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?

corpus
corpora

> 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:

utterance

> 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.

disfluency
fragment
filled pause

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 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 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' '\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.

2.3.2 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 20).

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*. 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 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 re-

hanzi

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.

maximum matching

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 D) 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, D)
            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 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 "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 2.4 how to compute edit distance. Even in Chinese, however, MaxMatch has problems, for example dealing

word error rate

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 **sequence 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 *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*. We'll introduce morphological parsing in Chapter 3.

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
```

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:

cascade

ATIONAL → ATE (e.g., relational → relate)

ING → ε if stem contains vowel (e.g., motoring → motor)
SSES → SS (e.g., grasses → 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):

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

2.3.5 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.