Basic Text Processing

Regular Expressions
Regular expressions

A formal language for specifying text strings

How can we search for any of these?

- woodchuck
- woodchucks
- Woodchuck
- Woodchucks
Regular Expressions: Disjunctions

Letters inside square brackets []

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>[wW]oodchuck</td>
<td>Woodchuck, woodchuck</td>
</tr>
<tr>
<td>[1234567890]</td>
<td>Any digit</td>
</tr>
</tbody>
</table>

Ranges [A–Z]

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A–Z]</td>
<td>An upper case letter</td>
</tr>
<tr>
<td></td>
<td>Drenched Blossoms</td>
</tr>
<tr>
<td>[a–z]</td>
<td>A lower case letter</td>
</tr>
<tr>
<td></td>
<td>my beans were impatient</td>
</tr>
<tr>
<td>[0–9]</td>
<td>A single digit</td>
</tr>
<tr>
<td></td>
<td>Chapter 1: Down the Rabbit Hole</td>
</tr>
</tbody>
</table>
Regular Expressions: Negation in Disjunction

### Negations \([ ^\mathsf{Ss} ]\)
- Carat means negation only when first in []

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>[^A-Z]</td>
<td>Not an upper case letter</td>
<td>Oyfn pripetchik</td>
</tr>
<tr>
<td>[^Ss]</td>
<td>Neither ‘S’ nor ‘s’</td>
<td>I have no exquisite reason”</td>
</tr>
<tr>
<td>[^e]</td>
<td>Neither e nor ^</td>
<td>Look here</td>
</tr>
<tr>
<td>a^b</td>
<td>The pattern a carat b</td>
<td>Look up a^b now</td>
</tr>
</tbody>
</table>
Woodchuck is another name for groundhog!

The pipe | for disjunction

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>groundhog</td>
<td>woodchuck</td>
</tr>
<tr>
<td>yours</td>
<td>mine</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>[gG]roundhog</td>
<td>[Ww]oodchuck</td>
</tr>
<tr>
<td>Pattern</td>
<td>Matches</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>color?r</td>
<td>Optional previous char</td>
</tr>
<tr>
<td></td>
<td>color</td>
</tr>
<tr>
<td></td>
<td>colour</td>
</tr>
<tr>
<td>oo*h!</td>
<td>0 or more of previous char</td>
</tr>
<tr>
<td></td>
<td>oh!</td>
</tr>
<tr>
<td></td>
<td>ooh!</td>
</tr>
<tr>
<td></td>
<td>oooh!</td>
</tr>
<tr>
<td></td>
<td>ooooh!</td>
</tr>
<tr>
<td>o+h!</td>
<td>1 or more of previous char</td>
</tr>
<tr>
<td></td>
<td>oh!</td>
</tr>
<tr>
<td></td>
<td>ooh!</td>
</tr>
<tr>
<td></td>
<td>oooh!</td>
</tr>
<tr>
<td></td>
<td>ooooh!</td>
</tr>
<tr>
<td>baa+</td>
<td>baa baaa baaaa baaaaa</td>
</tr>
<tr>
<td>beg.n</td>
<td>begin begun begun begun beg3n</td>
</tr>
</tbody>
</table>
## Regular Expressions: Anchors

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>^[A-Z]</td>
<td>Palo Alto</td>
</tr>
<tr>
<td>^[^A-Za-z]</td>
<td>Hello</td>
</tr>
<tr>
<td>.$</td>
<td>The end</td>
</tr>
<tr>
<td>.$</td>
<td>The end?  The end!</td>
</tr>
</tbody>
</table>
Example

Find me all instances of the word “the” in a text.

```
the
```

Misses capitalized examples

```
[tT]he
```

Incorrectly returns other or theology

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

The process we just went through was based on fixing two kinds of errors:

1. Matching strings that we should not have matched (there, then, other)
   False positives (Type I errors)

2. Not matching things that we should have matched (The)
   False negatives (Type II errors)
Errors cont.

In NLP we are always dealing with these kinds of errors.

Reducing the error rate for an application often involves two antagonistic efforts:

- Increasing accuracy or precision (minimizing false positives)
- Increasing coverage or recall (minimizing false negatives).
Summary

Regular expressions play a surprisingly large role

- Sophisticated sequences of regular expressions are often the first model for any text processing text

For hard tasks, we use machine learning classifiers

- But regular expressions are still used for pre-processing, or as features in the classifiers
- Can be very useful in capturing generalizations
Basic Text Processing

Regular Expressions
Basic Text Processing

More Regular Expressions: Substitutions and ELIZA
Substitutions

Substitution in Python and UNIX commands:

\[ s/\text{regexp1}/\text{pattern}/ \]

\[ \text{e.g.:} \]
\[ s/\text{colour}/\text{color}/ \]
Capture Groups

• Say we want to put angles around all numbers:
  \textit{the 35 boxes} \rightarrow \textit{the <35> boxes}

• Use parens (\) to "capture" a pattern into a numbered register (1, 2, 3...)

• Use \textbackslash 1 to refer to the contents of the register

\texttt{s/((\ [0-9]+))/<\textbackslash 1>/>}
Capture groups: multiple registers

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

Matches

the faster they ran, the faster we ran

But not

the faster they ran, the faster we ate
But suppose we don't want to capture?

Parentheses have a double function: grouping terms, and capturing

Non-capturing groups: add a ?: after paren:

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

matches

- some cats like some cats

but not

- some cats like some some
Lookahead assertions

(?= pattern)  is true if pattern matches, but is zero-width; doesn't advance character pointer

(?! pattern)  true if a pattern does not match

How to match, at the beginning of a line, any single word that doesn’t start with “Volcano”:

/^(?![^Volcano])[A-Za-z]+/
Simple Application: ELIZA

Early NLP system that imitated a Rogerian psychotherapist


Uses pattern matching to match, e.g.,:

- "I need X"

and translates them into, e.g.

- "What would it mean to you if you got X?"
Simple Application: ELIZA

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
How ELIZA works

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?/
More Regular Expressions: Substitutions and ELIZA
Basic Text Processing

Words and Corpora
How many words in a sentence?

"I do uh main- mainly business data processing"
  ◦ Fragments, filled pauses

"Seuss’s cat in the hat is different from other cats!"
  ◦ **Lemma**: same stem, part of speech, rough word sense
    ◦ *cat* and *cats* = same lemma
  ◦ **Wordform**: the full inflected surface form
    ◦ *cat* and *cats* = different wordforms
How many words in a sentence?

they lay back on the San Francisco grass and looked at the stars and their

**Type**: an element of the vocabulary.

**Token**: an instance of that type in running text.

How many?
- 15 tokens (or 14)
- 13 types (or 12) (or 11?)
How many words in a corpus?

$N =$ number of tokens

$V =$ vocabulary = set of types, $|V|$ is size of vocabulary

Heaps Law = Herdan's Law = $|V| = kN^\beta$ where often $0.67 < \beta < 0.75$

i.e., vocabulary size grows with $>\sqrt{\text{square root of the number of word tokens}}$

<table>
<thead>
<tr>
<th>Corpus</th>
<th>Tokens</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switchboard phone conversations</td>
<td>2.4 million</td>
<td>20 thousand</td>
</tr>
<tr>
<td>Shakespeare</td>
<td>884,000</td>
<td>31 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>

Figure 2.11 Rough numbers of types and tokens for some English language 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 Heaps' Law (Heaps, 1978) or Herdan's Law (Herdan, 1960) after its discoverers (in linguistics and information retrieval respectively). It is shown in Eq. 2.1, where $k$ and $\beta$ are positive constants, and $0 < \beta < 1$.

$|V| = kN^\beta$ (2.1)
Corpora

Words don't appear out of nowhere!

A text is produced by

• a specific writer(s),
• at a specific time,
• in a specific variety,
• of a specific language,
• for a specific function.
Corpora vary along dimension like

- **Language**: 7097 languages in the world
- **Variety**, like African American Language varieties.
  - AAE Twitter posts might include forms like "jont" (I don't)
- **Code switching**, e.g., Spanish/English, Hindi/English:
  - S/E: 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:]
  - H/E: dost tha or ra- hega ... dont worry ... but dherya rakhe
    - [“he was and will remain a friend ... don’t worry ... but have faith”]
- **Genre**: newswire, fiction, scientific articles, Wikipedia
- **Author Demographics**: writer's age, gender, ethnicity, SES
Motivation:
- Why was the corpus collected?
- By whom?
- Who funded it?

Situation: In what situation was the text written?

Collection process: If it is a subsample how was it sampled? Was there consent? Pre-processing?

+ Annotation process, language variety, demographics, etc.
Words and Corpora
Basic Text Processing

Word tokenization
Text Normalization

Every NLP task requires text normalization:

1. Tokenizing (segmenting) words
2. Normalizing word formats
3. Segmenting sentences
Space-based tokenization

A very simple way to tokenize
- For languages that use space characters between words
- Arabic, Cyrillic, Greek, Latin, etc., based writing systems
- Segment off a token between instances of spaces

Unix tools for space-based tokenization
- The "tr" command
- Inspired by Ken Church's UNIX for Poets
- Given a text file, output the word tokens and their frequencies
Simple Tokenization in UNIX  
(Inspired by Ken Church’s UNIX for Poets.)

Given a text file, output the word tokens and their frequencies

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

1945 A
72 AARON
19 ABBESS
5 ABBOT
...
...
25 Aaron
6 Abate
1 Abates
5 Abbess
6 Abbey
3 Abbot
...
The first step: tokenizing

tr -sc 'A-Za-z' '
' < shakes.txt | head

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

THE SONNETS by William Shakespeare From fairest creatures We
The second step: sorting

```
tr -sc 'A-Za-z' '\n' < shakes.txt | sort | head
```

A
A
A
A
A
A
A
A
A
A
A
A
A
...

...
More counting

Merging upper and lower case

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

Sorting the counts

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

```
23243 the
22225 i
18618 and
16339 to
15687 of
12780 a
12163 you
10839 my
10005 in
8954 d
```

What happened here?
Issues in Tokenization

Can't just blindly remove punctuation:
- m.p.h., Ph.D., AT&T, cap’n
- prices ($45.55)
- dates (01/02/06)
- URLs (http://www.stanford.edu)
- hashtags (#nlproc)
- email addresses (someone@cs.colorado.edu)

Clitic: a word that doesn't stand on its own
- "are" in we're, French "je" in j'ai, "le" in l'honneur

When should multiword expressions (MWE) be words?
- New York, rock ’n’ roll
Tokenization in NLTK

Bird, Loper and Klein (2009), *Natural Language Processing with Python*. O’Reilly

```python
>>> text = 'That U.S.A. poster-print costs $12.40...

>>> pattern = r''(?x) # set flag to allow verbose regexps
... ([A-Z]\.)+ # abbreviations, e.g. U.S.A.
... | \w+(-\w+)* # words with optional internal hyphens
... | \$?\d+(\.\d+)?%? # currency and percentages, e.g. $12.40, 82%
... | \.
... | [\[\.,;"'\']?():-_'"] # these are separate tokens; includes ], [
... ''

>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```
Tokenization in languages without spaces

Many languages (like Chinese, Japanese, Thai) don't use spaces to separate words!

How do we decide where the token boundaries should be?
Word tokenization in Chinese

Chinese words are composed of characters called "hanzi" (or sometimes just "zi")

Each one represents a meaning unit called a morpheme.

Each word has on average 2.4 of them.

But deciding what counts as a word is complex and not agreed upon.
How to do word tokenization in Chinese?

姚明进入总决赛 “Yao Ming reaches the finals”
How to do word tokenization in Chinese?

姚明进入总决赛 “Yao Ming reaches the finals”

3 words?
姚明 进入 总决赛
YaoMing reaches finals
How to do word tokenization in Chinese?

姚明进入总决赛 “Yao Ming reaches the finals”

3 words?
姚明 进入 总决赛
YaoMing reaches finals

5 words?
姚明 进入 总决赛
Yao Ming reaches overall finals
How to do word tokenization in Chinese?

姚明进入总决赛 “Yao Ming reaches the finals”

3 words?
姚明进入总决赛
YaoMing reaches finals

5 words?
姚明进入总决赛
Yao Ming reaches overall finals

7 characters? (don't use words at all):
姚明进入总决赛
Yao Ming enter enter overall decision game
Word tokenization / segmentation

So in Chinese it's common to just treat each character (zi) as a token.

- So the segmentation step is very simple

In other languages (like Thai and Japanese), more complex word segmentation is required.

- The standard algorithms are neural sequence models trained by supervised machine learning.
Basic Text Processing

Word tokenization
Basic Text Processing

Byte Pair Encoding
Another option for text tokenization

Instead of

• white-space segmentation
• single-character segmentation

Use the data to tell us how to tokenize.

Subword tokenization (because tokens can be parts of words as well as whole words)
Subword tokenization

Three common algorithms:

- **Byte-Pair Encoding (BPE)** (Sennrich et al., 2016)
- **Unigram language modeling tokenization** (Kudo, 2018)
- **WordPiece** (Schuster and Nakajima, 2012)

All have 2 parts:

- A token **learner** that takes a raw training corpus and induces a vocabulary (a set of tokens).
- A token **segmenter** that takes a raw test sentence and tokenizes it according to that vocabulary.
Byte Pair Encoding (BPE) token learner

Let vocabulary be the set of all individual characters

\[= \{A, B, C, D, \ldots, a, b, c, d, \ldots\}\]

Repeat:

- Choose the two symbols that are most frequently adjacent in the training corpus (say 'A', 'B')
- Add a new merged symbol 'AB' to the vocabulary
- Replace every adjacent 'A' 'B' in the corpus with 'AB'.

Until \( k \) merges have been done.
BPE token learner algorithm

\begin{verbatim}
function BYTE-PAIR ENCODING(strings C, number of merges k) returns vocab V

V ← all unique characters in C # initial set of tokens is characters
for i = 1 to k do # merge tokens til k times
    t_L, t_R ← Most frequent pair of adjacent tokens in C
    t_NEW ← t_L + t_R # make new token by concatenating
    V ← V + t_NEW # update the vocabulary
    Replace each occurrence of t_L, t_R in C with t_NEW # and update the corpus
return V
\end{verbatim}

2.4.4 Word Normalization, Lemmatization and Stemming

Word normalization is the task of putting words/tokens in a standard format, choosing a single normal form 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 or information extraction about the US, we might want to see information from documents whether they mention the US or the USA.

Case folding is another kind of normalization. Mapping everything to lower case means that Woodchuck and woodchuck are represented identically, which is very helpful for generalization in many tasks, such as information retrieval or speech recognition. For sentiment analysis and other text classification tasks, information extraction, and machine translation, by contrast, case can be quite helpful and case folding is generally not done. This is because maintaining the difference between, for example, US the country and us the pronoun can outweigh the advantage in generalization that case folding would have provided for other words.

For many natural language processing situations we also want two morphologically different forms of a word to behave similarly. For example in web search, someone may type the string woodchucks but a useful system might want to also return pages that mention woodchuck with no s. This is especially common in morphologically complex languages like Russian, where for example the word Moscow has different endings in the phrases Moscow, of Moscow, to Moscow, and so on.
Byte Pair Encoding (BPE) Addendum

Most subword algorithms are run inside space-separated tokens.

So we commonly first add a special end-of-word symbol '__' before space in training corpus

Next, separate into letters.
BPE token learner

Original (very fascinating 😐) corpus:

low low low low low lowest lowest newer newer newer newer newer newer newer newer wider wider wider wider wider wider new new

Add end-of-word tokens, resulting in this vocabulary:

vocabulary
__, d, e, i, l, n, o, r, s, t, w
BPE token learner

<table>
<thead>
<tr>
<th>corpus</th>
<th>vocabulary</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td>_, d, e, i, l, n, o, r, s, t, w</td>
</tr>
<tr>
<td>lowest</td>
<td></td>
</tr>
<tr>
<td>newer</td>
<td></td>
</tr>
<tr>
<td>wider</td>
<td></td>
</tr>
<tr>
<td>new</td>
<td></td>
</tr>
</tbody>
</table>

Merge `er` to `er`

<table>
<thead>
<tr>
<th>corpus</th>
<th>vocabulary</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td>_, d, e, i, l, n, o, r, s, t, w, er</td>
</tr>
<tr>
<td>lowest</td>
<td></td>
</tr>
<tr>
<td>newer</td>
<td></td>
</tr>
<tr>
<td>wider</td>
<td></td>
</tr>
<tr>
<td>new</td>
<td></td>
</tr>
<tr>
<td>new</td>
<td></td>
</tr>
</tbody>
</table>
BPE

corpus                        vocabulary
5  low _            __, d, e, i, l, n, o, r, s, t, w, er
2  lowest _       __, d, e, i, l, n, o, r, s, t, w, er, er_
6  newer _        __, d, e, i, l, n, o, r, s, t, w, er, er_
3  wider _        __, d, e, i, l, n, o, r, s, t, w, er, er_
2  new _          __, d, e, i, l, n, o, r, s, t, w, er, er_

Merge er _ to er_

corpus                        vocabulary
5  low _            __, d, e, i, l, n, o, r, s, t, w, er
2  lowest _       __, d, e, i, l, n, o, r, s, t, w, er, er_
6  newer _        __, d, e, i, l, n, o, r, s, t, w, er, er_
3  wider _        __, d, e, i, l, n, o, r, s, t, w, er, er_
2  new _          __, d, e, i, l, n, o, r, s, t, w, er, er_
BPE

<table>
<thead>
<tr>
<th>corpus</th>
<th>vocabulary</th>
</tr>
</thead>
<tbody>
<tr>
<td>low_</td>
<td><em>, d, e, i, l, n, o, r, s, t, w, er, er</em>, ne</td>
</tr>
<tr>
<td>lowest_</td>
<td></td>
</tr>
<tr>
<td>newer_</td>
<td></td>
</tr>
<tr>
<td>wider_</td>
<td></td>
</tr>
<tr>
<td>new_</td>
<td></td>
</tr>
</tbody>
</table>

**Merge newer to ne**

<table>
<thead>
<tr>
<th>corpus</th>
<th>vocabulary</th>
</tr>
</thead>
<tbody>
<tr>
<td>low_</td>
<td><em>, d, e, i, l, n, o, r, s, t, w, er, er</em>, ne</td>
</tr>
<tr>
<td>lowest_</td>
<td></td>
</tr>
<tr>
<td>newer_</td>
<td></td>
</tr>
<tr>
<td>wider_</td>
<td></td>
</tr>
<tr>
<td>new_</td>
<td></td>
</tr>
<tr>
<td>ne</td>
<td></td>
</tr>
</tbody>
</table>
The next merges are:

<table>
<thead>
<tr>
<th>Merge</th>
<th>Current Vocabulary</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ne, w)</td>
<td><em>, d, e, i, l, n, o, r, s, t, w, er, er</em>_, ne, new</td>
</tr>
<tr>
<td>(l, o)</td>
<td><em>, d, e, i, l, n, o, r, s, t, w, er, er</em>_, ne, new, lo</td>
</tr>
<tr>
<td>(lo, w)</td>
<td><em>, d, e, i, l, n, o, r, s, t, w, er, er</em>_, ne, new, lo, low</td>
</tr>
<tr>
<td>(new, er__)</td>
<td><em>, d, e, i, l, n, o, r, s, t, w, er, er</em>_, ne, new, lo, low, newer__</td>
</tr>
<tr>
<td>(low, _)</td>
<td><em>, d, e, i, l, n, o, r, s, t, w, er, er</em>_, ne, new, lo, low, newer__, low__</td>
</tr>
</tbody>
</table>
BPE token **segmenter** algorithm

On the test data, run each merge learned from the training data:

- Greedily
- In the order we learned them
- (test frequencies don't play a role)

So: merge every `er` to `er`, then merge `er_` to `er_`, etc.

Result:

- Test set "new e r _" would be tokenized as a full word
- Test set "low e r _" would be two tokens: "low er_"
Properties of BPE tokens

Usually include frequent words
And frequent subwords

• Which are often morphemes like -est or –er

A morpheme is the smallest meaning-bearing unit of a language

• unlikeliest has 3 morphemes un-, likely, and -est
Basic Text Processing

Byte Pair Encoding
Basic Text Processing

Word Normalization and other issues
Word Normalization

Putting words/tokens in a standard format

- U.S.A. or USA
- uhhuh or uh-huh
- Fed or fed
- am, is, be, are
Case folding

Applications like IR: reduce all letters to lower case
- Since users tend to use lower case
- Possible exception: upper case in mid-sentence?
  - e.g., *General Motors*
  - *Fed* vs. *fed*
  - *SAIL* vs. *sail*

For sentiment analysis, MT, Information extraction
- Case is helpful (*US* versus *us* is important)
Lemmatization

Represent all words as their lemma, their shared root = dictionary headword form:

- *am, are, is* $\rightarrow$ *be*
- *car, cars, car's, cars'* $\rightarrow$ *car*
- Spanish *quiero* (‘I want’), *quieres* (‘you want’)
  $\rightarrow$ *querer* ‘want'
- *He is reading detective stories*
  $\rightarrow$ *He be read detective story*
Lemmatization is done by Morphological Parsing

Morphemes:
- The small meaningful units that make up words
- **Stems**: The core meaning-bearing units
- **Affixes**: Parts that adhere to stems, often with grammatical functions

Morphological Parsers:
- Parse *cats* into two morphemes *cat* and *s*
- Parse Spanish *amaren* (‘if in the future they would love’) into morpheme *amar* ‘to love’, and the morphological features 3PL and *future subjunctive*. 
Stemming

Reduce terms to stems, chopping off affixes crudely

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.
Porter Stemmer

Based on a series of rewrite rules run in series
- A cascade, in which output of each pass fed to next pass

Some sample rules:

- ATIONAL → ATE (e.g., relational → relate)
- ING → ε if stem contains vowel (e.g., motoring → motor)
- SSES → SS (e.g., grasses → grass)
Dealing with complex morphology is necessary for many languages

- e.g., the Turkish word:
  - Uygarlastiramadiklarimizdanmissinizcasina
  - `(behaving) as if you are among those whom we could not civilize’
  - Uygar `civilized’ + las `become’
    + tir `cause’ + ama `not able’
    + dik `past’ + lar ‘plural’
    + imiz ‘p1pl’ + dan ‘abl’
    + mis ‘past’ + siniz ‘2pl’ + casina ‘as if’
Sentence Segmentation

!, ? mostly unambiguous but **period** “.” is very ambiguous
  ◦ Sentence boundary
  ◦ Abbreviations like Inc. or Dr.
  ◦ Numbers like .02% or 4.3

Common algorithm: Tokenize first: use rules or ML to classify a period as either (a) part of the word or (b) a sentence-boundary.
  ◦ An abbreviation dictionary can help

Sentence segmentation can then often be done by rules based on this tokenization.
Basic Text Processing

Word Normalization and other issues