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.

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 29. 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.

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.
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** 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**: 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*

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. 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 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.

### 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 `/urg1/`).

<table>
<thead>
<tr>
<th>RE</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/woodchucks/</code></td>
<td>“interesting links to woodchucks and lemurs”</td>
</tr>
<tr>
<td><code>/a/</code></td>
<td>“Mary Ann stopped by Mona’s”</td>
</tr>
<tr>
<td><code>/l/</code></td>
<td>“You’ve left the burglar behind again!” said Nori</td>
</tr>
</tbody>
</table>

*Figure 2.1* Some simple regex searches.
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 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`.

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[wW]oodchuck/</code></td>
<td>Woodchuck or woodchuck</td>
<td>“Woodchuck”</td>
</tr>
<tr>
<td><code>[abc]/</code></td>
<td>‘a’, ‘b’, or ‘c’</td>
<td>“In uomini, in soldati”</td>
</tr>
<tr>
<td><code>[1234567890]/</code></td>
<td>any digit</td>
<td>“plenty of 7 to 5”</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[A-Z]/</code></td>
<td>an upper case letter</td>
<td>“we should call it ‘Drenched Blossoms’”</td>
</tr>
<tr>
<td><code>[a-z]/</code></td>
<td>a lower case letter</td>
<td>“my beans were impatient to be hoed!”</td>
</tr>
<tr>
<td><code>[0-9]/</code></td>
<td>a single digit</td>
<td>“Chapter 1: Down the Rabbit Hole”</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>RE</th>
<th>Match (single characters)</th>
<th>Example Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[^A-Z]/</code></td>
<td>not an upper case letter</td>
<td>“Oyfn priyitchik”</td>
</tr>
<tr>
<td><code>[^Ss]/</code></td>
<td>neither ‘S’ nor ‘s’</td>
<td>“I have no exquisite reason for’t”</td>
</tr>
<tr>
<td><code>[^\./]</code></td>
<td>not a period</td>
<td>“our resident Djinn”</td>
</tr>
<tr>
<td><code>[^e</code>&quot;]/`</td>
<td>either ‘e’ or ‘’ ’’</td>
<td>“look up ^ now”</td>
</tr>
<tr>
<td><code>a\^b/</code></td>
<td>the pattern ‘a’ ‘b’</td>
<td>“look up a‘ b now”</td>
</tr>
</tbody>
</table>

**Figure 2.4** Uses of the caret `^` for negation or just to mean `^`. We discuss below the need to escape the period by a backslash.

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.

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!

... 

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 \( aa\ldots a \), 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 /\( a^* \)/, meaning one \( a \) followed by zero or more \( a \). 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 \( aa\ldots a \) or \( ababab \) or \( bbb \).

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 of the previous character”. 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: /\( ba\ldots a^* \)/ or /\( ba\ldots a^+ \)/. 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.

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 are special characters that anchor regular expressions to particular places in a string. The most common anchors are the caret “ and the dollar sign $. The caret “ 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.

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.

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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parenthesis</td>
<td>(</td>
</tr>
<tr>
<td>Counters</td>
<td>* + ? { }</td>
</tr>
<tr>
<td>Sequences and anchors</td>
<td>the ^ my end $</td>
</tr>
<tr>
<td>Disjunction</td>
<td></td>
</tr>
</tbody>
</table>

Thus, because counters have a higher precedence than sequences,
/the* / matches theeee 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.

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]|\$/)/  

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 more than 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. Regular expression parsers are in fact 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:

$/b$([0-9]+)(\.[0-9][0-9])?\b/$

How about specifications for processor speed? Here’s a pattern for that:

$/b[0-9]+␣*(GHz|[Gg]igahertz)\b/$

Note that we use $/␣*/$ to mean “zero or more spaces” since there might always be extra spaces lying around. We also need to allow for optional fractions again (5.5 GB); note the use of ? for making the final s optional:

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

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).

<table>
<thead>
<tr>
<th>RE</th>
<th>Expansion</th>
<th>Match</th>
<th>First Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>[0-9]</td>
<td>any digit</td>
<td>Party_of_____</td>
</tr>
<tr>
<td>\D</td>
<td>[^0-9]</td>
<td>any non-digit</td>
<td>Blue_moon</td>
</tr>
<tr>
<td>\w</td>
<td>[a-zA-Z0-9_]</td>
<td>any alphanumeric/underscore</td>
<td>Daiyu</td>
</tr>
<tr>
<td>\W</td>
<td>[^\w]</td>
<td>a non-alphanumeric</td>
<td>!!!</td>
</tr>
<tr>
<td>\s</td>
<td>[\r\t\n\f]</td>
<td>whitespace (space, tab)</td>
<td></td>
</tr>
<tr>
<td>\S</td>
<td>[^\s]</td>
<td>Non-whitespace</td>
<td>in_Concord</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>zero or more occurrences of the previous char or expression</td>
</tr>
<tr>
<td>+</td>
<td>one or more occurrences of the previous char or expression</td>
</tr>
<tr>
<td>?</td>
<td>exactly zero or one occurrence of the previous char or expression</td>
</tr>
<tr>
<td>{n}</td>
<td>(n) occurrences of the previous char or expression</td>
</tr>
<tr>
<td>{n,m}</td>
<td>from (n) to (m) occurrences of the previous char or expression</td>
</tr>
<tr>
<td>{n,}</td>
<td>at least (n) occurrences of the previous char or expression</td>
</tr>
</tbody>
</table>

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 /\\/).

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>an asterisk “*”</td>
</tr>
<tr>
<td>.</td>
<td>a period “.”</td>
</tr>
<tr>
<td>?</td>
<td>a question mark</td>
</tr>
<tr>
<td>\n</td>
<td>a newline</td>
</tr>
<tr>
<td>\t</td>
<td>a tab</td>
</tr>
</tbody>
</table>

2.1.6 Regular Expression Substitution, Capture Groups, and ELIZA

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*.

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

```
/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 )`.  

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

will match *some cats like some people* but not *some people 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:

<table>
<thead>
<tr>
<th>User 1:</th>
<th>Men are all alike.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELIZA1:</td>
<td>IN WHAT WAY</td>
</tr>
<tr>
<td>User 2:</td>
<td>They’re always bugging us about something or other.</td>
</tr>
<tr>
<td>ELIZA2:</td>
<td>CAN YOU THINK OF A SPECIFIC EXAMPLE</td>
</tr>
<tr>
<td>User 3:</td>
<td>Well, my boyfriend made me come here.</td>
</tr>
<tr>
<td>ELIZA3:</td>
<td>YOUR BOYFRIEND MADE YOU COME HERE</td>
</tr>
<tr>
<td>User 4:</td>
<td>He says I’m depressed much of the time.</td>
</tr>
<tr>
<td>ELIZA4:</td>
<td>I AM SORRY TO HEAR YOU ARE DEPRESSED</td>
</tr>
</tbody>
</table>

ELIZA works by having a series or cascade of regular expression substitutions each of which matches and changes some part of the input lines. The first substitutions 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 29.

### 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.
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:

```
/(ˆ?!Volcano)[A-Za-z]+/
```

### 2.2 Words and Corpora

Before we talk about processing words, we need to decide what counts as a word. Let’s start by looking at a **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 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 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:

```
I do uh main- mainly business data processing
```

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.
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.

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.

<table>
<thead>
<tr>
<th>Corpus</th>
<th>Tokens</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shakespeare</td>
<td>884 thousand</td>
<td>31 thousand</td>
</tr>
<tr>
<td>Brown corpus</td>
<td>1 million</td>
<td>38 thousand</td>
</tr>
<tr>
<td>Switchboard telephone conversations</td>
<td>2.4 million</td>
<td>20 thousand</td>
</tr>
<tr>
<td>COCA</td>
<td>440 million</td>
<td>2 million</td>
</tr>
<tr>
<td>Google N-grams</td>
<td>1 trillion</td>
<td>13 million</td>
</tr>
</tbody>
</table>

Fig. 2.10 shows the rough numbers of types and tokens computed from some popular English 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.

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 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.

2.3.1 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 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' '
' < 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' '
' < 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 to lower case:

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

whose output is
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.

### 2.3.2 Word Tokenization and 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, 5000.50. 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.

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 multi-
word expressions like New York or rock ’n’ roll as a single token, which re-
quires a multiword expression dictionary of some sort. Tokenization is thus inti-
mately tied up with named entity detection, the task of detecting names, dates, and 
organizations (Chapter 20).

One commonly used tokenization standard is known as the Penn Treebank to-
kenization standard, used for the parsed corpora (treebanks) released by the Lin-
guistic Data Consortium (LDC), the source of many useful datasets. This standard 
separates out clitics (doesn’t becomes does plus n’t), keeps hyphenated words to-
gether, 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, “doesn’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 standard-
ization may be valuable, despite the spelling information that is lost in the normal-
ization 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 is another kind of normalization. For tasks like speech recognition 
and information retrieval, everything is mapped to lower case. For sentiment anal-
ysis and other text classification tasks, information extraction, and machine transla-
tion, 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 out-
weigh the advantage in generality that case folding provides).

In practice, since tokenization needs to be run before any other language pro-
cessing, it is important for it to be very fast. The standard method for tokeniza-
tion/normalization is therefore to use deterministic algorithms based on regular ex-
pressions compiled into very efficient finite state automata. Carefully designed de-
terministic 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. We’ll discuss this use of automata in Chapter 3.

### 2.3.3 Word Segmentation in Chinese: the MaxMatch algorithm

Some languages, including Chinese, Japanese, and Thai, do not use spaces to mark 
potential word-boundaries, and so require alternative segmentation methods. In Chi-
nese, for example, words are composed of characters known as hanzi. Each charac-
ter 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 re-
markably 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.

```plaintext
function MAXMATCH(sentence, dictionary D) returns word sequence W
    if sentence is empty
        return empty list
    for i ← length(sentence) downto 1
        firstword = first i characters of sentence
        remainder = rest of sentence
        if InDictionary(firstword, D)
            return list(firstword, MAXMATCH(remainder, dictionary))
        # no word was found, so make a one-character word
        firstword = first character of sentence
        remainder = rest of sentence
        return list(firstword, MAXMATCH(remainder, dictionary D))
```

**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
**Output:** 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 1 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 1 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 2.4 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.

The most accurate Chinese segmentation algorithms generally use statistical se-
quence models trained via supervised machine learning on hand-segmented training
sets; we’ll introduce sequence models in Chapter 10.

2.3.4 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 wood-
chucks 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 mor-
pheme 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. We’ll
introduce morphological parsing in Chapter 3.

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 mor-
phological 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:

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:

This 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

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:

ATIONAL → ATE (e.g., relational → relate)
2.4 **Minimum Edit Distance**

\[ \text{ING} \rightarrow \varepsilon \quad \text{if stem contains vowel (e.g., motoring \rightarrow motor)} \]
\[ \text{SES} \rightarrow \text{SS} \quad \text{(e.g., grasses \rightarrow grass)} \]

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):

<table>
<thead>
<tr>
<th>Errors of Commission</th>
<th>Errors of Omission</th>
</tr>
</thead>
<tbody>
<tr>
<td>organization</td>
<td>European</td>
</tr>
<tr>
<td>doing</td>
<td>analysis</td>
</tr>
<tr>
<td>numerical</td>
<td>noise</td>
</tr>
<tr>
<td>policy</td>
<td>sparse</td>
</tr>
</tbody>
</table>

### 2.3.5 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.

### 2.4 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 *giraffe*. 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.
**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.

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.

![Figure 2.12](image)

**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.

### 2.4.1 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.

![Figure 2.13](image)

**Figure 2.13** Finding the edit distance viewed as a search problem.

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 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 and forward algorithms (Chapter 9) and the CKY algorithm for parsing (Chapter 12).

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.

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.

The minimum edit distance algorithm was named by Wagner and Fischer (1974) but independently discovered by many people (summarized later, in the Historical Notes section of Chapter 9).

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:

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

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:
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. 2.2.

$$D[i, j] = \min \begin{cases} D[i-1, j] + 1 \\ D[i, j-1] + 1 \\ D[i-1, j-1] + \begin{cases} 2; & \text{if } source[i] \neq target[j] \\ 0; & \text{if } source[i] = target[j] \end{cases} \end{cases} \quad (2.2)$$

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 the word error rate (Chapter 31). 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.
2.4 Minimum Edit Distance

Table 2.4.1 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.

<table>
<thead>
<tr>
<th>Src</th>
<th>#</th>
<th>e</th>
<th>x</th>
<th>e</th>
<th>c</th>
<th>u</th>
<th>t</th>
<th>i</th>
<th>o</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>n</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>t</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>n</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>t</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>i</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>o</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>n</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
</tbody>
</table>

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.

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, after a similar diagram in Gusfield (1997). 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.

Table 2.4.2 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.

<table>
<thead>
<tr>
<th>#</th>
<th>e</th>
<th>x</th>
<th>e</th>
<th>c</th>
<th>u</th>
<th>t</th>
<th>i</th>
<th>o</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>n</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>t</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>n</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>t</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>o</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>n</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>8</td>
</tr>
</tbody>
</table>

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. We'll discuss how these weights can be estimated in Ch. 5. The Viterbi algorithm, for example, is an extension of minimum edit distance that uses probabilistic definitions of the operations. Instead of computing the “mini-
mum edit distance” between two strings, Viterbi computes the “maximum probability alignment” of one string with another. We’ll discuss this more in Chapter 9.

2.5 Summary

This chapter introduced a fundamental tool in language processing, the regular expression, and showed how to perform basic text normalization tasks including 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. (2005) 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 multi-stage... 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., “Hum-bert 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 9. You may choose a different domain than a Rogerian psychologist, if you wish, 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.


