Fillmore, C. J. (1966). A proposal concerning english prepositions. In Dinneen, F. P. (Ed.), *17th annual Round Table.*, Vol. 17 of *Monograph Series on Language and Linguistics*, pp. 19–34. Georgetown University Press, Washington D.C.

Fillmore, C. J. (1968). The case for case. In Bach, E. W. and Harms, R. T. (Eds.), *Universals in Linguistic Theory*, pp. 1–88. Holt, Rinehart & Winston.

Fillmore, C. J. (1985). Frames and the semantics of understanding. *Quaderni di Semantica*, *VI*(2), 222–254.

Fillmore, C. J. (2003). Valency and semantic roles: the concept of deep structure case. In Ágel, V., Eichinger, L. M., Eroms, H. W., Hellwig, P., Heringer, H. J., and Lobin, H. (Eds.), *Dependenz und Valenz: Ein internationales Handbuch der zeitgenössischen Forschung*, chap. 36, pp. 457–475. Walter de Gruyter.

Fillmore, C. J. (2012). Encounters with language. *Computational Linguistics*, *38*(4), 701–718.

Fillmore, C. J. and Baker, C. F. (2009). A frames approach to semantic analysis. In Heine, B. and Narrog, H. (Eds.), *The Oxford Handbook of Linguistic Analysis*, pp. 313–340. Oxford University Press.

Fillmore, C. J., Johnson, C. R., and Petruck, M. R. L. (2003). Background to FrameNet. *International journal of lexicography*, *16*(3), 235–250.

Finkelstein, L., Gabrilovich, E., Matias, Y., Rivlin, E., Solan, Z., Wolfman, G., and Ruppin, E. (2002). Placing search in context: The concept revisited. *ACM Transactions on Information Systems*, *20*(1), 116—131.

Firth, J. R. (1935). The technique of semantics. *Transactions of the philological society*, *34*(1), 36–73.

Firth, J. R. (1957). A synopsis of linguistic theory 1930–1955. In *Studies in Linguistic Analysis*. Philological Society. Reprinted in Palmer, F. (ed.) 1968. Selected Papers of J. R. Firth. Longman, Harlow.

Foland, Jr., W. R. and Martin, J. H. (2015). Dependency-based semantic role labeling using convolutional neural networks. In **SEM 2015)*, pp. 279–289.

Forbes-Riley, K. and Litman, D. J. (2011). Benefits and challenges of real-time uncertainty detection and adaptation in a spoken dialogue computer tutor. *Speech Communication*, *53*(9), 1115–1136.

Forchini, P. (2013). Using movie corpora to explore spoken American English: Evidence from multi-dimensional analysis. In Bamford, J., Cavalieri, S., and Diani, G. (Eds.), *Variation and Change in Spoken and Written Discourse: Perspectives from corpus linguistics*, pp. 123–136. Benjamins.

Francis, H. S., Gregory, M. L., and Michaelis, L. A. (1999). Are lexical subjects deviant?. In *CLS-99*. University of Chicago.

Francis, W. N. and Kučera, H. (1982). *Frequency Analysis of English Usage*. Houghton Mifflin, Boston.

Franz, A. (1997). Independence assumptions considered harmful. In *ACL/EACL-97*, Madrid, Spain, pp. 182–189.

Franz, A. and Brants, T. (2006). All our n-gram are belong to you. http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html.

Fraser, N. M. and Gilbert, G. N. (1991). Simulating speech systems. *Computer Speech and Language*, *5*, 81–99.

Fyshe, A., Wehbe, L., Talukdar, P. P., Murphy, B., and Mitchell, T. M. (2015). A compositional and interpretable semantic space. In *NAACL HLT 2015*.

Gabow, H. N., Galil, Z., Spencer, T., and Tarjan, R. E. (1986). Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, *6*(2), 109–122.

Gale, W. A. and Church, K. W. (1994). What is wrong with adding one?. In Oostdijk, N. and de Haan, P. (Eds.), *Corpus-Based Research into Language*, pp. 189–198. Rodopi.

Gale, W. A., Church, K. W., and Yarowsky, D. (1992a). Estimating upper and lower bounds on the performance of word-sense disambiguation programs. In *ACL-92*, Newark, DE, pp. 249–256.

Gale, W. A., Church, K. W., and Yarowsky, D. (1992b). One sense per discourse. In *Proceedings DARPA Speech and Natural Language Workshop*, pp. 233–237.

Gale, W. A., Church, K. W., and Yarowsky, D. (1992c). Work on statistical methods for word sense disambiguation. In Goldman, R. (Ed.), *Proceedings of the 1992 AAAI Fall Symposium on Probabilistic Approaches to Natural Language*.

Garg, N., Schiebinger, L., Jurafsky, D., and Zou, J. (2018). Word embeddings quantify 100 years of gender and ethnic stereotypes. *Proceedings of the National Academy of Sciences*, *115*(16), E3635–E3644.

Garside, R. (1987). The CLAWS word-tagging system. In Garside, R., Leech, G., and Sampson, G. (Eds.), *The Computational Analysis of English*, pp. 30–41. Longman.

Garside, R., Leech, G., and McEnery, A. (1997). *Corpus Annotation*. Longman.

Gazdar, G., Klein, E., Pullum, G. K., and Sag, I. A. (1985). *Generalized Phrase Structure Grammar*. Blackwell.

Gerber, M. and Chai, J. Y. (2010). Beyond nombank: A study of implicit arguments for nominal predicates. In *ACL 2010*, pp. 1583–1592.

Gil, D. (2000). Syntactic categories, cross-linguistic variation and universal grammar. In Vogel, P. M. and Comrie, B. (Eds.), *Approaches to the Typology of Word Classes*, pp. 173–216. Mouton.

Gildea, D. and Jurafsky, D. (2000). Automatic labeling of semantic roles. In *ACL-00*, Hong Kong, pp. 512–520.

Gildea, D. and Jurafsky, D. (2002). Automatic labeling of semantic roles. *Computational Linguistics*, *28*(3), 245–288.

Gildea, D. and Palmer, M. (2002). The necessity of syntactic parsing for predicate argument recognition. In *ACL-02*, Philadelphia, PA.

Gillick, L. and Cox, S. J. (1989). Some statistical issues in the comparison of speech recognition algorithms. In *ICASSP-89*, pp. 532–535.

Ginzburg, J. and Sag, I. A. (2000). *Interrogative Investigations: the Form, Meaning and Use of English Interrogatives*. CSLI.

Giuliano, V. E. (1965). The interpretation of word associations. In Stevens, M. E., Giuliano, V. E., and Heilprin, L. B. (Eds.), *Statistical Association Methods For Mechanized Documentation. Symposium Proceedings. Washington, D.C., USA, March 17, 1964*, pp. 25–32. https://nvlpubs.nist.gov/nistpubs/Legacy/MP/nbsmiscellaneouspub269.pdf.

Givón, T. (1990). *Syntax: A Functional Typological Introduction*. John Benjamins.

Glennie, A. (1960). On the syntax machine and the construction of a universal compiler. Tech. rep. No. 2, Contr. NR 049-141, Carnegie Mellon University (at the time Carnegie Institute of Technology), Pittsburgh, PA.

Godfrey, J., Holliman, E., and McDaniel, J. (1992). SWITCHBOARD: Telephone speech corpus for research and development. In *ICASSP-92*, San Francisco, pp. 517–520.

Goffman, E. (1974). *Frame analysis: An essay on the organization of experience*. Harvard University Press.

Goldberg, J., Ostendorf, M., and Kirchhoff, K. (2003). The impact of response wording in error correction subdialogs. In *ISCA Tutorial and Research Workshop on Error Handling in Spoken Dialogue Systems*.

Goldberg, Y. (2017). *Neural Network Methods for Natural Language Processing*, Vol. 10 of *Synthesis Lectures on Human Language Technologies*. Morgan & Claypool.

Gondek, D., Lally, A., Kalyanpur, A., Murdock, J. W., Duboué, P. A., Zhang, L., Pan, Y., Qiu, Z., and Welty, C. (2012). A framework for merging and ranking of answers in deepqa. *IBM Journal of Research and Development*, *56*(3/4), 14:1–14:12.

Good, M. D., Whiteside, J. A., Wixon, D. R., and Jones, S. J. (1984). Building a user-derived interface. *Communications of the ACM*, *27*(10), 1032–1043.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

Goodman, J. (1997). Probabilistic feature grammars. In *IWPT-97*.

Goodman, J. (2006). A bit of progress in language modeling: Extended version. Tech. rep. MSR-TR-2001-72, Machine Learning and Applied Statistics Group, Microsoft Research, Redmond, WA.

Goodwin, C. (1996). Transparent vision. In Ochs, E., Schegloff, E. A., and Thompson, S. A. (Eds.), *Interaction and Grammar*, pp. 370–404. Cambridge University Press.

Gould, J. D., Conti, J., and Hovanyecz, T. (1983). Composing letters with a simulated listening typewriter. *Communications of the ACM*, *26*(4), 295–308.

Gould, J. D. and Lewis, C. (1985). Designing for usability: Key principles and what designers think. *Communications of the ACM*, *28*(3), 300–311.

Gould, S. J. (1980). *The Panda's Thumb*. Penguin Group.

Gravano, A., Hirschberg, J., and Beňuš, Š. (2012). Affirmative cue words in task-oriented dialogue. *Computational Linguistics*, *38*(1), 1–39.

Green, B. F., Wolf, A. K., Chomsky, C., and Laughery, K. (1961). Baseball: An automatic question answerer. In *Proceedings of the Western Joint Computer Conference 19*, pp. 219–224. Reprinted in Grosz et al. (1986).

Greene, B. B. and Rubin, G. M. (1971). Automatic grammatical tagging of English. Department of Linguistics, Brown University, Providence, Rhode Island.

Greenwald, A. G., McGhee, D. E., and Schwartz, J. L. K. (1998). Measuring individual differences in implicit cognition: the implicit association test.. *Journal of personality and social psychology*, *74*(6), 1464–1480.

Grenager, T. and Manning, C. D. (2006). Unsupervised Discovery of a Statistical Verb Lexicon. In *EMNLP 2006*.

Grishman, R. and Sundheim, B. (1995). Design of the MUC-6 evaluation. In *MUC-6*, San Francisco, pp. 1–11.

Grosz, B. J. (1977). *The Representation and Use of Focus in Dialogue Understanding*. Ph.D. thesis, University of California, Berkeley.

Grosz, B. J. and Sidner, C. L. (1980). Plans for discourse. In Cohen, P. R., Morgan, J., and Pollack, M. E. (Eds.), *Intentions in Communication*, pp. 417–444. MIT Press.

Gruber, J. S. (1965). *Studies in Lexical Relations*. Ph.D. thesis, MIT.

Guindon, R. (1988). A multidisciplinary perspective on dialogue structure in user-advisor dialogues. In Guindon, R. (Ed.), *Cognitive Science and Its Applications for Human-Computer Interaction*, pp. 163–200. Lawrence Erlbaum.

Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press.

Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *The Journal of Machine Learning Research*, *3*, 1157–1182.

Haghighi, A. and Klein, D. (2006). Prototype-driven grammar induction. In *COLING/ACL 2006*, pp. 881–888.

Hajič, J. (1998). *Building a Syntactically Annotated Corpus: The Prague Dependency Treebank*, pp. 106–132. Karolinum.

Hajič, J. (2000). Morphological tagging: Data vs. dictionaries. In *NAACL 2000*. Seattle.

Hajič, J., Ciaramita, M., Johansson, R., Kawahara, D., Martí, M. A., Màrquez, L., Meyers, A., Nivre, J., Padó, S., Štěpánek, J., et al. (2009). The conll-2009 shared task: Syntactic and semantic dependencies in multiple languages. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning: Shared Task*, pp. 1–18. Association for Computational Linguistics.

Hakkani-Tür, D., Oflazer, K., and Tür, G. (2002). Statistical morphological disambiguation for agglutinative languages. *Journal of Computers and Humanities*, *36*(4), 381–410.

Hakkani-Tür, D., Tür, G., Celikyilmaz, A., Chen, Y.-N., Gao, J., Deng, L., and Wang, Y.-Y. (2016). Multi-domain joint semantic frame parsing using bi-directional rnn-lstm.. In *INTERSPEECH*, pp. 715–719.

Hale, J. (2001). A probabilistic earley parser as a psycholinguistic model. In *NAACL 2001*, pp. 159–166.

Hamilton, W. L., Leskovec, J., and Jurafsky, D. (2016). Diachronic word embeddings reveal statistical laws of semantic change. In *ACL 2016*.

Harabagiu, S., Pasca, M., and Maiorano, S. (2000). Experiments with open-domain textual question answering. In *COLING-00*, Saarbrücken, Germany.

Harris, R. A. (2005). *Voice Interaction Design: Crafting the New Conversational Speech Systems*. Morgan Kaufmann.

Harris, Z. S. (1946). From morpheme to utterance. *Language*, *22*(3), 161–183.

Harris, Z. S. (1954). Distributional structure. *Word*, *10*, 146–162. Reprinted in J. Fodor and J. Katz, *The Structure of Language*, Prentice Hall, 1964 and in Z. S. Harris, *Papers in Structural and Transformational Linguistics*, Reidel, 1970, 775–794.

Harris, Z. S. (1962). *String Analysis of Sentence Structure*. Mouton, The Hague.

Hastie, T., Tibshirani, R. J., and Friedman, J. H. (2001). *The Elements of Statistical Learning*. Springer.

Hatzivassiloglou, V. and McKeown, K. R. (1997). Predicting the semantic orientation of adjectives. In *ACL/EACL-97*, pp. 174–181.

Hatzivassiloglou, V. and Wiebe, J. (2000). Effects of adjective orientation and gradability on sentence subjectivity. In *COLING-00*, pp. 299–305.

He, L., Lee, K., Lewis, M., and Zettlemoyer, L. (2017). Deep semantic role labeling: What works and what's next. In *ACL 2017*, pp. 473–483.

Heafield, K. (2011). KenLM: Faster and smaller language model queries. In *Workshop on Statistical Machine Translation*, pp. 187–197.

Heafield, K., Pouzyrevsky, I., Clark, J. H., and Koehn, P. (2013). Scalable modified Kneser-Ney language model estimation.. In *ACL 2013*, pp. 690–696.

Heaps, H. S. (1978). *Information retrieval. Computational and theoretical aspects*. Academic Press.

Hearst, M. A. (1991). Noun homograph disambiguation. In *Proceedings of the 7th Conference of the University of Waterloo Centre for the New OED and Text Research*, pp. 1–19.

Hearst, M. A. (1992a). Automatic acquisition of hyponyms from large text corpora. In *COLING-92*, Nantes, France.

Hearst, M. A. (1992b). Automatic acquisition of hyponyms from large text corpora. In *COLING-92*, Nantes, France. COLING.

Hearst, M. A. (1998). Automatic discovery of WordNet relations. In Fellbaum, C. (Ed.), *WordNet: An Electronic Lexical Database*. MIT Press.

Heckerman, D., Horvitz, E., Sahami, M., and Dumais, S. T. (1998). A bayesian approach to filtering junk e-mail. In *Proceeding of AAAI-98 Workshop on Learning for Text Categorization*, pp. 55–62.

Hemphill, C. T., Godfrey, J., and Doddington, G. (1990). The ATIS spoken language systems pilot corpus. In *Proceedings DARPA Speech and Natural Language Workshop*, Hidden Valley, PA, pp. 96–101.

Henderson, P., Sinha, K., Angelard-Gontier, N., Ke, N. R., Fried, G., Lowe, R., and Pineau, J. (2017). Ethical challenges in data-driven dialogue systems. In *AAAI/ACM AI Ethics and Society Conference*.

Hendrickx, I., Kim, S. N., Kozareva, Z., Nakov, P., Ó Séaghdha, D., Padó, S., Pennacchiotti, M., Romano, L., and Szpakowicz, S. (2009). Semeval-2010 task 8: Multi-way classification of semantic relations between pairs of nominals. In *Proceedings of the Workshop on Semantic Evaluations: Recent Achievements and Future Directions*, pp. 94–99.

Hendrix, G. G., Thompson, C. W., and Slocum, J. (1973). Language processing via canonical verbs and semantic models. In *Proceedings of IJCAI-73*.

Herdan, G. (1960). *Type-token mathematics*. The Hague, Mouton.

Hill, F., Reichart, R., and Korhonen, A. (2015). Simlex-999: Evaluating semantic models with (genuine) similarity estimation. *Computational Linguistics*, *41*(4), 665–695.

Hindle, D. and Rooth, M. (1990). Structural ambiguity and lexical relations. In *Proceedings DARPA Speech and Natural Language Workshop*, Hidden Valley, PA, pp. 257–262.

Hindle, D. and Rooth, M. (1991). Structural ambiguity and lexical relations. In *ACL-91*, Berkeley, CA, pp. 229–236.

Hinkelman, E. A. and Allen, J. (1989). Two constraints on speech act ambiguity. In *ACL-89*, Vancouver, Canada, pp. 212–219.

Hinton, G. E. (1986). Learning distributed representations of concepts. In *COGSCI-86*, pp. 1–12.

Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, *18*(7), 1527–1554.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. arXiv preprint arXiv:1207.0580.

Hirschberg, J., Litman, D. J., and Swerts, M. (2001). Identifying user corrections automatically in spoken dialogue systems. In *NAACL 2001*.

Hirschman, L. and Pao, C. (1993). The cost of errors in a spoken language system. In *EUROSPEECH-93*, pp. 1419–1422.

Hirst, G. (1987). *Semantic Interpretation and the Resolution of Ambiguity*. Cambridge University Press.

Hirst, G. (1988). Resolving lexical ambiguity computationally with spreading activation and polaroid words. In Small, S. L., Cottrell, G. W., and Tanenhaus, M. K. (Eds.), *Lexical Ambiguity Resolution*, pp. 73–108. Morgan Kaufmann.

Hirst, G. and Budanitsky, A. (2005). Correcting real-word spelling errors by restoring lexical cohesion. *Natural Language Engineering*, *11*, 87–111.

Hirst, G. and Charniak, E. (1982). Word sense and case slot disambiguation. In *AAAI-82*, pp. 95–98.

Hjelmslev, L. (1969). *Prologomena to a Theory of Language*. University of Wisconsin Press. Translated by Francis J. Whitfield; original Danish edition 1943.

Hobbs, J. R., Appelt, D. E., Bear, J., Israel, D., Kameyama, M., Stickel, M. E., and Tyson, M. (1997). FASTUS: A cascaded finite-state transducer for extracting information from natural-language text. In Roche, E. and Schabes, Y. (Eds.), *Finite-State Language Processing*, pp. 383–406. MIT Press.

Hockenmaier, J. and Steedman, M. (2007). Ccgbank: a corpus of ccg derivations and dependency structures extracted from the penn treebank. *Computational Linguistics*, *33*(3), 355–396.

Hofmann, T. (1999). Probabilistic latent semantic indexing. In *SIGIR-99*, Berkeley, CA.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Horning, J. J. (1969). *A Study of Grammatical Inference*. Ph.D. thesis, Stanford University.

Householder, F. W. (1995). Dionysius Thrax, the *technai*, and Sextus Empiricus. In Koerner, E. F. K. and Asher, R. E. (Eds.), *Concise History of the Language Sciences*, pp. 99–103. Elsevier Science.

Hovy, E. H., Hermjakob, U., and Ravichandran, D. (2002). A question/answer typology with surface text patterns. In *HLT-01*.

Hovy, E. H., Marcus, M. P., Palmer, M., Ramshaw, L. A., and Weischedel, R. (2006). Ontonotes: The 90% solution. In *HLT-NAACL-06*.

Hsu, B.-J. (2007). Generalized linear interpolation of language models. In *IEEE ASRU-07*, pp. 136–140.

Hu, M. and Liu, B. (2004a). Mining and summarizing customer reviews. In *KDD*, pp. 168–177.

Hu, M. and Liu, B. (2004b). Mining and summarizing customer reviews. In *SIGKDD-04*.

Huang, E. H., Socher, R., Manning, C. D., and Ng, A. Y. (2012). Improving word representations via global context and multiple word prototypes. In *ACL 2012*, pp. 873–882.

Huang, L. and Chiang, D. (2005). Better k-best parsing. In *IWPT-05*, pp. 53–64.

Huang, L. and Sagae, K. (2010). Dynamic programming for linear-time incremental parsing. In *Proceedings of ACL*, pp. 1077–1086. Association for Computational Linguistics.

Huang, Z., Xu, W., and Yu, K. (2015). Bidirectional LSTM-CRF models for sequence tagging. In *arXiv preprint arXiv:1508.01991*.

Huddleston, R. and Pullum, G. K. (2002). *The Cambridge Grammar of the English Language*. Cambridge University Press.

Hudson, R. A. (1984). *Word Grammar*. Blackwell.

Huffman, S. (1996). Learning information extraction patterns from examples. In Wertmer, S., Riloff, E., and Scheller, G. (Eds.), *Connectionist, Statistical, and Symbolic Approaches to Learning Natural Language Processing*, pp. 246–260. Springer.

Hutto, C. J., Folds, D., and Appling, S. (2015). Computationally detecting and quantifying the degree of bias in sentence-level text of news stories. In *HUSO 2015: The First International Conference on Human and Social Analytics*.

Hymes, D. (1974). Ways of speaking. In Bauman, R. and Sherzer, J. (Eds.), *Explorations in the ethnography of speaking*, pp. 433–451. Cambridge University Press.

Ide, N. M. and Véronis, J. (Eds.). (1998). *Computational Linguistics: Special Issue on Word Sense Disambiguation*, Vol. 24. MIT Press.

Irons, E. T. (1961). A syntax directed compiler for ALGOL 60. *Communications of the ACM*, *4*, 51–55.

Isbell, C. L., Kearns, M., Kormann, D., Singh, S., and Stone, P. (2000). Cobot in LambdaMOO: A social statistics agent. In *AAAI/IAAI*, pp. 36–41.

ISO8601 (2004). Data elements and interchange formats—information interchange—representation of dates and times. Tech. rep., International Organization for Standards (ISO).

Jacobs, P. S. and Rau, L. F. (1990). SCISOR: A system for extracting information from on-line news. *Communications of the ACM*, *33*(11), 88–97.

Jaech, A., Mulcaire, G., Hathi, S., Ostendorf, M., and Smith, N. A. (2016). Hierarchical character-word models for language identification. In *ACL Workshop on NLP for Social Media*, pp. 84—93.

Jafarpour, S., Burges, C., and Ritter, A. (2009). Filter, rank, and transfer the knowledge: Learning to chat. In *NIPS Workshop on Advances in Ranking*, Vancouver, Canada.

Jauhiainen, T., Lui, M., Zampieri, M., Baldwin, T., and Lindén, K. (2018). Automatic language identification in texts: A survey. arXiv preprint arXiv:1804.08186.

Jefferson, G. (1972). Side sequences. In Sudnow, D. (Ed.), *Studies in social interaction*, pp. 294–333. Free Press, New York.

Jefferson, G. (1984). Notes on a systematic deployment of the acknowledgement tokens 'yeah' and 'mm hm'. *Papers in Linguistics*, *17*(2), 197–216.

Jeffreys, H. (1948). *Theory of Probability* (2nd Ed.). Clarendon Press. Section 3.23.

Jekat, S., Klein, A., Maier, E., Maleck, I., Mast, M., and Quantz, J. (1995). Dialogue acts in verbmobil. Verbmobil–Report–65–95.

Jelinek, F. (1976). Continuous speech recognition by statistical methods. *Proceedings of the IEEE*, *64*(4), 532–557.

Jelinek, F. (1990). Self-organized language modeling for speech recognition. In Waibel, A. and Lee, K.-F. (Eds.), *Readings in Speech Recognition*, pp. 450–506. Morgan Kaufmann. Originally distributed as IBM technical report in 1985.

Jelinek, F. and Lafferty, J. D. (1991). Computation of the probability of initial substring generation by stochastic context-free grammars. *Computational Linguistics*, *17*(3), 315–323.

Jelinek, F., Lafferty, J. D., Magerman, D. M., Mercer, R. L., Ratnaparkhi, A., and Roukos, S. (1994). Decision tree parsing using a hidden derivation model. In *ARPA Human Language Technologies Workshop*, Plainsboro, N.J., pp. 272–277.

Jelinek, F. and Mercer, R. L. (1980). Interpolated estimation of Markov source parameters from sparse data. In Gelsema, E. S. and Kanal, L. N. (Eds.), *Proceedings, Workshop on Pattern Recognition in Practice*, pp. 381–397. North Holland.

Ji, H., Grishman, R., and Dang, H. T. (2010). Overview of the tac 2011 knowledge base population track. In *TAC-11*.

Jiang, J. J. and Conrath, D. W. (1997). Semantic similarity based on corpus statistics and lexical taxonomy. In *ROCLING X*, Taiwan.

Jiménez, V. M. and Marzal, A. (2000). Computation of the *n* best parse trees for weighted and stochastic context-free grammars. In *Advances in Pattern Recognition: Proceedings of the Joint IAPR International Workshops, SSPR 2000 and SPR 2000*, Alicante, Spain, pp. 183–192. Springer.

Johnson, M. (1998). PCFG models of linguistic tree representations. *Computational Linguistics*, *24*(4), 613–632.

Johnson, M. (2001). Joint and conditional estimation of tagging and parsing models. In *ACL-01*, pp. 314–321.

Johnson, M., Geman, S., Canon, S., Chi, Z., and Riezler, S. (1999). Estimators for stochastic "unification-based" grammars. In *ACL-99*, pp. 535–541.

Johnson, W. E. (1932). Probability: deductive and inductive problems (appendix to). *Mind*, *41*(164), 421–423.

Johnson-Laird, P. N. (1983). *Mental Models*. Harvard University Press, Cambridge, MA.

Jones, M. P. and Martin, J. H. (1997). Contextual spelling correction using latent semantic analysis. In *ANLP 1997*, Washington, D.C., pp. 166–173.

Jones, R., McCallum, A., Nigam, K., and Riloff, E. (1999). Bootstrapping for text learning tasks. In *IJCAI-99 Workshop on Text Mining: Foundations, Techniques and Applications*.

Jones, T. (2015). Toward a description of African American Vernacular English dialect regions using "Black Twitter". *American Speech*, *90*(4), 403–440.

Joos, M. (1950). Description of language design. *JASA*, *22*, 701–708.

Joshi, A. K. (1985). Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions?. In Dowty, D. R., Karttunen, L., and Zwicky, A. (Eds.), *Natural Language Parsing*, pp. 206–250. Cambridge University Press.

Joshi, A. K. and Hopely, P. (1999). A parser from antiquity. In Kornai, A. (Ed.), *Extended Finite State Models of Language*, pp. 6–15. Cambridge University Press.

Joshi, A. K. and Srinivas, B. (1994). Disambiguation of super parts of speech (or supertags): Almost parsing. In *COLING-94*, Kyoto, pp. 154–160.

Jurafsky, D., Chahuneau, V., Routledge, B. R., and Smith, N. A. (2014). Narrative framing of consumer sentiment in online restaurant reviews. *First Monday*, *19*(4).

Jurafsky, D., Wooters, C., Tajchman, G., Segal, J., Stolcke, A., Fosler, E., and Morgan, N. (1994). The Berkeley restaurant project. In *ICSLP-94*, Yokohama, Japan, pp. 2139–2142.

Jurgens, D. and Klapaftis, I. P. (2013). Semeval-2013 task 13: Word sense induction for graded and non-graded senses. In **SEM*, pp. 290–299.

Jurgens, D., Tsvetkov, Y., and Jurafsky, D. (2017). Incorporating dialectal variability for socially equitable language identification. In *ACL 2017*, pp. 51–57.

Justeson, J. S. and Katz, S. M. (1991). Co-occurrences of antonymous adjectives and their contexts. *Computational linguistics*, *17*(1), 1–19.

Kalyanpur, A., Boguraev, B. K., Patwardhan, S., Murdock, J. W., Lally, A., Welty, C., Prager, J. M., Coppola, B., Fokoue-Nkoutche, A., Zhang, L., Pan, Y., and Qiu, Z. M. (2012). Structured data and inference in deepqa. *IBM Journal of Research and Development*, *56*(3/4), 10:1–10:14.

Kang, J. S., Feng, S., Akoglu, L., and Choi, Y. (2014). Connotationwordnet: Learning connotation over the word+sense network. In *ACL 2014*.

Kannan, A. and Vinyals, O. (2016). Adversarial evaluation of dialogue models. In *NIPS 2016 Workshop on Adversarial Training*.

Kaplan, R. M. (1973). A general syntactic processor. In Rustin, R. (Ed.), *Natural Language Processing*, pp. 193–241. Algorithmics Press.

Kaplan, R. M., Riezler, S., King, T. H., Maxwell III, J. T., Vasserman, A., and Crouch, R. (2004). Speed and accuracy in shallow and deep stochastic parsing. In *HLT-NAACL-04*.

Karlsson, F., Voutilainen, A., Heikkilä, J., and Anttila, A. (Eds.). (1995). *Constraint Grammar: A Language-Independent System for Parsing Unrestricted Text*. Mouton de Gruyter.

Karttunen, L. (1999). Comments on Joshi. In Kornai, A. (Ed.), *Extended Finite State Models of Language*, pp. 16–18. Cambridge University Press.

Kasami, T. (1965). An efficient recognition and syntax analysis algorithm for context-free languages. Tech. rep. AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA.

Katz, J. J. and Fodor, J. A. (1963). The structure of a semantic theory. *Language*, *39*, 170–210.

Kawamoto, A. H. (1988). Distributed representations of ambiguous words and their resolution in connectionist networks. In Small, S. L., Cottrell, G. W., and Tanenhaus, M. (Eds.), *Lexical Ambiguity Resolution*, pp. 195–228. Morgan Kaufman.

Kay, M. (1967). Experiments with a powerful parser. In *Proc. 2eme Conference Internationale sur le Traitement Automatique des Langues*, Grenoble.

Kay, M. (1973). The MIND system. In Rustin, R. (Ed.), *Natural Language Processing*, pp. 155–188. Algorithmics Press.

Kay, M. (1982). Algorithm schemata and data structures in syntactic processing. In Allén, S. (Ed.), *Text Processing: Text Analysis and Generation, Text Typology and Attribution*, pp. 327–358. Almqvist and Wiksell, Stockholm.

Kay, P. and Fillmore, C. J. (1999). Grammatical constructions and linguistic generalizations: The What's X Doing Y? construction. *Language*, *75*(1), 1–33.

Keller, F. and Lapata, M. (2003). Using the web to obtain frequencies for unseen bigrams. *Computational Linguistics*, *29*, 459–484.

Kelly, E. F. and Stone, P. J. (1975). *Computer Recognition of English Word Senses*. North-Holland.

Kiela, D. and Clark, S. (2014). A systematic study of semantic vector space model parameters. In *Proceedings of the EACL 2nd Workshop on Continuous Vector Space Models and their Compositionality (CVSC)*, pp. 21–30.

Kilgarriff, A. (2001). English lexical sample task description. In *Proceedings of Senseval-2: Second International Workshop on Evaluating Word Sense Disambiguation Systems*, Toulouse, France, pp. 17–20.

Kilgarriff, A. and Palmer, M. (Eds.). (2000). *Computing and the Humanities: Special Issue on SENSEVAL*, Vol. 34. Kluwer.

Kilgarriff, A. and Rosenzweig, J. (2000). Framework and results for English SENSEVAL. *Computers and the Humanities*, *34*, 15–48.

Kim, S. M. and Hovy, E. H. (2004). Determining the sentiment of opinions. In *COLING-04*.

Kingma, D. and Ba, J. (2015). Adam: A method for stochastic optimization. In *ICLR 2015*.

Kipper, K., Dang, H. T., and Palmer, M. (2000). Class-based construction of a verb lexicon. In *AAAI-00*, Austin, TX, pp. 691–696.

Kleene, S. C. (1951). Representation of events in nerve nets and finite automata. Tech. rep. RM-704, RAND Corporation. RAND Research Memorandum.

Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In Shannon, C. and McCarthy, J. (Eds.), *Automata Studies*, pp. 3–41. Princeton University Press.

Klein, D. (2005). *The Unsupervised Learning of Natural Language Structure*. Ph.D. thesis, Stanford University.

Klein, D. and Manning, C. D. (2001). Parsing and hypergraphs. In *IWPT-01*, pp. 123–134.

Klein, D. and Manning, C. D. (2002). A generative constituent-context model for improved grammar induction. In *ACL-02*.

Klein, D. and Manning, C. D. (2003a). A* parsing: Fast exact Viterbi parse selection. In *HLT-NAACL-03*.

Klein, D. and Manning, C. D. (2003b). Accurate unlexicalized parsing. In *HLT-NAACL-03*.

Klein, D. and Manning, C. D. (2004). Corpus-based induction of syntactic structure: Models of dependency and constituency. In *ACL-04*, pp. 479–486.

Klein, S. and Simmons, R. F. (1963). A computational approach to grammatical coding of English words. *Journal of the Association for Computing Machinery*, *10*(3), 334–347.

Kneser, R. and Ney, H. (1995). Improved backing-off for M-gram language modeling. In *ICASSP-95*, Vol. 1, pp. 181–184.

Krovetz, R. (1993). Viewing morphology as an inference process. In *SIGIR-93*, pp. 191–202.

Krovetz, R. (1998). More than one sense per discourse. In *Proceedings of the ACL-SIGLEX SENSEVAL Workshop*.

Kruskal, J. B. (1983). An overview of sequence comparison. In Sankoff, D. and Kruskal, J. B. (Eds.), *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, pp. 1–44. Addison-Wesley.

Kudo, T. and Matsumoto, Y. (2002). Japanese dependency analysis using cascaded chunking. In *CoNLL-02*, pp. 63–69.

Kullback, S. and Leibler, R. A. (1951). On information and sufficiency. *Annals of Mathematical Statistics*, *22*, 79–86.

Kuno, S. (1965). The predictive analyzer and a path elimination technique. *Communications of the ACM*, *8*(7), 453–462.

Kuno, S. and Oettinger, A. G. (1963). Multiple-path syntactic analyzer. In Popplewell, C. M. (Ed.), *Information Processing 1962: Proceedings of the IFIP Congress 1962*, Munich, pp. 306–312. North-Holland. Reprinted in Grosz et al. (1986).

Kupiec, J. (1992). Robust part-of-speech tagging using a hidden Markov model. *Computer Speech and Language*, *6*, 225–242.

Kučera, H. and Francis, W. N. (1967). *Computational Analysis of Present-Day American English*. Brown University Press, Providence, RI.

Labov, W. and Fanshel, D. (1977). *Therapeutic Discourse*. Academic Press.

Lafferty, J. D., McCallum, A., and Pereira, F. C. N. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML 2001*, Stanford, CA.

Lafferty, J. D., Sleator, D., and Temperley, D. (1992). Grammatical trigrams: A probabilistic model of link grammar. In *Proceedings of the 1992 AAAI Fall Symposium on Probabilistic Approaches to Natural Language*.

Lakoff, G. (1965). *On the Nature of Syntactic Irregularity*. Ph.D. thesis, Indiana University. Published as *Irregularity in Syntax*. Holt, Rinehart, and Winston, New York, 1970.

Lally, A., Prager, J. M., McCord, M. C., Boguraev, B. K., Patwardhan, S., Fan, J., Fodor, P., and Chu-Carroll, J. (2012). Question analysis: How Watson reads a clue. *IBM Journal of Research and Development*, *56*(3/4), 2:1–2:14.

Lample, G., Ballesteros, M., Subramanian, S., Kawakami, K., and Dyer, C. (2016). Neural architectures for named entity recognition. In *NAACL HLT 2016*.

Landauer, T. K. (Ed.). (1995). *The Trouble with Computers: Usefulness, Usability, and Productivity*. MIT Press.

Landauer, T. K. and Dumais, S. T. (1997). A solution to Plato's problem: The Latent Semantic Analysis theory of acquisition, induction, and representation of knowledge. *Psychological Review*, *104*, 211–240.

Landes, S., Leacock, C., and Tengi, R. I. (1998). Building semantic concordances. In Fellbaum, C. (Ed.), *WordNet: An Electronic Lexical Database*, pp. 199–216. MIT Press.

Lang, J. and Lapata, M. (2014). Similarity-driven semantic role induction via graph partitioning. *Computational Linguistics*, *40*(3), 633–669.

Lapesa, G. and Evert, S. (2014). A large scale evaluation of distributional semantic models: Parameters, interactions and model selection. *TACL*, *2*, 531–545.

Lari, K. and Young, S. J. (1990). The estimation of stochastic context-free grammars using the Inside-Outside algorithm. *Computer Speech and Language*, *4*, 35–56.

Lau, J. H., Cook, P., McCarthy, D., Newman, D., and Baldwin, T. (2012). Word sense induction for novel sense detection. In *EACL-12*, pp. 591–601.

Leacock, C. and Chodorow, M. S. (1998). Combining local context and WordNet similarity for word sense identification. In Fellbaum, C. (Ed.), *WordNet: An Electronic Lexical Database*, pp. 265–283. MIT Press.

Leacock, C., Towell, G., and Voorhees, E. M. (1993). Corpus-based statistical sense resolution. In *HLT-93*, pp. 260–265.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, *1*(4), 541–551.

LeCun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. E., and Jackel, L. D. (1990). Handwritten digit recognition with a back-propagation network. In *NIPS 1990*, pp. 396–404.

Lee, D. D. and Seung, H. S. (1999). Learning the parts of objects by non-negative matrix factorization. *Nature*, *401*(6755), 788–791.

Lehnert, W. G., Cardie, C., Fisher, D., Riloff, E., and Williams, R. (1991). Description of the CIRCUS system as used for MUC-3. In Sundheim, B. (Ed.), *MUC-3*, pp. 223–233.

Lemon, O., Georgila, K., Henderson, J., and Stuttle, M. (2006). An ISU dialogue system exhibiting reinforcement learning of dialogue policies: Generic slot-filling in the TALK in-car system. In *EACL-06*.

Lesk, M. E. (1986). Automatic sense disambiguation using machine readable dictionaries: How to tell a pine cone from an ice cream cone. In *Proceedings of the 5th International Conference on Systems Documentation*, Toronto, CA, pp. 24–26.

Leuski, A. and Traum, D. (2011). NPCEditor: Creating virtual human dialogue using information retrieval techniques. *AI Magazine*, *32*(2), 42–56.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, *10*(8), 707–710. Original in *Doklady Akademii Nauk SSSR* 163(4): 845–848 (1965).

Levesque, H. J., Cohen, P. R., and Nunes, J. H. T. (1990). On acting together. In *AAAI-90*, Boston, MA, pp. 94–99. Morgan Kaufmann.

Levin, B. (1977). Mapping sentences to case frames. Tech. rep. 167, MIT AI Laboratory. AI Working Paper 143.

Levin, B. (1993). *English Verb Classes and Alternations: A Preliminary Investigation*. University of Chicago Press.

Levin, B. and Rappaport Hovav, M. (2005). *Argument Realization*. Cambridge University Press.

Levin, E., Pieraccini, R., and Eckert, W. (2000). A stochastic model of human-machine interaction for learning dialog strategies. *IEEE Transactions on Speech and Audio Processing*, *8*, 11–23.

Levinson, S. C. (1983). *Conversational Analysis*, chap. 6. Cambridge University Press.

Levow, G.-A. (1998). Characterizing and recognizing spoken corrections in human-computer dialogue. In *COLING-ACL*, pp. 736–742.

Levy, O. and Goldberg, Y. (2014a). Dependency-based word embeddings. In *ACL 2014*.

Levy, O. and Goldberg, Y. (2014b). Linguistic regularities in sparse and explicit word representations. In *CoNLL-14*.

Levy, O. and Goldberg, Y. (2014c). Neural word embedding as implicit matrix factorization. In *NIPS 14*, pp. 2177–2185.

Levy, O., Goldberg, Y., and Dagan, I. (2015). Improving distributional similarity with lessons learned from word embeddings. *TACL*, *3*, 211–225.

Levy, R. (2008). Expectation-based syntactic comprehension. *Cognition*, *106*(3), 1126–1177.

Lewis, M. and Steedman, M. (2014). A* ccg parsing with a supertag-factored model.. In *EMNLP*, pp. 990–1000.

Li, J., Chen, X., Hovy, E. H., and Jurafsky, D. (2015). Visualizing and understanding neural models in NLP. In *NAACL HLT 2015*.

Li, J., Galley, M., Brockett, C., Gao, J., and Dolan, B. (2016a). A diversity-promoting objective function for neural conversation models. In *NAACL HLT 2016*.

Li, J., Monroe, W., Ritter, A., Galley, M., Gao, J., and Jurafsky, D. (2016b). Deep reinforcement learning for dialogue generation. In *EMNLP 2016*.

Li, J., Monroe, W., Shi, T., Ritter, A., and Jurafsky, D. (2017). Adversarial learning for neural dialogue generation. In *EMNLP 2017*.

Li, X. and Roth, D. (2002). Learning question classifiers. In *COLING-02*, pp. 556–562.

Li, X. and Roth, D. (2005). Learning question classifiers: The role of semantic information. *Journal of Natural Language Engineering*, *11*(4).

Lin, D. (1995). A dependency-based method for evaluating broad-coverage parsers. In *IJCAI-95*, Montreal, pp. 1420–1425.

Lin, D. (1998). An information-theoretic definition of similarity. In *ICML 1998*, San Francisco, pp. 296–304.

Lin, D. (2003). Dependency-based evaluation of minipar. In *Workshop on the Evaluation of Parsing Systems*.

Lin, J. (2007). An exploration of the principles underlying redundancy-based factoid question answering. *ACM Transactions on Information Systems*, *25*(2).

Lin, Y., Michel, J.-B., Lieberman Aiden, E., Orwant, J., Brockman, W., and Petrov, S. (2012). Syntactic annotations for the google books ngram corpus. In *ACL 2012*, pp. 169–174.

Litman, D. J., Swerts, M., and Hirschberg, J. (2000). Predicting automatic speech recognition performance using prosodic cues. In *NAACL 2000*.

Litman, D. J., Walker, M. A., and Kearns, M. (1999). Automatic detection of poor speech recognition at the dialogue level. In *ACL-99*, College Park, MA, pp. 309–316.

Liu, B. (2015). *Sentiment Analysis: Mining Opinions, Sentiments, and Emotions*. Cambridge University Press.

Liu, B. and Zhang, L. (2012). A survey of opinion mining and sentiment analysis. In Aggarwal, C. C. and Zhai, C. (Eds.), *Mining text data*, pp. 415–464. Springer.

Liu, C.-W., Lowe, R. T., Serban, I. V., Noseworthy, M., Charlin, L., and Pineau, J. (2016). How NOT to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation. In *EMNLP 2016*.

Liu, X., Gales, M. J. F., and Woodland, P. C. (2013). Use of contexts in language model interpolation and adaptation. *Computer Speech & Language*, *27*(1), 301–321.

Lochbaum, K. E., Grosz, B. J., and Sidner, C. L. (2000). Discourse structure and intention recognition. In Dale, R., Moisl, H., and Somers, H. L. (Eds.), *Handbook of Natural Language Processing*. Marcel Dekker.

Lovins, J. B. (1968). Development of a stemming algorithm. *Mechanical Translation and Computational Linguistics*, *11*(1–2), 9–13.

Lowe, R. T., Noseworthy, M., Serban, I. V., Angelard-Gontier, N., Bengio, Y., and Pineau, J. (2017a). Towards an automatic Turing test: Learning to evaluate dialogue responses. In *ACL 2017*.

Lowe, R. T., Pow, N., Serban, I. V., Charlin, L., Liu, C.-W., and Pineau, J. (2017b). Training end-to-end dialogue systems with the ubuntu dialogue corpus. *Dialogue & Discourse*, *8*(1), 31–65.

Luhn, H. P. (1957). A statistical approach to the mechanized encoding and searching of literary information. *IBM Journal of Research and Development*, *1*(4), 309–317.

Lui, M. and Baldwin, T. (2011). Cross-domain feature selection for language identification. In *IJCNLP-11*, pp. 553–561.

Lui, M. and Baldwin, T. (2012). `langid.py`: An off-the-shelf language identification tool. In *ACL 2012*, pp. 25–30.

Ma, X. and Hovy, E. (2016). End-to-end sequence labeling via bi-directional LSTM-CNNs-CRF. In *ACL 2016*.

Madhu, S. and Lytel, D. (1965). A figure of merit technique for the resolution of non-grammatical ambiguity. *Mechanical Translation*, *8*(2), 9–13.

Magerman, D. M. (1994). *Natural Language Parsing as Statistical Pattern Recognition*. Ph.D. thesis, University of Pennsylvania.

Magerman, D. M. (1995). Statistical decision-tree models for parsing. In *ACL-95*, pp. 276–283.

Magerman, D. M. and Marcus, M. P. (1991). Pearl: A probabilistic chart parser. In *EACL-91*, Berlin.

Mairesse, F. and Walker, M. A. (2008). Trainable generation of big-five personality styles through data-driven parameter estimation. In *ACL-08*, Columbus.

Manandhar, S., Klapaftis, I. P., Dligach, D., and Pradhan, S. (2010). Semeval-2010 task 14: Word sense induction & disambiguation. In *SemEval-2010*, pp. 63–68.

Manning, C. D. (2011). Part-of-speech tagging from 97% to 100%: Is it time for some linguistics?. In *CICLing 2011*, pp. 171–189.

Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge.

Manning, C. D. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT Press.

Marcus, M. P. (1980). *A Theory of Syntactic Recognition for Natural Language*. MIT Press.

Marcus, M. P. (1990). Summary of session 9: Automatic acquisition of linguistic structure. In *Proceedings DARPA Speech and Natural Language Workshop*, Hidden Valley, PA, pp. 249–250.

Marcus, M. P., Kim, G., Marcinkiewicz, M. A., MacIntyre, R., Bies, A., Ferguson, M., Katz, K., and Schasberger, B. (1994). The Penn Treebank: Annotating predicate argument structure. In *ARPA Human Language Technology Workshop*, Plainsboro, NJ, pp. 114–119. Morgan Kaufmann.

Marcus, M. P., Santorini, B., and Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English: The Penn treebank. *Computational Linguistics*, *19*(2), 313–330.

Markov, A. A. (1913). Essai d'une recherche statistique sur le texte du roman "Eugene Onegin" illustrant la liaison des epreuve en chain ('Example of a statistical investigation of the text of "Eugene Onegin" illustrating the dependence between samples in chain'). *Izvistia Imperatorskoi Akademii Nauk (Bulletin de l'Académie Impériale des Sciences de St.-Pétersbourg)*, *7*, 153–162.

Maron, M. E. (1961). Automatic indexing: an experimental inquiry. *Journal of the ACM (JACM)*, *8*(3), 404–417.

Màrquez, L., Carreras, X., Litkowski, K. C., and Stevenson, S. (2008). Semantic role labeling: An introduction to the special issue. *Computational linguistics*, *34*(2), 145–159.

Marshall, I. (1983). Choice of grammatical word-class without GLobal syntactic analysis: Tagging words in the LOB corpus. *Computers and the Humanities*, *17*, 139–150.

Marshall, I. (1987). Tag selection using probabilistic methods. In Garside, R., Leech, G., and Sampson, G. (Eds.), *The Computational Analysis of English*, pp. 42–56. Longman.

Martin, J. H. (1986). The acquisition of polysemy. In *ICML 1986*, Irvine, CA, pp. 198–204.

Masterman, M. (1957). The thesaurus in syntax and semantics. *Mechanical Translation*, *4*(1), 1–2.

McCallum, A., Freitag, D., and Pereira, F. C. N. (2000). Maximum entropy Markov models for information extraction and segmentation. In *ICML 2000*, pp. 591–598.

McCallum, A. and Nigam, K. (1998). A comparison of event models for naive bayes text classification. In *AAAI/ICML-98 Workshop on Learning for Text Categorization*, pp. 41–48.

McCawley, J. D. (1998). *The Syntactic Phenomena of English*. University of Chicago Press.

McClelland, J. L. and Elman, J. L. (1986). The TRACE model of speech perception. *Cognitive Psychology*, *18*, 1–86.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, *5*, 115–133. Reprinted in *Neurocomputing: Foundations of Research, ed. by J. A. Anderson and E Rosenfeld. MIT Press 1988.*

McDonald, R., Crammer, K., and Pereira, F. C. N. (2005). Online large-margin training of dependency parsers. In *ACL-05*, Ann Arbor, pp. 91–98.

McDonald, R. and Nivre, J. (2011). Analyzing and integrating dependency parsers. *Computational Linguistics*, *37*(1), 197–230.

McDonald, R., Pereira, F. C. N., Ribarov, K., and Hajič, J. (2005). Non-projective dependency parsing using spanning tree algorithms. In *HLT-EMNLP-05*.

Mehl, M. R., Gosling, S. D., and Pennebaker, J. W. (2006). Personality in its natural habitat: manifestations and implicit folk theories of personality in daily life.. *Journal of Personality and Social Psychology*, *90*(5).

Mel'čuk, I. A. (1988). *Dependency Syntax: Theory and Practice*. State University of New York Press.

Merialdo, B. (1994). Tagging English text with a probabilistic model. *Computational Linguistics*, *20*(2), 155–172.

Mesnil, G., Dauphin, Y., Yao, K., Bengio, Y., Deng, L., Hakkani-Tür, D., He, X., Heck, L., Tür, G., Yu, D., and Zweig, G. (2015). Using recurrent neural networks for slot filling in spoken language understanding. *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, *23*(3), 530–539.

Metsis, V., Androutsopoulos, I., and Paliouras, G. (2006). Spam filtering with naive bayes-which naive bayes?. In *CEAS*, pp. 27–28.

Meyers, A., Reeves, R., Macleod, C., Szekely, R., Zielinska, V., Young, B., and Grishman, R. (2004). The nombank project: An interim report. In *Proceedings of the NAACL/HLT Workshop: Frontiers in Corpus Annotation*.

Microsoft (2014). `http://www.msxiaoice.com`..

Mihalcea, R. (2007). Using wikipedia for automatic word sense disambiguation. In *NAACL-HLT 07*, pp. 196–203.

Mihalcea, R. and Moldovan, D. (2001). Automatic generation of a coarse grained WordNet. In *NAACL Workshop on WordNet and Other Lexical Resources*.

Mikheev, A., Moens, M., and Grover, C. (1999). Named entity recognition without gazetteers. In *EACL-99*, Bergen, Norway, pp. 1–8.

Mikolov, T. (2012). *Statistical language models based on neural networks*. Ph.D. thesis, Ph. D. thesis, Brno University of Technology.

Mikolov, T., Chen, K., Corrado, G. S., and Dean, J. (2013). Efficient estimation of word representations in vector space. In *ICLR 2013*.

Mikolov, T., Kombrink, S., Burget, L., Černockỳ, J. H., and Khudanpur, S. (2011). Extensions of recurrent neural network language model. In *ICASSP-11*, pp. 5528–5531.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013a). Distributed representations of words and phrases and their compositionality. In *NIPS 13*, pp. 3111–3119.

Mikolov, T., Yih, W.-t., and Zweig, G. (2013b). Linguistic regularities in continuous space word representations. In *NAACL HLT 2013*, pp. 746–751.

Miller, G. A. and Charles, W. G. (1991). Contextual correlates of semantics similarity. *Language and Cognitive Processes*, *6*(1), 1–28.

Miller, G. A. and Chomsky, N. (1963). Finitary models of language users. In Luce, R. D., Bush, R. R., and Galanter, E. (Eds.), *Handbook of Mathematical Psychology*, Vol. II, pp. 419–491. John Wiley.

Miller, G. A., Leacock, C., Tengi, R. I., and Bunker, R. T. (1993). A semantic concordance. In *Proceedings ARPA Workshop on Human Language Technology*, pp. 303–308.

Miller, G. A. and Selfridge, J. A. (1950). Verbal context and the recall of meaningful material. *American Journal of Psychology*, *63*, 176–185.

Miller, S., Bobrow, R. J., Ingria, R., and Schwartz, R. (1994). Hidden understanding models of natural language. In *ACL-94*, Las Cruces, NM, pp. 25–32.

Minsky, M. (1961). Steps toward artificial intelligence. *Proceedings of the IRE*, *49*(1), 8–30.

Minsky, M. (1974). A framework for representing knowledge. Tech. rep. 306, MIT AI Laboratory. Memo 306.

Minsky, M. and Papert, S. (1969). *Perceptrons*. MIT Press.

Mintz, M., Bills, S., Snow, R., and Jurafsky, D. (2009). Distant supervision for relation extraction without labeled data. In *ACL IJCNLP 2009*.

Miwa, M. and Bansal, M. (2016). End-to-end relation extraction using lstms on sequences and tree structures. In *ACL 2016*, pp. 1105–1116.

Mohammad, S. M. and Turney, P. D. (2013). Crowdsourcing a word-emotion association lexicon. *Computational Intelligence*, *29*(3), 436–465.

Monroe, B. L., Colaresi, M. P., and Quinn, K. M. (2008). Fightin'words: Lexical feature selection and evaluation for identifying the content of political conflict. *Political Analysis*, *16*(4), 372–403.

Monz, C. (2004). Minimal span weighting retrieval for question answering. In *SIGIR Workshop on Information Retrieval for Question Answering*, pp. 23–30.

Morgan, A. A., Hirschman, L., Colosimo, M., Yeh, A. S., and Colombe, J. B. (2004). Gene name identification and normalization using a model organism database. *Journal of Biomedical Informatics*, *37*(6), 396–410.

Morgan, N. and Bourlard, H. (1989). Generalization and parameter estimation in feedforward nets: Some experiments. In *Advances in neural information processing systems*, pp. 630–637.

Morgan, N. and Bourlard, H. (1990). Continuous speech recognition using multilayer perceptrons with hidden markov models. In *ICASSP-90*, pp. 413–416.

Morris, W. (Ed.). (1985). *American Heritage Dictionary* (2nd College Edition Ed.). Houghton Mifflin.

Mosteller, F. and Wallace, D. L. (1963). Inference in an authorship problem: A comparative study of discrimination methods applied to the authorship of the disputed federalist papers. *Journal of the American Statistical Association*, *58*(302), 275–309.

Mosteller, F. and Wallace, D. L. (1964). *Inference and Disputed Authorship: The Federalist*. Springer-Verlag. A second edition appeared in 1984 as *Applied Bayesian and Classical Inference*.

Mrkšić, N., O'Séaghdha, D., Wen, T.-H., Thomson, B., and Young, S. J. (2017). Neural belief tracker: Data-driven dialogue state tracking. In *ACL 2017*.

Murdock, J. W., Fan, J., Lally, A., Shima, H., and Boguraev, B. K. (2012a). Textual evidence gathering and analysis. *IBM Journal of Research and Development*, *56*(3/4), 8:1–8:14.

Murdock, J. W., Kalyanpur, A., Welty, C., Fan, J., Ferrucci, D. A., Gondek, D. C., Zhang, L., and Kanayama, H. (2012b). Typing candidate answers using type coercion. *IBM Journal of Research and Development*, *56*(3/4), 7:1–7:13.

Murphy, K. P. (2012). *Machine learning: A probabilistic perspective*. MIT press.

Nádas, A. (1984). Estimation of probabilities in the language model of the IBM speech recognition system. *IEEE Transactions on Acoustics, Speech, Signal Processing*, *32*(4), 859–861.

Nagata, M. and Morimoto, T. (1994). First steps toward statistical modeling of dialogue to predict the speech act type of the next utterance. *Speech Communication*, *15*, 193–203.

Nash-Webber, B. L. (1975). The role of semantics in automatic speech understanding. In Bobrow, D. G. and Collins, A. (Eds.), *Representation and Understanding*, pp. 351–382. Academic Press.

Naur, P., Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., van Wijnagaarden, A., and Woodger, M. (1960). Report on the algorithmic language ALGOL 60. *Communications of the ACM*, *3*(5), 299–314. Revised in CACM 6:1, 1-17, 1963.

Navigli, R. (2006). Meaningful clustering of senses helps boost word sense disambiguation performance. In *COLING/ACL 2006*, pp. 105–112.

Navigli, R. (2009). Word sense disambiguation: A survey. *ACM Computing Surveys*, *41*(2).

Navigli, R. and Lapata, M. (2010). An experimental study of graph connectivity for unsupervised word sense disambiguation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *32*(4), 678–692.

Navigli, R. and Vannella, D. (2013). Semeval-2013 task 11: Word sense induction & disambiguation within an end-user application. In **SEM*, pp. 193–201.

Neff, G. and Nagy, P. (2016). Talking to bots: Symbiotic agency and the case of Tay. *International Journal of Communication*, *10*, 4915–4931.

Newell, A., Langer, S., and Hickey, M. (1998). The rôle of natural language processing in alternative and augmentative communication. *Natural Language Engineering*, *4*(1), 1–16.

Ney, H. (1991). Dynamic programming parsing for context-free grammars in continuous speech recognition. *IEEE Transactions on Signal Processing*, *39*(2), 336–340.

Ng, A. Y. and Jordan, M. I. (2002). On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In *NIPS 14*, pp. 841–848.

Nielsen, J. (1992). The usability engineering life cycle. *IEEE Computer*, *25*(3), 12–22.

Nielsen, M. A. (2015). *Neural networks and Deep learning*. Determination Press USA.

Nigam, K., Lafferty, J. D., and McCallum, A. (1999). Using maximum entropy for text classification. In *IJCAI-99 workshop on machine learning for information filtering*, pp. 61–67.

Nilsson, J., Riedel, S., and Yuret, D. (2007). The conll 2007 shared task on dependency parsing. In *Proceedings of the CoNLL shared task session of EMNLP-CoNLL*, pp. 915–932. sn.

NIST (2005). Speech recognition scoring toolkit (sctk) version 2.1. http://www.nist.gov/speech/tools/.

Nivre, J. (2007). Incremental non-projective dependency parsing. In *NAACL-HLT 07*.

Nivre, J. (2003). An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT*.

Nivre, J. (2006). *Inductive Dependency Parsing*. Springer.

Nivre, J. (2009). Non-projective dependency parsing in expected linear time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1 - Volume 1*, Stroudsburg, PA, USA, pp. 351–359. Association for Computational Linguistics.

Nivre, J., de Marneffe, M.-C., Ginter, F., Goldberg, Y., Hajič, J., Manning, C. D., McDonald, R. T., Petrov, S., Pyysalo, S., Silveira, N., Tsarfaty, R., and Zeman, D. (2016a). Universal Dependencies v1: A multilingual treebank collection. In *LREC*.

Nivre, J., de Marneffe, M.-C., Ginter, F., Goldberg, Y., Hajič, J., Manning, C. D., McDonald, R. T., Petrov, S., Pyysalo, S., Silveira, N., Tsarfaty, R., and Zeman, D. (2016b). Universal Dependencies v1: A multilingual treebank collection. In *LREC*.

Nivre, J., Hall, J., Nilsson, J., Chanev, A., Eryigit, G., Kübler, S., Marinov, S., and Marsi, E. (2007). Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, *13*(02), 95–135.

Nivre, J. and Nilsson, J. (2005). Pseudo-projective dependency parsing. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pp. 99–106. Association for Computational Linguistics.

Nivre, J. and Scholz, M. (2004). Deterministic dependency parsing of english text. In *Proceedings of the 20th international conference on Computational Linguistics*, p. 64. Association for Computational Linguistics.

Niwa, Y. and Nitta, Y. (1994). Co-occurrence vectors from corpora vs. distance vectors from dictionaries. In *ACL-94*, pp. 304–309.

Noreen, E. W. (1989). *Computer Intensive Methods for Testing Hypothesis*. Wiley.

Norman, D. A. (1988). *The Design of Everyday Things*. Basic Books.

Norvig, P. (1991). Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, *17*(1), 91–98.

Nosek, B. A., Banaji, M. R., and Greenwald, A. G. (2002a). Harvesting implicit group attitudes and beliefs from a demonstration web site. *Group Dynamics: Theory, Research, and Practice*, *6*(1), 101.

Nosek, B. A., Banaji, M. R., and Greenwald, A. G. (2002b). Math=male, me=female, therefore math≠ me. *Journal of personality and social psychology*, *83*(1), 44.

O'Connor, B., Krieger, M., and Ahn, D. (2010). Tweetmotif: Exploratory search and topic summarization for twitter. In *ICWSM*.

Oh, A. H. and Rudnicky, A. I. (2000). Stochastic language generation for spoken dialogue systems. In *Proceedings of the 2000 ANLP/NAACL Workshop on Conversational systems-Volume 3*, pp. 27–32.

Oravecz, C. and Dienes, P. (2002). Efficient stochastic part-of-speech tagging for Hungarian. In *LREC-02*, Las Palmas, Canary Islands, Spain, pp. 710–717.

Osgood, C. E., Suci, G. J., and Tannenbaum, P. H. (1957). *The Measurement of Meaning*. University of Illinois Press.

Packard, D. W. (1973). Computer-assisted morphological analysis of ancient Greek. In Zampolli, A. and Calzolari, N. (Eds.), *Computational and Mathematical Linguistics: Proceedings of the International Conference on Computational Linguistics*, Pisa, pp. 343–355. Leo S. Olschki.

Palmer, D. (2012). Text preprocessing. In Indurkhya, N. and Damerau, F. J. (Eds.), *Handbook of Natural Language Processing*, pp. 9–30. CRC Press.

Palmer, M., Babko-Malaya, O., and Dang, H. T. (2004). Different sense granularities for different applications. In *HLT-NAACL Workshop on Scalable Natural Language Understanding*, Boston, MA, pp. 49–56.

Palmer, M., Dang, H. T., and Fellbaum, C. (2006). Making fine-grained and coarse-grained sense distinctions, both manually and automatically. *Natural Language Engineering*, *13*(2), 137–163.

Palmer, M., Fellbaum, C., Cotton, S., Delfs, L., and Dang, H. T. (2001). English tasks: All-words and verb lexical sample. In *Proceedings of Senseval-2: 2nd International Workshop on Evaluating Word Sense Disambiguation Systems*, Toulouse, France, pp. 21–24.

Palmer, M., Gildea, D., and Xue, N. (2010). Semantic role labeling. *Synthesis Lectures on Human Language Technologies*, *3*(1), 1–103.

Palmer, M., Kingsbury, P., and Gildea, D. (2005). The proposition bank: An annotated corpus of semantic roles. *Computational Linguistics*, *31*(1), 71–106.

Palmer, M., Ng, H. T., and Dang, H. T. (2006). Evaluation of wsd systems. In Agirre, E. and Edmonds, P. (Eds.), *Word Sense Disambiguation: Algorithms and Applications*. Kluwer.

Pang, B. and Lee, L. (2008). Opinion mining and sentiment analysis. *Foundations and trends in information retrieval*, *2*(1-2), 1–135.

Pang, B., Lee, L., and Vaithyanathan, S. (2002). Thumbs up? Sentiment classification using machine learning techniques. In *EMNLP 2002*, pp. 79–86.

Paolino, J. (2017). Google Home vs Alexa: Two simple user experience design gestures that delighted a female user. In *Medium*. Jan 4, 2017. https://medium.com/startup-grind/google-home-vs-alexa-56e26f69ac77.

Pasca, M. (2003). *Open-Domain Question Answering from Large Text Collections*. CSLI.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch. In *NIPS-W*.

Pearl, C. (2017). *Designing Voice User Interfaces: Principles of Conversational Experiences*. O'Reilly.

Pedersen, T. and Bruce, R. (1997). Distinguishing word senses in untagged text. In *EMNLP 1997*, Providence, RI.

Peng, N., Poon, H., Quirk, C., Toutanova, K., and Yih, W.-t. (2017). Cross-sentence n-ary relation extraction with graph LSTMs. *TACL*, *5*, 101–115.

Penn, G. and Kiparsky, P. (2012). On Pāṇini and the generative capacity of contextualized replacement systems. In *COLING-12*, pp. 943–950.

Pennebaker, J. W., Booth, R. J., and Francis, M. E. (2007). *Linguistic Inquiry and Word Count: LIWC 2007*. Austin, TX.

Pennebaker, J. W. and King, L. A. (1999). Linguistic styles: language use as an individual difference. *Journal of Personality and Social Psychology*, *77*(6).

Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *EMNLP 2014*, pp. 1532–1543.

Percival, W. K. (1976). On the historical source of immediate constituent analysis. In McCawley, J. D. (Ed.), *Syntax and Semantics Volume 7, Notes from the Linguistic Underground*, pp. 229–242. Academic Press.

Perrault, C. R. and Allen, J. (1980). A plan-based analysis of indirect speech acts. *American Journal of Computational Linguistics*, *6*(3-4), 167–182.

Petrov, S., Barrett, L., Thibaux, R., and Klein, D. (2006). Learning accurate, compact, and interpretable tree annotation. In *COLING/ACL 2006*, Sydney, Australia, pp. 433–440.

Petrov, S., Das, D., and McDonald, R. (2012). A universal part-of-speech tagset. In *LREC 2012*.

Petrov, S. and McDonald, R. (2012). Overview of the 2012 shared task on parsing the web. In *Notes of the First Workshop on Syntactic Analysis of Non-Canonical Language (SANCL)*, Vol. 59.

Phillips, A. V. (1960). A question-answering routine. Tech. rep. 16, MIT AI Lab.

Picard, R. W. (1995). Affective computing. Tech. rep. 321, MIT Media Lab Perceputal Computing Technical Report. Revised November 26, 1995.

Pieraccini, R., Levin, E., and Lee, C.-H. (1991). Stochastic representation of conceptual structure in the ATIS task. In *Proceedings DARPA Speech and Natural Language Workshop*, Pacific Grove, CA, pp. 121–124.

Plutchik, R. (1962). *The emotions: Facts, theories, and a new model*. Random House.

Plutchik, R. (1980). A general psychoevolutionary theory of emotion. In Plutchik, R. and Kellerman, H. (Eds.), *Emotion: Theory, Research, and Experience, Volume 1*, pp. 3–33. Academic Press.

Polifroni, J., Hirschman, L., Seneff, S., and Zue, V. W. (1992). Experiments in evaluating interactive spoken language systems. In *Proceedings DARPA Speech and Natural Language Workshop*, Harriman, NY, pp. 28–33.

Pollard, C. and Sag, I. A. (1994). *Head-Driven Phrase Structure Grammar*. University of Chicago Press.

Ponzetto, S. P. and Navigli, R. (2010). Knowledge-rich word sense disambiguation rivaling supervised systems. In *ACL 2010*, pp. 1522–1531.

Porter, M. F. (1980). An algorithm for suffix stripping. *Program*, *14*(3), 130–127.

Potts, C. (2011). On the negativity of negation. In Li, N. and Lutz, D. (Eds.), *Proceedings of Semantics and Linguistic Theory 20*, pp. 636–659. CLC Publications, Ithaca, NY.

Pradhan, S., Moschitti, A., Xue, N., Ng, H. T., Björkelund, A., Uryupina, O., Zhang, Y., and Zhong, Z. (2013). Towards robust linguistic analysis using OntoNotes. In *CoNLL-13*, pp. 143–152.

Pradhan, S., Ward, W., Hacioglu, K., Martin, J. H., and Jurafsky, D. (2005). Semantic role labeling using different syntactic views. In *ACL-05*, Ann Arbor, MI.

Purver, M. (2004). *The theory and use of clarification requests in dialogue*. Ph.D. thesis, University of London.

Pustejovsky, J. (1995). *The Generative Lexicon*. MIT Press.

Pustejovsky, J. and Boguraev, B. (Eds.). (1996). *Lexical Semantics: The Problem of Polysemy*. Oxford University Press.

Pustejovsky, J., Castaño, J., Ingria, R., Saurí, R., Gaizauskas, R., Setzer, A., and Katz, G. (2003a). TimeML: robust specification of event and temporal expressions in text. In *Proceedings of the 5th International Workshop on Computational Semantics (IWCS-5)*.

Pustejovsky, J., Hanks, P., Saurí, R., See, A., Gaizauskas, R., Setzer, A., Radev, D., Sundheim, B., Day, D. S., Ferro, L., and Lazo, M. (2003b). The TIMEBANK corpus. In *Proceedings of Corpus Linguistics 2003 Conference*, pp. 647–656. UCREL Technical Paper number 16.

Pustejovsky, J., Ingria, R., Saurí, R., Castaño, J., Littman, J., Gaizauskas, R., Setzer, A., Katz, G., and Mani, I. (2005). *The Specification Language TimeML*, chap. 27. Oxford.

Qiu, G., Liu, B., Bu, J., and Chen, C. (2009). Expanding domain sentiment lexicon through double propagation.. In *IJCAI-09*, pp. 1199–1204.

Quillian, M. R. (1968). Semantic memory. In Minsky, M. (Ed.), *Semantic Information Processing*, pp. 227–270. MIT Press.

Quillian, M. R. (1969). The teachable language comprehender: A simulation program and theory of language. *Communications of the ACM*, *12*(8), 459–476.

Quirk, R., Greenbaum, S., Leech, G., and Svartvik, J. (1985). *A Comprehensive Grammar of the English Language*. Longman.

Radford, A. (1988). *Transformational Grammar: A First Course*. Cambridge University Press.

Radford, A. (1997). *Syntactic Theory and the Structure of English: A Minimalist Approach*. Cambridge University Press.

Ranganath, R., Jurafsky, D., and McFarland, D. A. (2013). Detecting friendly, flirtatious, awkward, and assertive speech in speed-dates. *Computer Speech and Language*, *27*(1), 89–115.

Ratnaparkhi, A. (1996). A maximum entropy part-of-speech tagger. In *EMNLP 1996*, Philadelphia, PA, pp. 133–142.

Ratnaparkhi, A. (1997). A linear observed time statistical parser based on maximum entropy models. In *EMNLP 1997*, Providence, RI, pp. 1–10.

Ratnaparkhi, A., Reynar, J. C., and Roukos, S. (1994). A maximum entropy model for prepositional phrase attachment. In *ARPA Human Language Technologies Workshop*, Plainsboro, N.J., pp. 250–255.

Raymond, C. and Riccardi, G. (2007). Generative and discriminative algorithms for spoken language understanding. In *INTERSPEECH-07*, pp. 1605–1608.

Rehder, B., Schreiner, M. E., Wolfe, M. B. W., Laham, D., Landauer, T. K., and Kintsch, W. (1998). Using Latent Semantic Analysis to assess knowledge: Some technical considerations. *Discourse Processes*, *25*(2-3), 337–354.

Resnik, P. (1992). Probabilistic tree-adjoining grammar as a framework for statistical natural language processing. In *COLING-92*, Nantes, France, pp. 418–424.

Resnik, P. (1993). Semantic classes and syntactic ambiguity. In *Proceedings of the workshop on Human Language Technology*, pp. 278–283.

Resnik, P. (1995). Using information content to evaluate semantic similarity in a taxanomy. In *International Joint Conference for Artificial Intelligence (IJCAI-95)*, pp. 448–453.

Resnik, P. (1996). Selectional constraints: An information-theoretic model and its computational realization. *Cognition*, *61*, 127–159.

Resnik, P. (2006). Word sense disambiguation in NLP appplications. In Agirre, E. and Edmonds, P. (Eds.), *Word Sense Disambiguation: Algorithms and Applications*. Kluwer.

Riedel, S., Yao, L., and McCallum, A. (2010). Modeling relations and their mentions without labeled text. In *Machine Learning and Knowledge Discovery in Databases*, pp. 148–163. Springer.

Riedel, S., Yao, L., McCallum, A., and Marlin, B. M. (2013). Relation extraction with matrix factorization and universal schemas. In *NAACL HLT 2013*.

Riesbeck, C. K. (1975). Conceptual analysis. In Schank, R. C. (Ed.), *Conceptual Information Processing*, pp. 83–156. American Elsevier, New York.

Riezler, S., King, T. H., Kaplan, R. M., Crouch, R., Maxwell III, J. T., and Johnson, M. (2002). Parsing the Wall Street Journal using a Lexical-Functional Grammar and discriminative estimation techniques. In *ACL-02*, Philadelphia, PA.

Riloff, E. (1993). Automatically constructing a dictionary for information extraction tasks. In *AAAI-93*, Washington, D.C., pp. 811–816.

Riloff, E. and Jones, R. (1999). Learning dictionaries for information extraction by multi-level bootstrapping. In *AAAI-99*, pp. 474–479.

Riloff, E. and Schmelzenbach, M. (1998). An empirical approach to conceptual case frame acquisition. In *Proceedings of the Sixth Workshop on Very Large Corpora*, Montreal, Canada, pp. 49–56.

Riloff, E. and Wiebe, J. (2003). Learning extraction patterns for subjective expressions. In *EMNLP 2003*, Sapporo, Japan.

Ritter, A., Cherry, C., and Dolan, B. (2011). Data-driven response generation in social media. In *EMNLP-11*, pp. 583–593.

Ritter, A., Etzioni, O., and Mausam (2010). A latent dirichlet allocation method for selectional preferences. In *ACL 2010*, pp. 424–434.

Ritter, A., Zettlemoyer, L., Mausam, and Etzioni, O. (2013). Modeling missing data in distant supervision for information extraction.. *TACL*, *1*, 367–378.

Roark, B. (2001). Probabilistic top-down parsing and language modeling. *Computational Linguistics*, *27*(2), 249–276.

Roark, B., Saraclar, M., and Collins, M. (2007). Discriminative n-gram language modeling. *Computer Speech & Language*, *21*(2), 373–392.

Rohde, D. L. T., Gonnerman, L. M., and Plaut, D. C. (2006). An improved model of semantic similarity based on lexical co-occurrence. *Communications of the ACM*, *8*, 627–633.

Rooth, M., Riezler, S., Prescher, D., Carroll, G., and Beil, F. (1999). Inducing a semantically annotated lexicon via EM-based clustering. In *ACL-99*, College Park, MA, pp. 104–111.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain.. *Psychological review*, *65*(6), 386–408.

Rosenfeld, R. (1996). A maximum entropy approach to adaptive statistical language modeling. *Computer Speech and Language*, *10*, 187–228.

Roy, N., Pineau, J., and Thrun, S. (2000). Spoken dialog management for robots. In *ACL-00*, Hong Kong.

Rubenstein, H. and Goodenough, J. B. (1965). Contextual correlates of synonymy. *Communications of the ACM*, *8*(10), 627–633.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In Rumelhart, D. E. and McClelland, J. L. (Eds.), *Parallel Distributed Processing*, Vol. 2, pp. 318–362. MIT Press.

Rumelhart, D. E. and McClelland, J. L. (1986a). On learning the past tense of English verbs. In Rumelhart, D. E. and McClelland, J. L. (Eds.), *Parallel Distributed Processing*, Vol. 2, pp. 216–271. MIT Press.

Rumelhart, D. E. and McClelland, J. L. (Eds.). (1986b). *Parallel Distributed Processing*. MIT Press.

Ruppenhofer, J., Ellsworth, M., Petruck, M. R. L., Johnson, C. R., Baker, C. F., and Scheffczyk, J. (2016). FrameNet II: Extended theory and practice..

Ruppenhofer, J., Sporleder, C., Morante, R., Baker, C., and Palmer, M. (2010). Semeval-2010 task 10: Linking events and their participants in discourse. In *Proceedings of the 5th International Workshop on Semantic Evaluation*, pp. 45–50.

Russell, J. A. (1980). A circumplex model of affect. *Journal of personality and social psychology*, *39*(6), 1161–1178.

Russell, S. and Norvig, P. (2002). *Artificial Intelligence: A Modern Approach* (2nd Ed.). Prentice Hall.

Sacks, H., Schegloff, E. A., and Jefferson, G. (1974). A simplest systematics for the organization of turn-taking for conversation. *Language*, *50*(4), 696–735.

Sag, I. A. and Liberman, M. Y. (1975). The intonational disambiguation of indirect speech acts. In *CLS-75*, pp. 487–498. University of Chicago.

Sag, I. A., Wasow, T., and Bender, E. M. (Eds.). (2003). *Syntactic Theory: A Formal Introduction*. CSLI Publications, Stanford, CA.

Sahami, M., Dumais, S. T., Heckerman, D., and Horvitz, E. (1998). A Bayesian approach to filtering junk e-mail. In *AAAI Workshop on Learning for Text Categorization*, pp. 98–105.

Salomaa, A. (1969). Probabilistic and weighted grammars. *Information and Control*, *15*, 529–544.

Salton, G. (1971). *The SMART Retrieval System: Experiments in Automatic Document Processing*. Prentice Hall.

Sampson, G. (1987). Alternative grammatical coding systems. In Garside, R., Leech, G., and Sampson, G. (Eds.), *The Computational Analysis of English*, pp. 165–183. Longman.

Samuelsson, C. (1993). Morphological tagging based entirely on Bayesian inference. In *9th Nordic Conference on Computational Linguistics NODALIDA-93*. Stockholm.

Sankoff, D. and Labov, W. (1979). On the uses of variable rules. *Language in society*, *8*(2-3), 189–222.

Schabes, Y. (1990). *Mathematical and Computational Aspects of Lexicalized Grammars*. Ph.D. thesis, University of Pennsylvania, Philadelphia, PA.

Schabes, Y. (1992). Stochastic lexicalized tree-adjoining grammars. In *COLING-92*, Nantes, France, pp. 426–433.

Schabes, Y., Abeillé, A., and Joshi, A. K. (1988). Parsing strategies with 'lexicalized' grammars: Applications to Tree Adjoining Grammars. In *COLING-88*, Budapest, pp. 578–583.

Schank, R. C. and Abelson, R. P. (1975). Scripts, plans, and knowledge. In *Proceedings of IJCAI-75*, pp. 151–157.

Schank, R. C. and Abelson, R. P. (1977). *Scripts, Plans, Goals and Understanding*. Lawrence Erlbaum.

Schegloff, E. A. (1968). Sequencing in conversational openings. *American Anthropologist*, *70*, 1075–1095.

Schegloff, E. A. (1972). Notes on a conversational practice: Formulating place. In Sudnow, D. (Ed.), *Studies in social interaction*, New York. Free Press.

Schegloff, E. A. (1982). Discourse as an interactional achievement: Some uses of 'uh huh' and other things that come between sentences. In Tannen, D. (Ed.), *Analyzing Discourse: Text and Talk*, pp. 71–93. Georgetown University Press, Washington, D.C.

Scherer, K. R. (2000). Psychological models of emotion. In Borod, J. C. (Ed.), *The neuropsychology of emotion*, pp. 137–162. Oxford.

Schone, P. and Jurafsky, D. (2000). Knowlege-free induction of morphology using latent semantic analysis. In *CoNLL-00*.

Schone, P. and Jurafsky, D. (2001). Knowledge-free induction of inflectional morphologies. In *NAACL 2001*.

Schütze, H. (1992a). Context space. In Goldman, R. (Ed.), *Proceedings of the 1992 AAAI Fall Symposium on Probabilistic Approaches to Natural Language*.

Schütze, H. (1992b). Dimensions of meaning. In *Proceedings of Supercomputing '92*, pp. 787–796. IEEE Press.

Schütze, H. (1997a). *Ambiguity Resolution in Language Learning – Computational and Cognitive Models*. CSLI, Stanford, CA.

Schütze, H. (1997b). *Ambiguity Resolution in Language Learning: Computational and Cognitive Models*. CSLI Publications, Stanford, CA.

Schütze, H. (1998). Automatic word sense discrimination. *Computational Linguistics*, *24*(1), 97–124.

Schütze, H., Hull, D. A., and Pedersen, J. (1995). A comparison of classifiers and document representations for the routing problem. In *SIGIR-95*, pp. 229–237.

Schütze, H. and Pedersen, J. (1993). A vector model for syntagmatic and paradigmatic relatedness. In *Proceedings of the 9th Annual Conference of the UW Centre for the New OED and Text Research*, pp. 104–113.

Schütze, H. and Singer, Y. (1994). Part-of-speech tagging using a variable memory Markov model. In *ACL-94*, Las Cruces, NM, pp. 181–187.

Schwartz, H. A., Eichstaedt, J. C., Kern, M. L., Dziurzynski, L., Ramones, S. M., Agrawal, M., Shah, A., Kosinski, M., Stillwell, D., Seligman, M. E. P., and Ungar, L. H. (2013). Personality, gender, and age in the language of social media: The open-vocabulary approach. *PloS one*, *8*(9), e73791.

Schwartz, R. and Chow, Y.-L. (1990). The N-best algorithm: An efficient and exact procedure for finding the N most likely sentence hypotheses. In *ICASSP-90*, Vol. 1, pp. 81–84.

Schwenk, H. (2007). Continuous space language models. *Computer Speech & Language*, *21*(3), 492–518.

Scott, M. and Shillcock, R. (2003). Eye movements reveal the on-line computation of lexical probabilities during reading. *Psychological Science*, *14*(6), 648–652.

Séaghdha, D. O. (2010). Latent variable models of selectional preference. In *ACL 2010*, pp. 435–444.

Seddah, D., Tsarfaty, R., Kübler, S., Candito, M., Choi, J., Farkas, R., Foster, J., Goenaga, I., Gojenola, K., Goldberg, Y., et al. (2013). Overview of the spmrl 2013 shared task: cross-framework evaluation of parsing morphologically rich languages. In *Proceedings of the 4th Workshop on Statistical Parsing of Morphologically-Rich Languages*. Association for Computational Linguistics.

Sekine, S. and Collins, M. (1997). The evalb software. http://cs.nyu.edu/cs/projects/proteus/evalb.

Serban, I. V., Lowe, R. T., Charlin, L., and Pineau, J. (2017). A survey of available corpora for building data-driven dialogue systems.. arXiv preprint arXiv:1512.05742.

Sgall, P., Hajičová, E., and Panevova, J. (1986). *The Meaning of the Sentence in its Pragmatic Aspects*. Reidel.

Shang, L., Lu, Z., and Li, H. (2015). Neural responding machine for short-text conversation. In *ACL 2015*, pp. 1577–1586.

Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, *27*(3), 379–423. Continued in the following volume.

Shannon, C. E. (1951). Prediction and entropy of printed English. *Bell System Technical Journal*, *30*, 50–64.

Sheil, B. A. (1976). Observations on context free parsing. *SMIL: Statistical Methods in Linguistics*, *1*, 71–109.

Shriberg, E., Bates, R., Taylor, P., Stolcke, A., Jurafsky, D., Ries, K., Coccaro, N., Martin, R., Meteer, M., and Van Ess-Dykema, C. (1998). Can prosody aid the automatic classification of dialog acts in conversational speech?. *Language and Speech (Special Issue on Prosody and Conversation)*, *41*(3-4), 439–487.

Simmons, R. F. (1965). Answering English questions by computer: A survey. *Communications of the ACM*, *8*(1), 53–70.

Simmons, R. F. (1973). Semantic networks: Their computation and use for understanding English sentences. In Schank, R. C. and Colby, K. M. (Eds.), *Computer Models of Thought and Language*, pp. 61–113. W.H. Freeman and Co.

Simmons, R. F., Klein, S., and McConlogue, K. (1964). Indexing and dependency logic for answering english questions. *American Documentation*, *15*(3), 196–204.

Simons, G. F. and Fennig, C. D. (2018). Ethnologue: Languages of the world, twenty-first edition.. Dallas, Texas. SIL International.

Singh, S. P., Litman, D. J., Kearns, M., and Walker, M. A. (2002). Optimizing dialogue management with reinforcement learning: Experiments with the NJFun system. *Journal of Artificial Intelligence Research (JAIR)*, *16*, 105–133.

Sleator, D. and Temperley, D. (1993). Parsing English with a link grammar. In *IWPT-93*.

Small, S. L., Cottrell, G. W., and Tanenhaus, M. (Eds.). (1988). *Lexical Ambiguity Resolution*. Morgan Kaufman.

Small, S. L. and Rieger, C. (1982). Parsing and comprehending with Word Experts. In Lehnert, W. G. and Ringle, M. H. (Eds.), *Strategies for Natural Language Processing*, pp. 89–147. Lawrence Erlbaum.

Smith, D. A. and Eisner, J. (2007). Bootstrapping feature-rich dependency parsers with entropic priors. In *EMNLP/-CoNLL 2007*, Prague, pp. 667–677.

Smith, N. A. and Eisner, J. (2005). Guiding unsupervised grammar induction using contrastive estimation. In *IJCAI Workshop on Grammatical Inference Applications*, Edinburgh, pp. 73–82.

Smith, V. L. and Clark, H. H. (1993). On the course of answering questions. *Journal of Memory and Language*, *32*, 25–38.

Smolensky, P. (1988). On the proper treatment of connectionism. *Behavioral and brain sciences*, *11*(1), 1–23.

Smolensky, P. (1990). Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial intelligence*, *46*(1-2), 159–216.

Snow, R., Jurafsky, D., and Ng, A. Y. (2005). Learning syntactic patterns for automatic hypernym discovery. In Saul, L. K., Weiss, Y., and Bottou, L. (Eds.), *NIPS 17*, pp. 1297–1304. MIT Press.

Snow, R., Prakash, S., Jurafsky, D., and Ng, A. Y. (2007). Learning to merge word senses. In *EMNLP/CoNLL 2007*, pp. 1005–1014.

Socher, R., Huval, B., Manning, C. D., and Ng, A. Y. (2012). Semantic compositionality through recursive matrix-vector spaces. In *EMNLP 2012*, pp. 1201–1211.

Soderland, S., Fisher, D., Aseltine, J., and Lehnert, W. G. (1995). CRYSTAL: Inducing a conceptual dictionary. In *IJCAI-95*, Montreal, pp. 1134–1142.

Søgaard, A. (2010). Simple semi-supervised training of part-of-speech taggers. In *ACL 2010*, pp. 205–208.

Søgaard, A., Johannsen, A., Plank, B., Hovy, D., and Alonso, H. M. (2014). What's in a p-value in NLP?. In *CoNLL-14*.

Solorio, T., Blair, E., Maharjan, S., Bethard, S., Diab, M., Ghoneim, M., Hawwari, A., AlGhamdi, F., Hirschberg, J., Chang, A., and Fung, P. (2014). Overview for the first shared task on language identification in code-switched data. In *Proceedings of the First Workshop on Computational Approaches to Code Switching*, pp. 62–72.

Sordoni, A., Galley, M., Auli, M., Brockett, C., Ji, Y., Mitchell, M., Nie, J.-Y., Gao, J., and Dolan, B. (2015). A neural network approach to context-sensitive generation of conversational responses. In *NAACL HLT 2015*, pp. 196–205.

Sparck Jones, K. (1972). A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, *28*(1), 11–21.

Sparck Jones, K. (1986). *Synonymy and Semantic Classification*. Edinburgh University Press, Edinburgh. Republication of 1964 PhD Thesis.

Spitkovsky, V. I. and Chang, A. X. (2012). A cross-lingual dictionary for English Wikipedia concepts. In *LREC-12*, Istanbul, Turkey.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2014). Dropout: a simple way to prevent neural networks from overfitting.. *Journal of Machine Learning Research*, *15*(1), 1929–1958.

Stalnaker, R. C. (1978). Assertion. In Cole, P. (Ed.), *Pragmatics: Syntax and Semantics Volume 9*, pp. 315–332. Academic Press.

Stamatatos, E. (2009). A survey of modern authorship attribution methods. *JASIST*, *60*(3), 538–556.

Steedman, M. (1989). Constituency and coordination in a combinatory grammar. In Baltin, M. R. and Kroch, A. S. (Eds.), *Alternative Conceptions of Phrase Structure*, pp. 201–231. University of Chicago.

Steedman, M. (1996). *Surface Structure and Interpretation*. MIT Press. Linguistic Inquiry Monograph, 30.

Steedman, M. (2000). *The Syntactic Process*. The MIT Press.

Stetina, J. and Nagao, M. (1997). Corpus based PP attachment ambiguity resolution with a semantic dictionary. In Zhou, J. and Church, K. W. (Eds.), *Proceedings of the Fifth Workshop on Very Large Corpora*, Beijing, China, pp. 66–80.

Stifelman, L. J., Arons, B., Schmandt, C., and Hulteen, E. A. (1993). VoiceNotes: A speech interface for a hand-held voice notetaker. In *Human Factors in Computing Systems: INTERCHI '93 Conference Proceedings*, pp. 179–186.

Stolcke, A. (1995). An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics*, *21*(2), 165–202.

Stolcke, A. (1998). Entropy-based pruning of backoff language models. In *Proc. DARPA Broadcast News Transcription and Understanding Workshop*, Lansdowne, VA, pp. 270–274.

Stolcke, A. (2002). SRILM – an extensible language modeling toolkit. In *ICSLP-02*, Denver, CO.

Stolcke, A., Ries, K., Coccaro, N., Shriberg, E., Bates, R., Jurafsky, D., Taylor, P., Martin, R., Meteer, M., and Van Ess-Dykema, C. (2000). Dialogue act modeling for automatic tagging and recognition of conversational speech. *Computational Linguistics*, *26*(3), 339–371.

Stolz, W. S., Tannenbaum, P. H., and Carstensen, F. V. (1965). A stochastic approach to the grammatical coding of English. *Communications of the ACM*, *8*(6), 399–405.

Stone, P., Dunphry, D., Smith, M., and Ogilvie, D. (1966). *The General Inquirer: A Computer Approach to Content Analysis*. Cambridge, MA: MIT Press.

Stoyanchev, S. and Johnston, M. (2015). Localized error detection for targeted clarification in a virtual assistant. In *ICASSP-15*, pp. 5241–5245.

Stoyanchev, S., Liu, A., and Hirschberg, J. (2013). Modelling human clarification strategies. In *SIGDIAL 2013*, pp. 137–141.

Stoyanchev, S., Liu, A., and Hirschberg, J. (2014). Towards natural clarification questions in dialogue systems. In *AISB symposium on questions, discourse and dialogue*.

Strötgen, J. and Gertz, M. (2013). Multilingual and cross-domain temporal tagging. *Language Resources and Evaluation*, *47*(2), 269–298.

Suendermann, D., Evanini, K., Liscombe, J., Hunter, P., Dayanidhi, K., and Pieraccini, R. (2009). From rule-based to statistical grammars: Continuous improvement of large-scale spoken dialog systems. In *ICASSP-09*, pp. 4713–4716.

Sundheim, B. (Ed.). (1991). *Proceedings of MUC-3*.

Sundheim, B. (Ed.). (1992). *Proceedings of MUC-4*.

Sundheim, B. (Ed.). (1993). *Proceedings of MUC-5*, Baltimore, MD.

Sundheim, B. (Ed.). (1995). *Proceedings of MUC-6*.

Surdeanu, M. (2013). Overview of the TAC2013 Knowledge Base Population evaluation: English slot filling and temporal slot filling. In *TAC-13*.

Surdeanu, M., Ciaramita, M., and Zaragoza, H. (2011). Learning to rank answers to non-factoid questions from web collections. *Computational Linguistics*, *37*(2), 351–383.

Surdeanu, M., Harabagiu, S., Williams, J., and Aarseth, P. (2003). Using predicate-argument structures for information extraction. In *ACL-03*, pp. 8–15.

Surdeanu, M., Johansson, R., Meyers, A., Màrquez, L., and Nivre, J. (2008a). The conll-2008 shared task on joint parsing of syntactic and semantic dependencies. In *Proceedings of the Twelfth Conference on Computational Natural Language Learning*, pp. 159–177. Association for Computational Linguistics.

Surdeanu, M., Johansson, R., Meyers, A., Màrquez, L., and Nivre, J. (2008b). The conll-2008 shared task on joint parsing of syntactic and semantic dependencies. In *CoNLL-08*, pp. 159–177.

Swerts, M., Litman, D. J., and Hirschberg, J. (2000). Corrections in spoken dialogue systems. In *ICSLP-00*, Beijing, China.

Swier, R. and Stevenson, S. (2004). Unsupervised semantic role labelling. In *EMNLP 2004*, pp. 95–102.

Switzer, P. (1965). Vector images in document retrieval. In Stevens, M. E., Giuliano, V. E., and Heilprin, L. B. (Eds.), *Statistical Association Methods For Mechanized Documentation. Symposium Proceedings. Washington, D.C., USA, March 17, 1964*, pp. 163–171. https://nvlpubs.nist.gov/nistpubs/Legacy/MP/nbsmiscellaneouspub269.pdf.

Talbot, D. and Osborne, M. (2007). Smoothed Bloom Filter Language Models: Tera-Scale LMs on the Cheap. In *EMNLP/CoNLL 2007*, pp. 468–476.

Tannen, D. (1979). What's in a frame? Surface evidence for underlying expectations. In Freedle, R. (Ed.), *New Directions in Discourse Processing*, pp. 137–181. Ablex.

Taskar, B., Klein, D., Collins, M., Koller, D., and Manning, C. D. (2004). Max-margin parsing. In *EMNLP 2004*, pp. 1–8.

Tesnière, L. (1959). *Éléments de Syntaxe Structurale*. Librairie C. Klincksieck, Paris.

Thede, S. M. and Harper, M. P. (1999). A second-order hidden Markov model for part-of-speech tagging. In *ACL-99*, College Park, MA, pp. 175–182.

Thompson, K. (1968). Regular expression search algorithm. *Communications of the ACM*, *11*(6), 419–422.

Tibshirani, R. J. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, *58*(1), 267–288.

Titov, I. and Henderson, J. (2006). Loss minimization in parse reranking. In *EMNLP 2006*.

Titov, I. and Khoddam, E. (2014). Unsupervised induction of semantic roles within a reconstruction-error minimization framework. In *NAACL HLT 2015*.

Titov, I. and Klementiev, A. (2012). A Bayesian approach to unsupervised semantic role induction. In *EACL-12*, pp. 12–22.

Tomkins, S. S. (1962). *Affect, imagery, consciousness: Vol. I. The positive affects*. Springer.

Toutanova, K., Klein, D., Manning, C. D., and Singer, Y. (2003). Feature-rich part-of-speech tagging with a cyclic dependency network. In *HLT-NAACL-03*.

Toutanova, K., Manning, C. D., Flickinger, D., and Oepen, S. (2005). Stochastic HPSG Parse Disambiguation using the Redwoods Corpus. *Research on Language & Computation*, *3*(1), 83–105.

Tseng, H., Chang, P.-C., Andrew, G., Jurafsky, D., and Manning, C. D. (2005a). Conditional random field word segmenter. In *Proceedings of the Fourth SIGHAN Workshop on Chinese Language Processing*.

Tseng, H., Jurafsky, D., and Manning, C. D. (2005b). Morphological features help POS tagging of unknown words across language varieties. In *Proceedings of the 4th SIGHAN Workshop on Chinese Language Processing*.

Turian, J., Ratinov, L., and Bengio, Y. (2010). Word representations: a simple and general method for semi-supervised learning. In *ACL 2010*, pp. 384–394.

Turney, P. D. (2002). Thumbs up or thumbs down? semantic orientation applied to unsupervised classification of reviews. In *ACL-02*.

UzZaman, N., Llorens, H., Derczynski, L., Allen, J., Verhagen, M., and Pustejovsky, J. (2013). Semeval-2013 task 1: Tempeval-3: Evaluating time expressions, events, and temporal relations. In *SemEval-13*, pp. 1–9.

van der Maaten, L. and Hinton, G. E. (2008). Visualizing high-dimensional data using t-sne. *Journal of Machine Learning Research*, *9*, 2579–2605.

van Rijsbergen, C. J. (1975). *Information Retrieval*. Butterworths.

Van Valin, Jr., R. D. and La Polla, R. (1997). *Syntax: Structure, Meaning, and Function*. Cambridge University Press.

VanLehn, K., Jordan, P. W., Rosé, C., Bhembe, D., Böttner, M., Gaydos, A., Makatchev, M., Pappuswamy, U., Ringenberg, M., Roque, A., Siler, S., Srivastava, R., and Wilson, R. (2002). The architecture of Why2-Atlas: A coach for qualitative physics essay writing. In *Proc. Intelligent Tutoring Systems*.

Vasilescu, F., Langlais, P., and Lapalme, G. (2004). Evaluating variants of the lesk approach for disambiguating words. In *LREC-04*, Lisbon, Portugal, pp. 633–636. ELRA.

Velikovich, L., Blair-Goldensohn, S., Hannan, K., and McDonald, R. (2010). The viability of web-derived polarity lexicons. In *NAACL HLT 2010*, pp. 777–785.

Verhagen, M., Gaizauskas, R., Schilder, F., Hepple, M., Moszkowicz, J., and Pustejovsky, J. (2009). The tempeval challenge: identifying temporal relations in text. *Language Resources and Evaluation*, *43*(2), 161–179.

Verhagen, M., Mani, I., Sauri, R., Knippen, R., Jang, S. B., Littman, J., Rumshisky, A., Phillips, J., and Pustejovsky, J. (2005). Automating temporal annotation with tarsqi. In *ACL-05*, pp. 81–84.

Vinyals, O. and Le, Q. (2015). A neural conversational model. In *Proceedings of ICML Deep Learning Workshop*, Lille, France.

Voutilainen, A. (1995). Morphological disambiguation. In Karlsson, F., Voutilainen, A., Heikkilä, J., and Anttila, A. (Eds.), *Constraint Grammar: A Language-Independent System for Parsing Unrestricted Text*, pp. 165–284. Mouton de Gruyter.

Voutilainen, A. (1999). Handcrafted rules. In van Halteren, H. (Ed.), *Syntactic Wordclass Tagging*, pp. 217–246. Kluwer.

Wade, E., Shriberg, E., and Price, P. J. (1992). User behaviors affecting speech recognition. In *ICSLP-92*, pp. 995–998.

Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. *Journal of the Association for Computing Machinery*, *21*, 168–173.

Walker, M. A. (2000). An application of reinforcement learning to dialogue strategy selection in a spoken dialogue system for email. *Journal of Artificial Intelligence Research*, *12*, 387–416.

Walker, M. A., Fromer, J. C., and Narayanan, S. S. (1998). Learning optimal dialogue strategies: A case study of a spoken dialogue agent for email. In *COLING/ACL-98*, Montreal, Canada, pp. 1345–1351.

Walker, M. A., Kamm, C. A., and Litman, D. J. (2001). Towards developing general models of usability with PARADISE. *Natural Language Engineering: Special Issue on Best Practice in Spoken Dialogue Systems*, *6*(3), 363–377.

Walker, M. A. and Whittaker, S. (1990). Mixed initiative in dialogue: An investigation into discourse segmentation. In *ACL-90*, Pittsburgh, PA, pp. 70–78.

Wang, H., Lu, Z., Li, H., and Chen, E. (2013). A dataset for research on short-text conversations.. In *EMNLP 2013*, pp. 935–945.

Wang, S. and Manning, C. D. (2012). Baselines and bigrams: Simple, good sentiment and topic classification. In *ACL 2012*, pp. 90–94.

Ward, N. and Tsukahara, W. (2000). Prosodic features which cue back-channel feedback in English and Japanese. *Journal of Pragmatics*, *32*, 1177–1207.

Ward, W. and Issar, S. (1994). Recent improvements in the CMU spoken language understanding system. In *ARPA Human Language Technologies Workshop*, Plainsboro, N.J.

Warriner, A. B., Kuperman, V., and Brysbaert, M. (2013). Norms of valence, arousal, and dominance for 13,915 English lemmas. *Behavior Research Methods*, *45*(4), 1191–1207.

Weaver, W. (1949/1955). Translation. In Locke, W. N. and Boothe, A. D. (Eds.), *Machine Translation of Languages*, pp. 15–23. MIT Press. Reprinted from a memorandum written by Weaver in 1949.

Weinschenk, S. and Barker, D. T. (2000). *Designing Effective Speech Interfaces*. Wiley.

Weischedel, R., Hovy, E. H., Marcus, M., Palmer, M., Belvin, R., Pradhan, S., Ramshaw, L. A., and Xue, N. (2011). Ontonotes: A large training corpus for enhanced processing. In Joseph Olive, Caitlin Christianson, J. M. (Ed.), *Handbook of Natural Language Processing and Machine Translation: DARPA Global Automatic Language Exploitation*, pp. 54–63. Springer.

Weischedel, R., Meteer, M., Schwartz, R., Ramshaw, L. A., and Palmucci, J. (1993). Coping with ambiguity and unknown words through probabilistic models. *Computational Linguistics*, *19*(2), 359–382.

Weizenbaum, J. (1966). ELIZA – A computer program for the study of natural language communication between man and machine. *Communications of the ACM*, *9*(1), 36–45.

Weizenbaum, J. (1976). *Computer Power and Human Reason: From Judgement to Calculation*. W.H. Freeman and Company.

Wen, T.-H., Gašić, M., Kim, D., Mrkšić, N., Su, P.-H., Vandyke, D., and Young, S. J. (2015a). Stochastic language generation in dialogue using recurrent neural networks with convolutional sentence reranking. In *SIGDIAL 2015*, pp. 275—284.

Wen, T.-H., Gašić, M., Mrkšić, N., Su, P.-H., Vandyke, D., and Young, S. J. (2015b). Semantically conditioned lstm-based natural language generation for spoken dialogue systems. In *EMNLP 2015*.

Widrow, B. and Hoff, M. E. (1960). Adaptive switching circuits. In *IRE WESCON Convention Record*, Vol. 4, pp. 96–104.

Wiebe, J. (1994). Tracking point of view in narrative. *Computational Linguistics*, *20*(2), 233–287.

Wiebe, J. (2000). Learning subjective adjectives from corpora. In *AAAI-00*, Austin, TX, pp. 735–740.

Wiebe, J., Bruce, R. F., and O'Hara, T. P. (1999). Development and use of a gold-standard data set for subjectivity classifications. In *ACL-99*, pp. 246–253.

Wiebe, J., Wilson, T., and Cardie, C. (2005). Annotating expressions of opinions and emotions in language. *Language resources and evaluation*, *39*(2-3), 165–210.

Wierzbicka, A. (1992). *Semantics, Culture, and Cognition: University Human Concepts in Culture-Specific Configurations*. Oxford University Press.

Wierzbicka, A. (1996). *Semantics: Primes and Universals*. Oxford University Press.

Wilensky, R. (1983). *Planning and Understanding: A Computational Approach to Human Reasoning*. Addison-Wesley.

Wilks, Y. (1973). An artificial intelligence approach to machine translation. In Schank, R. C. and Colby, K. M. (Eds.), *Computer Models of Thought and Language*, pp. 114–151. W.H. Freeman.

Wilks, Y. (1975a). An intelligent analyzer and understander of English. *Communications of the ACM*, *18*(5), 264–274.

Wilks, Y. (1975b). Preference semantics. In Keenan, E. L. (Ed.), *The Formal Semantics of Natural Language*, pp. 329–350. Cambridge Univ. Press.

Wilks, Y. (1975c). A preferential, pattern-seeking, semantics for natural language inference. *Artificial Intelligence*, *6*(1), 53–74.

Williams, J. D., Raux, A., and Henderson, M. (2016). The dialog state tracking challenge series: A review. *Dialogue & Discourse*, *7*(3), 4–33.

Williams, J. D. and Young, S. J. (2007). Partially observable markov decision processes for spoken dialog systems. *Computer Speech and Language*, *21*(1), 393–422.

Wilson, T., Wiebe, J., and Hoffmann, P. (2005). Recognizing contextual polarity in phrase-level sentiment analysis. In *HLT-EMNLP-05*, pp. 347–354.

Winston, P. H. (1977). *Artificial Intelligence*. Addison Wesley.

Witten, I. H. and Bell, T. C. (1991). The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, *37*(4), 1085–1094.

Witten, I. H. and Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques* (2nd Ed.). Morgan Kaufmann.

Wittgenstein, L. (1953). *Philosophical Investigations. (Translated by Anscombe, G.E.M.)*. Blackwell.

Woods, W. A. (1978). Semantics and quantification in natural language question answering. In Yovits, M. (Ed.), *Advances in Computers*, pp. 2–64. Academic.

Woods, W. A., Kaplan, R. M., and Nash-Webber, B. L. (1972). The lunar sciences natural language information system: Final report. Tech. rep. 2378, BBN.

Woodsend, K. and Lapata, M. (2015). Distributed representations for unsupervised semantic role labeling. In *EMNLP 2015*, pp. 2482–2491.

Wu, F. and Weld, D. S. (2007). Autonomously semantifying Wikipedia. In *CIKM-07*, pp. 41–50.

Wu, F. and Weld, D. S. (2010). Open information extraction using Wikipedia. In *ACL 2010*, pp. 118–127.

Wu, Z. and Palmer, M. (1994). Verb semantics and lexical selection. In *ACL-94*, Las Cruces, NM, pp. 133–138.

Wundt, W. (1900). *Völkerpsychologie: eine Untersuchung der Entwicklungsgesetze von Sprache, Mythus, und Sitte*. W. Engelmann, Leipzig. Band II: Die Sprache, Zweiter Teil.

Xia, F. and Palmer, M. (2001). Converting dependency structures to phrase structures. In *HLT-01*, San Diego, pp. 1–5.

Xue, N. and Palmer, M. (2004). Calibrating features for semantic role labeling. In *EMNLP 2004*.

Yamada, H. and Matsumoto, Y. (2003). Statistical dependency analysis with support vector machines. In Noord, G. V. (Ed.), *IWPT-03*, pp. 195–206.

Yan, Z., Duan, N., Bao, J.-W., Chen, P., Zhou, M., Li, Z., and Zhou, J. (2016). DocChat: An information retrieval approach for chatbot engines using unstructured documents. In *ACL 2016*.

Yang, Y. and Pedersen, J. (1997). A comparative study on feature selection in text categorization. In *ICML*, pp. 412–420.

Yankelovich, N., Levow, G.-A., and Marx, M. (1995). Designing SpeechActs: Issues in speech user interfaces. In *Human Factors in Computing Systems: CHI '95 Conference Proceedings*, Denver, CO, pp. 369–376.

Yarowsky, D. (1995). Unsupervised word sense disambiguation rivaling supervised methods. In *ACL-95*, Cambridge, MA, pp. 189–196.

Yasseri, T., Kornai, A., and Kertész, J. (2012). A practical approach to language complexity: a Wikipedia case study. *PloS one*, *7*(11).

Yngve, V. H. (1955). Syntax and the problem of multiple meaning. In Locke, W. N. and Booth, A. D. (Eds.), *Machine Translation of Languages*, pp. 208–226. MIT Press.

Yngve, V. H. (1970). On getting a word in edgewise. In *CLS-70*, pp. 567–577. University of Chicago.

Young, S. J., Gašić, M., Keizer, S., Mairesse, F., Schatzmann, J., Thomson, B., and Yu, K. (2010). The Hidden Information State model: A practical framework for POMDP-based spoken dialogue management. *Computer Speech & Language*, *24*(2), 150–174.

Younger, D. H. (1967). Recognition and parsing of context-free languages in time n^3. *Information and Control*, *10*, 189–208.

Yuret, D. (1998). *Discovery of Linguistic Relations Using Lexical Attraction*. Ph.D. thesis, MIT.

Yuret, D. (2004). Some experiments with a Naive Bayes WSD system. In *Senseval-3: 3rd International Workshop on the Evaluation of Systems for the Semantic Analysis of Text*.

Zapirain, B., Agirre, E., Màrquez, L., and Surdeanu, M. (2013). Selectional preferences for semantic role classification. *Computational Linguistics*, *39*(3), 631–663.

Zavrel, J. and Daelemans, W. (1997). Memory-based learning: Using similarity for smoothing. In *ACL/EACL-97*, Madrid, Spain, pp. 436–443.

Zelle, J. M. and Mooney, R. J. (1996). Learning to parse database queries using inductive logic programming. In *AAAI-96*, pp. 1050–1055.

Zeman, D. (2008). Reusable tagset conversion using tagset drivers.. In *LREC*.

Zeman, D., Popel, M., Straka, M., Hajič, J., Nivre, J., Ginter, F., Luotolahti, J., Pyysalo, S., Petrov, S., Potthast, M., Tyers, F. M., Badmaeva, E., Gokirmak, M., Nedoluzhko, A., Cinková, S., Jr., J. H., Hlavácová, J., Kettnerová, V., Uresová, Z., Kanerva, J., Ojala, S., Missilä, A., Manning, C. D., Schuster, S., Reddy, S., Taji, D., Habash, N., Leung, H., de Marneffe, M.-C., Sanguinetti, M., Simi, M., Kanayama, H., de Paiva, V., Droganova, K., Alonso, H. M., Çöltekin, Ç., Sulubacak, U., Uszkoreit, H., Macketanz, V., Burchardt, A., Harris, K., Marheinecke, K., Rehm, G., Kayadelen, T., Attia, M., El-Kahky, A., Yu, Z., Pitler, E., Lertpradit, S., Mandl, M., Kirchner, J., Alcalde, H. F., Strnadová, J., Banerjee, E., Manurung, R., Stella, A., Shimada, A., Kwak, S., Mendonça, G., Lando, T., Nitisaroj, R., and Li, J. (2017). Conll 2017 shared task: Multilingual parsing from raw text to universal dependencies. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies, Vancouver, Canada, August 3-4, 2017*, pp. 1–19.

Zernik, U. (1991). Train1 vs. train2: Tagging word senses in corpus. In *Lexical Acquisition: Exploiting On-Line Resources to Build a Lexicon*, pp. 91–112. Lawrence Erlbaum.

Zettlemoyer, L. and Collins, M. (2005). Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Uncertainty in Artificial Intelligence, UAI'05*, pp. 658–666.

Zhang, Y. and Clark, S. (2008). A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pp. 562–571. Association for Computational Linguistics.

Zhang, Y. and Nivre, J. (2011). Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers-Volume 2*, pp. 188–193. Association for Computational Linguistics.

Zhao, H., Chen, W., Kit, C., and Zhou, G. (2009). Multilingual dependency learning: A huge feature engineering method to semantic dependency parsing. In *CoNLL-09*, pp. 55–60.

Zhao, J., Wang, T., Yatskar, M., Ordonez, V., and Chang, K.-W. (2017). Men also like shopping: Reducing gender bias amplification using corpus-level constraints. In *EMNLP 2017*.

Zhong, Z. and Ng, H. T. (2010). It makes sense: A wide-coverage word sense disambiguation system for free text. In *ACL 2010*, pp. 78–83.

Zhou, G., Su, J., Zhang, J., and Zhang, M. (2005). Exploring various knowledge in relation extraction. In *ACL-05*, Ann Arbor, MI, pp. 427–434.

Zhou, J. and Xu, W. (2015). End-to-end learning of semantic role labeling using recurrent neural networks. In *ACL 2015*, pp. 1127–1137.

Zue, V. W., Glass, J., Goodine, D., Leung, H., Phillips, M., Polifroni, J., and Seneff, S. (1989). Preliminary evaluation of the VOYAGER spoken language system. In *Proceedings DARPA Speech and Natural Language Workshop*, Cape Cod, MA, pp. 160–167.

Author Index

Subject Index