

Elementary Language Processing: Tokenizing Text and Classifying Words

Steven Bird

Ewan Klein

Edward Loper

2005-11-26

Version: 0.6

Revision: 1.19

Copyright: © 2001-2005 University of Pennsylvania

License: Creative Commons Attribution-NonCommercial-ShareAlike License

Note

This is a draft. Please send any feedback to the authors.

1 Introduction

Texts are usually represented in a computer as files containing a potentially long sequence of characters. For most kinds of linguistic processing, we need to identify and categorize the words of the text. This turns out to be a non-trivial task. In this chapter we introduce *tokens* as the building blocks of text, and show how texts can be *tokenized*. Next we consider the categorization of tokens according to their parts-of-speech, and do some preliminary exploration of the Brown Corpus, a collection of over a million words of tagged English text. Along the way we take a look at some interesting applications: generating random text, classifying words automatically, and analyzing the modal verbs of different genres.

2 Tokens: the building blocks of text

How do we know that piece of text is a *word*, and how do we represent words and associated information in a machine? It might seem needlessly picky to ask what a word is. Can't we just say that a word is a string of characters which has white space before and after it? However, it turns out that things are quite a bit more complex. To get a flavour of the problems, consider the following text from the Wall Street Journal:

Paragraph 12 from ‘‘wsj_0034’’

It's probably worth paying a premium for funds that invest in markets that are partially closed to foreign investors, such as South Korea, some specialists say. But some European funds recently have skyrocketed; Spain Fund has surged to a startling 120% premium. It has been targeted by Japanese investors as a good long-term play tied to 1992's European economic integration. And several new funds that aren't even fully invested yet have jumped to trade at big premiums.

"I'm very alarmed to see these rich valuations," says Smith Barney's Mr. Porter.

Let's start with the string `aren't`. According to our naive definition, it counts as only one word. But consider a situation where we wanted to check whether all the words in our text occurred in a dictionary, and our dictionary had entries for `are` and `not`, but not for `aren't`. In this case, we would probably be happy to say that `aren't` is a contraction of two distinct words.

If we take our naive definition of word literally (as we should, if we are thinking of implementing it in code), then there are some other minor but real problems. For example, assuming our file consists of a number of separate lines, as in the WSJ text, then all the words which come at the beginning of a line will fail to be preceded by whitespace (unless we treat the newline character as a whitespace). Second, according to our criterion, punctuation symbols will form part of words; that is, a string like `investors`, will also count as a word, since there is no whitespace between `investors` and the following comma. Consequently, we run the risk of failing to recognise that `investors`, (with appended comma) is a token of the same type as `investors` (without appended comma). More importantly, we would like punctuation to be a "first-class citizen" for tokenization and subsequent processing. For example, we might want to implement a rule which says that a word followed by a period is likely to be an abbreviation if the immediately following word has a lowercase initial. However, to formulate such a rule, we must be able to identify a period as a token in its own right.

A slightly different challenge is raised by examples such as the following (drawn from the MedLine corpus):

1. This is a alpha-galactosyl-1,4-beta-galactosyl-specific adhesin.
2. The corresponding free cortisol fractions in these sera were 4.53 +/- 0.15% and 8.16 +/- 0.23%, respectively.

In these cases, we encounter terms which are unlikely to be found in any general purpose English lexicon. Moreover, we will have no success in trying to syntactically analyse these strings using a standard grammar of English. Now for some applications, we would like to "bundle up" expressions such as `alpha-galactosyl-1,4-beta-galactosyl-specific adhesin` and `4.53 +/- 0.15%` so that they are presented as unanalysable atoms to the parser. That is, we want to treat them as single "words" for the purposes of subsequent processing. The upshot is that, even if we confine our attention to English text, the question of what we treat as word may depend a great deal on what our purposes are.

Note

If we turn to languages other than English, segmenting words can be even more of a challenge. For example, in Chinese orthography, characters correspond to monosyllabic morphemes. Many morphemes are words in their own right, but many words contain more than one morpheme; most of them consist of two morphemes. However, there is no visual representation of word boundaries in Chinese text.

Let's look in more detail at the words in the WSJ text. Suppose we use white space as the delimiter for words, and then list all the words of the text in alphabetical order; we would expect to get something like the following:

120, 1992, And, Barney, But, European, European, Fund, I, It, It,
Japanese, Korea, Mr, Porter, Smith, South, Spain, a, a, a, ...

Now, if we ask a program utility to tell us how many words there in the text, it will probably return the answer: 90. This calculation depends on treating each of the three occurrences of `a` as a separate word. Yet what do we mean by saying that is some object `a` which occurs three times? Are there three words `a` or just one? We can in fact answer "Both" if we draw a distinction between a word *token* versus a word *type*. A word type is somewhat abstract; it's what we're talking about when we say that we know the meaning of the word `deprecate`, or when we say that the words `barf` and `vomit` are synonyms. On the other hand, a word token is something which exists in time and space. For example, we could talk about my uttering a token of the word `grunge` in Edinburgh on July 14, 2003; equally,

we can say that the second word token in the WSJ text is a token of the word type **probably**, or that there are two tokens of the type **European** in the text. More generally, we want to say that there are 90 word tokens in the WSJ text, but only 76 word types.

The terms *token* and *type* can also be applied to other linguistic entities. For example, a *sentence token* is an individual occurrence of a sentence; but a *sentence type* is an abstract sentence, without context. If someone repeats a sentence twice, they have uttered two sentence tokens, but only one sentence type. When the kind of token or type is obvious from context, we will simply use the terms token and type.

2.1 Representing tokens

When written language is stored in a computer file it is normally represented as a sequence or *string* of characters. That is, in a standard text file, individual words are strings, sentences are strings, and indeed the whole text is one long string. The characters in a string don't have to be just the ordinary alphanumerics; strings can also include special characters which represent space, tab and newline.

Most computational processing is performed above the level of characters. In compiling a programming language, for example, the compiler expects its input to be a sequence of tokens that it knows how to deal with; for example, the classes of identifiers, string constants and numerals. Analogously, a parser will expect its input to be a sequence of word tokens rather than a sequence of individual characters. At its simplest, then, tokenization of a text involves searching for locations in the string of characters containing whitespace (space, tab, or newline) or certain punctuation symbols, and breaking the string into word tokens at these points. For example, suppose we have a file containing the following two lines:

```
The cat climbed
the tree.
```

From the parser's point of view, this file is just a string of characters:

```
'The_cat_climbed\n_the_tree.'
```

Note that we use single quotes to delimit strings, “_” to represent space and “n” to represent newline.

As we just pointed out, to tokenize this text for consumption by the parser, we need to explicitly indicate which substrings are words. One convenient way to do this in Python is to split the string into a *list* of words, where each word is a string, such as 'dog'.¹ In Python, lists are printed as a series of objects (in this case, strings), surrounded by square brackets and separated by commas:

```
>>> words = ['the', 'cat', 'climbed', 'the', 'tree']
>>> words
['the', 'cat', 'climbed', 'the', 'tree']
```

Notice that we have introduced a new variable `words` which is bound to the list, and that we entered the variable on a new line to check its value.

To summarize, we have just illustrated how, at its simplest, tokenization of a text can be carried out by converting the single string representing the text into a list of strings, each of which corresponds to a word.

3 Tokenization

Many natural language processing tasks involve analyzing texts of varying sizes, ranging from single sentences to very large corpora. There are a number of ways to represent texts using NLTK. The simplest is as a single string. These strings can be loaded directly from files:

¹ We say “convenient” because Python makes it easy to iterate through a list, processing the items one by one.

```
>>> text_str = open('corpus.txt').read()
>>> text_str
'Hello world. This is a test file.\n'
```

However, as we noted in [representing.tokens](#), it is usually preferable to represent a text as a list of tokens. These lists are typically created using a *tokenizer*, such as `tokenize.whitespace` which splits strings into words at whitespaces:

```
>>> from nltk_lite import tokenize
>>> text = 'Hello world. This is a test string.'
>>> list(tokenize.whitespace(text))
['Hello', 'world.', 'This', 'is', 'a', 'test', 'string.']
```

Note

By “whitespace”, we mean not only interword space, but also tab and line-end.

Note that tokenization may normalize the text, mapping all words to lowercase, expanding contractions, and possibly even stemming the words. An example for stemming is shown below:

```
>>> text = 'stemming can be fun and exciting'
>>> tokens = tokenize.whitespace(text)
>>> porter = tokenize.PorterStemmer()
>>> for token in tokens:
...     print porter.stem(token),
stem can be fun and excit
```

Tokenization based on whitespace is too simplistic for most applications; for instance, it fails to separate the last word of a phrase or sentence from punctuation characters, such as comma, period, exclamation mark and question mark. As its name suggests, `tokenize.regexp` employs a regular expression to determine how text should be split up. This regular expression specifies the characters that can be included in a valid word. To define a tokenizer that includes punctuation as separate tokens, we could use:

```
>>> text = '''Hello. Isn't this fun?'''
>>> pattern = r'\w+|[\^\w\s]+'
>>> list(tokenize.regexp(text, pattern))
['Hello', '.', 'Isn', "'", 't', 'this', 'fun', '?']
```

Tip

Recall that `\w+|[\^\w\s]+` is a disjunction of two subexpressions, namely `w+` and `[\^\w\s]+`. The first of these matches one or more “word” characters; i.e., characters other than whitespace or punctuation. The second pattern is a negated range expression; it matches on or more characters which are not word characters (i.e., not a match for `\w`) and not a whitespace character (i.e., not a match for `\s`).

Tip

The regular expression in this example will match a sequence consisting of one or more word characters `\w+`. It will also match a sequence consisting of one or more punctuation characters (or non-word, non-space characters `[\^\w\s]+`).

There are a number of ways we might want to improve this regular expression. For example, it currently breaks the string `'$22.50'` into four tokens; but we might want it to include this as a single token. One approach to making this change would be to add a new clause to the tokenizer’s regular expression, which is specialized for handling strings of this form:

```
>>> text = 'That poster costs $22.40.'
>>> pattern = r'\w+|\$\d+\.\d+|[\^\w\s]+'
>>> list(tokenize.regexp(text, pattern))
['That', 'poster', 'costs', '$22.40', '.']
```

It is sometimes more convenient to write a regular expression matching the material that appears *between* tokens, such as whitespace and punctuation. The `tokenize()` function constructor permits an optional boolean parameter `gaps`; when set to `True` the pattern is matched against the gaps. For example, here is how `whitespaceTokenize()` is defined:

```
>>> list(tokenize.regexp(text, pattern=r'\s+', gaps=True))
['That', 'poster', 'costs', '$22.40.']
```

The `nltk_lite.corpora` package provides ready access to several corpora included with NLTK, along with built-in tokenizers. For example, `brown.tagged()` is an iterator over tagged sentences from the Brown Corpus. We use `extract()` to extract a sentence of interest:

```
>>> from nltk_lite.corpora import brown, extract
>>> print extract(0, brown.tagged('a'))
[('The', 'at'), ('Fulton', 'np-t1'), ('County', 'nn-t1'), ('Grand', 'jj-t1'), ('Jury', 'nn-t1')]
```

In particular, `brown.tagged()` doesn't just read the relevant text into a string, it also invokes the appropriate tokenizer.

4 Counting Tokens

Perhaps the simplest thing to do once we have pulled tokens out of text is to count them. We can do this as follows, to compare the lengths of the English and Finnish translations of the book of Genesis:

```
>>> from nltk_lite.corpora import genesis
>>> len(list(genesis.raw('english-kjv')))
38240
>>> len(list(genesis.raw('finnish')))
26595
```

We can do more sophisticated counting using *frequency distribution*. In general, a frequency distribution records the number of times each outcome of an experiment has occurred. For instance, a frequency distribution could be used to record the frequency of each word in a document. Frequency distributions are generally initialized by repeatedly running an experiment, and incrementing the count for a sample every time it is an outcome of the experiment. The following program produces a frequency distribution that records how often each word occurs in a text, and prints the most frequently occurring word:

```
>>> from nltk_lite.probability import FreqDist
>>> fd = FreqDist()
>>> for token in genesis.raw():
...     fd.inc(token)
>>> fd.max()
'the'
```

Once we construct a frequency distribution that records the outcomes of an experiment, we can use it to examine a number of interesting properties of the experiment. These properties are summarized below:

Frequency Distribution Module		
Name	Sample	Description
Count	fd.count('the')	number of times a given sample occurred
Frequency	fd.freq('the')	frequency of a given sample
N	fd.N()	number of samples
Samples	fd.samples()	list of distinct samples recorded
Max	fd.max()	sample with the greatest number of outcomes

We can use a `FreqDist` to examine the distribution of word lengths in a corpus. For each word, we find its length, and increment the count for words of this length.

```
>>> def length_dist(text):
...     fd = FreqDist()                # initialize an empty frequency distribution
...     for token in genesis.raw(text): # for each token
...         fd.inc(len(token))         # found another word with this length
...     for i in range(15):           # for each length from 0 to 14
...         print "%2d" % int(100*fd.freq(i)), # print the percentage of words with this length
...     print

>>> length_dist('english-kjv')
 0  2 14 28 21 13  7  5  2  2  0  0  0  0  0
>>> length_dist('finnish')
 0  0  9  6 10 16 16 12  9  6  3  2  2  1  0
```

A *condition* specifies the context in which an experiment is performed. Often, we are interested in the effect that conditions have on the outcome for an experiment. For example, we might want to examine how the distribution of a word's length (the outcome) is affected by the word's initial letter (the condition). Conditional frequency distributions provide a tool for exploring this type of question.

A *conditional frequency distribution* is a collection of frequency distributions for the same experiment, run under different conditions. The individual frequency distributions are indexed by the condition.

```
>>> from nltk_lite.probability import ConditionalFreqDist
>>> cfdist = ConditionalFreqDist()

>>> for text in genesis.items:
...     for word in genesis.raw(text):
...         cfdist[text].inc(len(word))
```

To plot the results, we construct a list of points, where the x coordinate is the word length, and the y coordinate is the frequency with which that word length is used:

```
>>> for cond in cfdist.conditions():
...     wordlens = cfdist[cond].samples()
...     wordlens.sort()
...     points = [(i, cfdist[cond].freq(i)) for i in wordlens]
```

We can plot these points using the `Plot` function defined in `nltk_lite.draw.plot`, as follows: `Plot(points).mainloop()`

5 An Application: Predicting the Next Word

Conditional frequency distributions are often used for prediction. *Prediction* is the problem of deciding a likely outcome for a given run of an experiment. The decision of which outcome to predict is usually

based on the context in which the experiment is performed. For example, we might try to predict a word's text (outcome), based on the text of the word that it follows (context).

To predict the outcomes of an experiment, we first examine a representative *training corpus*, where the context and outcome for each run of the experiment are known. When presented with a new run of the experiment, we simply choose the outcome that occurred most frequently for the experiment's context.

We can use a `ConditionalFreqDist` to find the most frequent occurrence for each context. First, we record each outcome in the training corpus, using the context that the experiment was run under as the condition. Then, we can access the frequency distribution for a given context with the indexing operator, and use the `max()` method to find the most likely outcome.

We will now use a `ConditionalFreqDist` to predict the most likely next word in a text. To begin, we load a corpus from a text file, and create an empty `ConditionalFreqDist`:

```
>>> from nltk_lite.probability import ConditionalFreqDist
>>> cfdist = ConditionalFreqDist()
```

We then examine each token in the corpus, and increment the appropriate sample's count. We use the variable `prev` to record the previous word.

```
>>> prev = None
>>> for word in genesis.raw():
...     cfdist[prev].inc(word)
...     prev = word
```

Note

Sometimes the context for an experiment is unavailable, or does not exist. For example, the first token in a text does not follow any word. In these cases, we must decide what context to use. For this example, we use `None` as the context for the first token. Another option would be to discard the first token.

Once we have constructed a conditional frequency distribution for the training corpus, we can use it to find the most likely word for any given context. For example, taking the word `living` as our context, we can inspect all the words that occurred in that context.

```
>>> word = 'living'
>>> cfdist['living'].samples()
['creature,', 'substance', 'soul.', 'thing', 'thing,', 'creature']
```

We can set up a simple loop to generate text: we set an initial context, picking the most likely token in that context as our next word, and then using that word as our new context:

```
>>> word = 'living'
>>> for i in range(20):
...     print word,
...     word = cfdist[word].max()
living creature that he said, I will not be a wife of the land of the land of the land
```

This simple approach to text generation tends to get stuck in loops, as demonstrated by the text generated above. A more advanced approach would be to randomly choose each word, with more frequent words chosen more often.

6 Word Classes and Parts of Speech

In the preceding sections, we have pretty much treated all words alike: things are either tokens or not. However, for many applications, we want to distinguish between *different kinds* of tokens. For

example, we want to be able to make explicit that we have recognized one string as an ordinary lexical item, another as numerical expression, and yet another as a punctuation character. Moreover, we want to distinguish different kinds of lexical items. In fact, there is a long tradition within linguistics of classifying words into categories called *parts of speech*. These are sometimes also called word classes or *lexical categories*. Familiar examples are *noun*, *verb*, *preposition*, *adjective* and *adverb*. In this section we present the standard criteria for categorising words in this way, then discuss the main classes of words in English.

6.1 Categorising Words

How do we decide what category a word should belong to? In general, linguists invoke three kinds of criteria for making the decision: formal; syntactic (or distributional); notional (or semantic). A *formal* criterion is one which looks at the internal structure of a word. For example, *-ness* is a suffix which combines with an adjective to produce a noun. Examples are *happy* > *happiness*, *ill* > *illness*.² So if we encounter a word which ends in *-ness*, this is very likely to be a noun. </para>

A *syntactic* criterion refers to the syntactic contexts in which a word can occur. For example, assume that we have already determined the category of nouns. Then we might say that a syntactic criterion for an adjective in English is that it can occur immediately before a noun, or immediately following the words *be* or *very*. According to these tests, *near* should be categorised as an adjective:

1. the near window
2. The end is (very) near.

A familiar example of a *notional* criterion is that a noun is “the name of a person, place or thing”. Within modern linguistics, notional criteria for word classes have been viewed with considerable suspicion, mainly because they are hard to formalise. Nevertheless, notional criteria underpin many of our intuitions about word classes, and enable us to make a good guess about the categorisation of words in languages that we are unfamiliar with; that is, if all we know about the Dutch *verjaardag* is that it means the same as the English word *birthday*, then we can guess that *verjaardag* is a noun in Dutch. However, some care is needed: although we might translate *zij is van dag jarig* as *it’s her birthday today*, the word *jarig* is in fact an adjective in Dutch, and has no exact equivalent in English.

All languages acquire new lexical items. A list of words recently added to the Oxford Dictionary of English includes *cyberslacker*, *fatoush*, *blamestorm*, *SARS*, *cantopop*, *bupkis*, *noughties*, *muggle*, and *robata*. Notice that all these new words are nouns, and this is reflected in calling nouns an *open class*. By contrast, prepositions are regarded as a *closed class*. That is, there is a limited set of words belonging to the class (e.g., *above*, *along*, *at*, *below*, *beside*, *between*, *during*, *for*, *from*, *in*, *near*, *on*, *outside*, *over*, *past*, *through*, *towards*, *under*, *up*, *with*), and membership of the set only changes very gradually over time.

6.2 English Word Classes

This section presents a brief overview of English word classes. Readers requiring more detail are encouraged to consult a grammar of English.

Linguists commonly recognize four major categories of open class words in English, namely nouns, verbs, adjectives and adverbs. Nouns generally refer to people, places, things, or concepts, e.g.: *woman*, *Scotland*, *book*, *intelligence*. In the context of a sentence, nouns can appear after determiners and adjectives, and can be the subject or object of the verb:

Syntactic Patterns involving some Nouns

² We use > to mean ‘is derived from’.

Word	After a determiner	Subject of the verb
woman	<i>the</i> woman who I saw yesterday ...	the woman <i>sat</i> down
Scotland	<i>the</i> Scotland I remember as a child ...	Scotland <i>has</i> five million people
book	<i>the</i> book I bought yesterday ...	this book <i>recounts</i> the colonisation of Australia
intelligence	<i>the</i> intelligence displayed by the child ...	Mary's intelligence <i>impressed</i> her teachers

English nouns can be morphologically complex. For example, words like **books** and **women** are plural. As we saw earlier, words with the **-ness** suffix are nouns that have been derived from adjectives, e.g. **happiness** and **illness**. The **-ment** suffix appears on certain nouns derived from verbs, e.g. **government** and **establishment**.

Nouns are usually further classified as *common nouns* and *proper nouns*. Proper nouns identify particular individuals or entities, e.g. **Moses** and **Scotland**, while common nouns are all the rest. Another important distinction exists between *count nouns* and *mass nouns*. Count nouns are thought of as distinct entities which can be counted, such as **pig** (e.g. **one pig**, **two pigs**, **many pigs**). They cannot occur with the word **much** (i.e. ***much pigs**). Mass nouns, on the other hand, are not thought of as distinct entities (e.g. **sand**). They cannot be pluralised, and do not occur with numbers (e.g. ***two sands**, ***many sands**). However, they can occur with **much** (i.e. **much sand**).

Verbs are words which describe events and actions, e.g. **fall**, **eat**. In the context of a sentence, verbs express a relation involving the referents of one or more noun phrases.

Syntactic Patterns involving some Verbs		
Word	Simple	With modifiers and adjuncts (italicised)
fall	Rome fell	Dot com stocks <i>suddenly</i> fell <i>like a stone</i>
eat	Mice eat cheese	John ate the pizza <i>with gusto</i>

Verbs can be classified according to the number of arguments (usually noun phrases) that they co-occur with. The word **fall** is *intransitive*, requiring exactly one argument (the entity which falls). The word **eat** is *transitive*, requiring two arguments (the eater and the eaten). Other verbs are more complex; for instance **put** requires three arguments, the agent doing the putting, the entity being put somewhere, and a location. The **-ing** suffix appears on nouns derived from verbs, e.g. **the falling of the leaves** (this is known as the *gerund*).

English verbs can be morphologically complex. For instance, the *present participle* of a verb ends in **-ing**, and expresses the idea of ongoing, incomplete action (e.g. **falling**, **eating**). The *past participle* of a verb often ends in **-ed**, and expresses the idea of a completed action (e.g. **fell**, **ate**).

Two other important word classes are *adjectives* and *adverbs*. Adjectives describe nouns, and can be used as modifiers (e.g. **large in the large pizza**), or in predicates (e.g. **the pizza is large**). English adjectives can be morphologically complex (e.g. **fall**<subscript>V</subscript>**+ing** in **the falling stocks**). Adverbs modify verbs to specify the time, manner, place or direction of the event described by the verb (e.g. **quickly in the stocks fell quickly**). Adverbs may also modify adjectives (e.g. **really in Mary's teacher was really nice**).

English has several categories of closed class words in addition to prepositions, and each dictionary and grammar classifies them differently. The following table gives a sample of closed class words, following the classification of the Brown Corpus.³

Some English Closed Class Words, with Brown Tag

³ Note that part-of-speech tags may be presented as either upper-case or lower-case strings -- there is no significance attached to this difference.

ap	determiner/pronoun, post-determiner	many other next more last former little several enough most least only very few fewer past same
at	article	the an no a every th' ever' ye
cc	conjunction, coordinating	and or but plus & either neither nor yet 'n' and/or minus an'
cs	conjunction, subordinating	that as after whether before while like because if since for than until so unless though providing once lest till whereas whereupon supposing albeit then
in	preposition	of in for by considering to on among at through with under into regarding than since despite ...
md	modal auxiliary	should may might will would must can could shall ought need wilt
pn	pronoun, nominal	none something everything one anyone nothing nobody everybody everyone anybody anything someone no-one nothin'
ppl	pronoun, singular, reflexive	itself himself myself yourself herself oneself ownself
pp\$	determiner, possessive	our its his their my your her out thy mine thine
pp\$\$	pronoun, possessive	ours mine his hers theirs yours
pps	pronoun, personal, nom, 3rd pers sng	it he she thee
ppss	pronoun, personal, nom, not 3rd pers sng	they we I you ye thou you'uns
wdt	WH-determiner	which what whatever whichever
wps	WH-pronoun, nominative	that who whoever whosoever what whatsoever

6.3 Part-of-Speech Tag Sets

Most part-of-speech tag sets make use of the same basic categories, such as noun, verb, adjective, and preposition. However, tag sets differ both in how finely they divide words into categories; and in how they define their categories. For example, *is* might be tagged as a verb in one tag set; but as a distinct form of *to be* in another tag set -- in fact, we just observed the latter situation in the Brown Corpus tag set. This variation in tag sets is unavoidable, since part-of-speech tags are used in different ways for different tasks. In other words, there is no one 'right way' to assign tags, only more or less useful ways, depending on one's goals.

Observe that the tagging process simultaneously collapses distinctions (i.e., lexical identity is usually lost when all personal pronouns are tagged *prp*), while introducing distinctions and removing ambiguities (e.g. *deal* tagged as *vb* or *nn*). This move facilitates classification and prediction. Observe that when we introduce finer distinctions in a tag set, we get better information about linguistic context, but we have to do more work to classify the current token (there are more tags to choose from). Conversely, with fewer distinctions, we have less work to do for classifying the current token, but less information about the context to draw on.

In this tutorial, we will use the following tags: *at* (article) *nn* (Noun), *vb* (Verb), *jj* (Adjective), *in* (Preposition), *cd* (Number), and *end* (Sentence-ending punctuation). As we mentioned, this is a radically simplified version of the Brown Corpus tag set, which in its entirety has 87 basic tags plus many combinations.

7 Representing Tagged Tokens and Tagged Corpora

The preceding sections have discussed the nature and use of tags in language processing. In this section the computational representation of tags is presented. First we consider individual tagged tokens, and show how they are created and accessed. Then we address tagged corpora.

By convention, a tagged token is represented using a Python tuple:

```
>>> tok = ('fly', 'nn')
>>> tok
('fly', 'nn')
```

We can access the properties of this token in the usual way, as shown below:

```
>>> print tok[0]
fly
>>> print tok[1]
nn
```

Several large corpora (such as the Brown Corpus and portions of the Wall Street Journal) have been manually tagged with part-of-speech tags. Before we can use these corpora, we must read them from files and tokenize them.

Tagged texts are usually stored in files as sequences of whitespace-separated tokens, where each token is of the form `text/tag`, as illustrated below for a sample from the Brown Corpus:

```
The/at grand/jj jury/nn commented/vbd on/in a/at number/nn of/in
other/ap topics/nns ,/, among/in them/ppo the/at Atlanta/np and/cc
Fulton/np-tl County/nn-tl purchasing/vbg departments/nns which/wdt it/pps
said/vbd ‘‘/‘‘ are/ber well/ql operated/vbn and/cc follow/vb generally/rb
accepted/vbn practices/nns which/wdt inure/vb to/in the/at best/jjt
interest/nn of/in both/abx governments/nns ‘’’’ ./.
```

It is possible to use the `nltk_lite.corpora` module to read and tokenize data from a tagged corpus, as we saw above.

Here is another example which constructs tokens from a string:

```
>>> sent = """
... John/nn saw/vb the/at book/nn on/in the/at table/nn ./end He/nn sighed/vb ./end
... """
>>> from nltk_lite.tag import tag2tuple
>>> for t in tokenize.whitespace(sent):
...     print tag2tuple(t),
('John', 'nn') ('saw', 'vb') ('the', 'at') ('book', 'nn') ('on', 'in') ('the', 'at') ('table',
```

8 More Applications

Now that we can access tagged text, it is possible to do a variety of useful processing tasks. Here we consider just two: guessing the part-of-speech tag of a word, and exploring the frequency distribution of modal verbs according to text genre.

8.1 Classifying Words Automatically

A tagged corpus can be used to *train* a simple classifier, which can then be used to guess the tag for untagged words. For each word, we can count the number of times it is tagged with each tag. For instance, the word `deal` is tagged 89 times as `nn` and 41 times as `vb`. On this evidence, if we were asked

`brown.groups()`). The material for each genre lives in a set of files (accessed using `brown.items`). Each of these is tokenized in turn. The next step is to check if the token has the `md` tag. For each of these words we increment a count. This uses the conditional frequency distribution, where the condition is the current genre, and the event is the modal.

```
>>> cfdist = ConditionalFreqDist()
>>> for genre in brown.items:           # each genre
...     for sent in brown.tagged(genre): # each sentence
...         for (word,tag) in sent:     # each tagged token
...             if tag == 'md':        # found a modal
...                 cfdist[genre].inc(word.lower())
```

The conditional frequency distribution is nothing more than a mapping from each genre to the distribution of modals in that genre. The following code fragment identifies a small set of modals of interest, and processes the data structure to output the required counts.

```
>>> modals = ['can', 'could', 'may', 'might', 'must', 'will']
>>> print "%-40s" % 'Genre', ' '.join(["%6s" % m for m in modals])
Genre                               can  could   may  might   must   will
>>> for genre in cfdist.conditions(): # generate rows
...     print "%-40s" % brown.item_name[genre],
...     for modal in modals:
...         print "%6d" % cfdist[genre].count(modal),
...     print
press: reportage                    94    86    66    36    50    387
press: reviews                      44    40    45    26    18    56
press: editorial                    122   56    74    37    53    225
skill and hobbies                   273   59   130    22    83    259
religion                            84    59    79    12    54    64
belles-lettres                      249  216  213   113   169   222
popular lore                         168  142  165    45    95   163
miscellaneous: government & house organs 115   37   152    13    99   237
fiction: general                     39   168    8    42    55    50
learned                             366  159  325   126   202   330
fiction: science                     16    49    4    12    8    16
fiction: mystery                     44   145   13    57   31   17
fiction: adventure                    48   154    6    58   27   48
fiction: romance                      79   195   11    51   46   43
humor                                17    33    8     8    9    13
```

There are some interesting patterns in this table. For instance, compare the rows for government literature and adventure literature; the former is dominated by the use of `can`, `may`, `must`, `will` while the latter is characterised by the use of `could` and `might`. With some further work it might be possible to guess the genre of a new text automatically, according to its distribution of modals.

Now that we have seen how tagged tokens and tagged corpora are created and accessed, we are ready to take a look at the automatic categorization of words.

9 Further Reading

John Hopkins Center for Language and Speech Processing, 1999 Summer Workshop on Normalization of Non-Standard Words: Final Report <http://www.clsp.jhu.edu/ws99/projects/normal/report.pdf>

SIL Glossary of Linguistic Terms: <http://www.sil.org/linguistics/GlossaryOfLinguisticTerms/>

Language Files: Materials for an Introduction to Language and Linguistics (Eighth Edition), The Ohio State University Department of Linguistics, <http://www.ling.ohio-state.edu/publications/files/>

10 Exercises

1. Accessing and tokenizing a text file: Obtain some plain text data (e.g. visit a web-page and save it as plain text), and store it in a file 'corpus.txt'.
 - a. Using the `open()` and `read()` functions, load the text into a string variable and print it.
 - b. Now, initialize a new token with `Token()`, using this text. Tokenize the text with `WhitespaceTokenizer`, and specify that the result should be stored in the `WORDS` property. Print the result.
 - c. Next, compute the number of tokens, using the `len()` function, and print the result.
 - d. Finally, discuss shortcomings of this method for tokenizing text. In particular, identify any material which has not been correctly tokenized. (You may need to look for a more complex text.)
2. Tokenizing text using regular expressions: Obtain some plain text data (e.g. visit a web-page and save it as plain text), and store it in a file 'corpus.txt'. so that you can answer the following questions.
 - a. Word processors typically hyphenate words when they are split across a linebreak. When word-processed documents are converted to plain text, the pieces are usually not recombined. It is easy to discover such texts by searching on the web for broken words, e.g. `depart- ment`. Create a `RegexTokenizer` which treats such broken words as a single token.
 - b. Consider the following book title: *This Is the Beat Generation: New York-San Francisco-Paris*. What would it take to be able to tokenize such strings so that each city name was stored as a single token?
3. Concordancing: Write a function which takes a word as its argument, and which searches the Brown Corpus for instances of this word. For each instance, generate a line of text with the target word in the middle.
4. Zipf's Law: Let $f(w)$ be the frequency of a word w in free text. Suppose that all the words of a text are ranked according to their frequency, with the most frequent word first. Zipf's law states that the frequency of a word type is inversely proportional to its rank (i.e. $f \cdot r = k$, for some constant k). For example, the 50th most common word type should occur three times as frequently as the 150th most common word type.
 - a. Write a Python function to process a large text and plot word frequency against word rank using the `nlk_lite.draw.plot` module. Do you confirm Zipf's law? (Hint: it helps to set the axes to log-log.) What is going on at the extreme ends of the plotted line?
 - b. Generate random text, e.g. using `random.choice("abcdefg ")`, taking care to include the space character. Use the string concatenation operator to accumulate characters into a (very) long string. Then tokenize this string, and generate the Zipf plot as before, and compare the two plots. What do you make of Zipf's Law in the light of this?
5. Working with tagged text: Write a program which loads the Brown corpus, then, given a word, lists the possible tags for the word each with a frequency count. For example, for the word `strike` the program would generate: `[('nn', 25), ('vb', 21)]`. (Hint: this task involves sorting and reversing a list of tuples which has the form `[(21, 'vb'), (25, 'nn')]`. To convert such lists into the required form, use `word_freq = [(y,x) for (x,y) in freq_word]`.)

- a. Use your program to print the tags and their frequencies for the following words: **can, fox, get, lift, like, but, frank, line, interest**. Check that you know the meaning of the high-frequency tags.
 - b. Write a program to find the 20 words which have the greatest variety of different possible tags.
 - c. Pick words which can be either a noun or a verb (e.g. **deal**). Guess which is the most likely tag for each word, then check whether you were right.
6. Predicting the next word: The word prediction program we saw in this chapter quickly gets stuck in a cycle. Modify the program to choose the next word randomly, from a list of the n most likely words in the given context. (Hint: store the n most likely words in a list `lwords` then randomly choose a word from the list using `random.choice()`.)
- a. Select a particular genre, such as a section of the Brown Corpus, or a genesis translation, or one of the newsgroups corpora. Train your system on this corpus and get it to generate random text. You may have to experiment with different start words. How intelligible is the text? Examine the strengths and weaknesses of this method of generating random text.
 - b. Try the same approach with different genres, and with different amounts of training data. What do you observe?
 - c. Now train your system using two distinct genres and experiment with generating text in the hybrid genre. As before, discuss your observations.
7. Classifying words automatically: The program for classifying words as nouns or adjectives scored 71%. We will try to come up with better conditions, to get the system to score 80% or better.
- a. Revise the condition to use a longer suffix of the word, such as the last two characters, or the last three characters. What happens to the performance? Which suffixes are diagnostic for adjectives?
 - b. Explore other conditions, such as variable length prefixes of a word, or the length of a word, or the number of vowels in a word.
 - c. Finally, combine multiple conditions into a tuple, and explore which combination of conditions gives the best result.
8. Write a program to implement one or more text readability scores (<http://en.wikipedia.org/wiki/Readability>).
9. Design an algorithm to find the statistically improbable phrases of a document collection. <http://www.amazon.com/gp/search-inside/sipshelp.html/>

NLTK