Chapter 4

Categorizing and Tagging Words

4.1 Introduction

In Chapter 3 we dealt with words in their own right. We saw that some distinctions can be collapsed using normalization, but we did not make any further generalizations. We looked at the distribution of *often*, identifying the words that follow it; we noticed that *often* frequently modifies verbs. We also assumed that you knew that words such as *was*, *called* and *appears* are all verbs, and that you knew that *often* is an adverb. In fact, we take it for granted that most people have a rough idea about how to group words into different categories.

There is a long tradition of classifying words into categories called **parts of speech**. These are sometimes also called word classes or **lexical categories**. Apart from verb and adverb, other familiar examples are **noun**, **preposition**, and **adjective**. One of the notable features of the Brown corpus is that all the words have been **tagged** for their part-of-speech. Now, instead of just looking at the words that immediately follow *often*, we can look at the **part-of-speech tags** (or **POS tags**). Table 4.1 lists the top eight, ordered by frequency, along with explanations of each tag. As we can see, the majority of words following *often* are verbs.

Tag	Freq	Example	Comment
vbn	61	*burnt*, *gone*	verb: past participle
vb	51	*make*, *achieve*	verb: base form
vbd	36	*saw*, *looked*	verb: simple past tense
jj	30	*ambiguous*, *acceptable*	adjective
vbz	24	*sees*, *goes*	verb: third-person singular present
in	18	*by*, *in*	preposition
at	18	*a*, *this*	article
,	16	,	comma

Table 4.1: Part of Speech Tags Following *often* in the Brown Corpus

The process of classifying words into their parts-of-speech and labeling them accordingly is known as **part-of-speech tagging**, **POS-tagging**, or simply **tagging**. The collection of tags used for a particular task is known as a **tag set**. Our emphasis in this chapter is on exploiting tags, and tagging text automatically.

Automatic tagging can bring a number of benefits. We have already seen an example of how to exploit tags in corpus analysis — we get a clear understanding of the distribution of *often* by looking at the tags of adjacent words. Automatic tagging also helps predict the behavior of previously unseen words. For example, if we encounter the word *blogging* we can probably infer that it is a verb, with the root *blog*, and likely to occur after forms of the auxiliary `to be` (e.g. *he was blogging*). Parts of speech are also used in speech synthesis and recognition. For example, `wind/nn`, as in *the wind blew*, is pronounced with a short vowel, whereas `wind/vb`, as in *wind the clock*, is pronounced with a long vowel. Other examples can be found where the stress pattern differs depending on whether the word is a noun or a verb, e.g. *contest*, *insult*, *present*, *protest*, *rebel*, *suspect*. Without knowing the part of speech we cannot be sure of pronouncing the word correctly.

In the next section we will see how to access and explore the Brown Corpus. Following this we will take a more in depth look at the linguistics of word classes. The rest of the chapter will deal with automatic tagging: simple taggers, evaluation, n-gram taggers, and the Brill tagger.

4.2 Getting Started with Tagging

Several large corpora, such as the Brown Corpus and portions of the Wall Street Journal, have been tagged for part-of-speech, and we will be able to process this tagged data. Tagged corpus files typically contain text of the following form (this example is 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 ''/'' ./.
```

4.2.1 Representing Tags and Reading Tagged Corpora

By convention in NLTK, a tagged token is represented using a Python tuple. A tuple is just like a list, only it cannot be modified. We can access the components of a tuple using indexing:

```
>>> tok = ('fly', 'nn')
>>> tok
('fly', 'nn')
>>> tok[0]
'fly'
>>> tok[1]
'nn'
```

We can create one of these special tuples from the standard string representation of a tagged token, using the function `tag2tuple()`:

```
>>> from nltk_lite.tag import tag2tuple
>>> tag2tuple('fly/nn')
('fly', 'nn')
```

We can construct tagged tokens directly from a string. The first step is to tokenize the string (using `tokenize.whitespace()`) to access the individual `word/tag` strings, and then to convert each of these into a tuple (using `tag2tuple()`). We do this in two ways. The first method, starting at line

①, initializes an empty list `tagged_words`, loops over the `word/tag` tokens, converts them into tuples, appends them to `tagged_words`, and finally displays the result. The second method, on line ②, uses a list comprehension to do the same work in a way that is not only more compact, but also more readable. (List comprehensions were introduced in section 3.3.3).

```
>>> from nltk_lite import tokenize
>>> sent = '''
... 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 ''/'' ./.
... '''
>>> tagged_words = []                                          ①
>>> for t in tokenize.whitespace(sent):
...     tagged_words.append(tag2tuple(t))
>>> tagged_words
[('The', 'at'), ('grand', 'jj'), ('jury', 'nn'), ('commented', 'vbd'),
('on', 'in'), ('a', 'at'), ('number', 'nn'), ... ('.', '.')]
>>> [tag2tuple(t) for t in tokenize.whitespace(sent)]    ②
[('The', 'at'), ('grand', 'jj'), ('jury', 'nn'), ('commented', 'vbd'),
('on', 'in'), ('a', 'at'), ('number', 'nn'), ... ('.', '.')]
```

We can also conveniently access tagged corpora directly from Python. The first step is to load the Brown Corpus reader, `brown`. We then use one of its functions, `brown.tagged()` to produce a sequence of sentences, where each sentence is a list of tagged words.

```
>>> from nltk_lite.corpora import brown, extract
>>> extract(6, brown.tagged('a'))
[('The', 'at'), ('grand', 'jj'), ('jury', 'nn'), ('commented', 'vbd'),
('on', 'in'), ('a', 'at'), ('number', 'nn'), ('of', 'in'), ('other', 'ap'),
('topics', 'nns'), (',', ','), ... ('.', '.')]
```

Tagged corpora are available for several other languages. For example, we can access tagged corpora for some Indian languages with `nltk_lite.corpora.indian`. Figure 4.1 shows the output of the demonstration code `indian.demo()`. The tags used with these corpora are documented in the `README` file that comes with this data.

```
Bangla: কুঁড়েঘরগুলি/'NN' আকার/'NN' বাংলার/'NNP' বা/'CC' ভারতের/'NNP' ?/None
নয়/'JJ' ?/None এঁচলের/'NN' প্রচলিত/'JJ' কুঁড়ে/'NN' ঘর/'NN' নয়/'VM' ৷/'SYM'
Hindi: पाकिस्तान/'NNP' की/'PREP' पूर्व/'JJ' प्रधानमंत्री/'NN' बेनजीर/'NNPC' भुट्टो/'NNP'
पर/'PREP' लगे/'VFM' भ्रष्टाचार/'NN' के/'PREP' आरोपों/'NN' के/'PREP' खिलाफ/'PREP' भुट्टो/'NNP'
द्वारा/'PREP' दायर/'NVB' की/'VFM' गई/'VAUX' याचिका/'NN' की/'PREP' सुनवाई/'NN'
मंगलवार/'NN' को/'PREP' वकीलों/'NN' की/'PREP' हड़ताल/'NN' के/'PREP' कारण/'PREP'
स्थगित/'JVB' कर/'VFM' दी/'VAUX' गई/'VAUX' ।/'PUNC'
Marathi: ग्रामीण/'JJ' जिल्हाध्यक्ष/'NN' बाळासाहेब/'NNPC' भोसले/'NNP' यांच्या/'PRP' ?/None
दयक्षतेखाली/'NN' पक्षाची/'NN' आज/'NN' बै?/None क/'NN' झाली/'VM' ./'SYM'
Telugu: ఖజురుల/'NN' నుంచి/'PREP' వచ్చిన/'VJJ' పళ్లాల/'NN' ను/'PREP' సొఖ్ఖా/'NN'
```

Figure 4.1: POS-Tagged Data from Four Indian Languages

Note

Contributions of tagged data for other languages are invited. These will be incorporated into NLTK.

4.2.2 Nouns and Verbs

Linguists recognize several major categories of words in English, such as nouns, verbs, adjectives and determiners. In this section we will discuss the most important categories, namely nouns and verbs.

Nouns generally refer to people, places, things, or concepts, e.g.: *woman, Scotland, book, intelligence*. Nouns can appear after determiners and adjectives, and can be the subject or object of the verb, as shown in Table 4.2.

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

Table 4.2: Syntactic Patterns involving some Nouns

Nouns can be classified as **common nouns** and **proper nouns**. Proper nouns identify particular individuals or entities, e.g. *Moses* and *Scotland*. Common nouns are all the rest. Another 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 pluralized, 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 Table 4.3. In the context of a sentence, verbs express a relation involving the referents of one or more noun phrases.

Word	Simple	With modifiers and adjuncts (italicized)
fall	Rome fell	Dot com stocks *suddenly* fell *like a stone*
eat	Mice eat cheese	John ate the pizza *with gusto*

Table 4.3: Syntactic Patterns involving some Verbs

Verbs can be classified according to the number of arguments (usually noun phrases) that they require. 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. We will return to this topic when we come to look at grammars and parsing (see Chapter 7).

In the Brown Corpus, verbs have a range of possible tags, e.g.: `give/vb` (present), `gives/vbz` (present, 3ps), `giving/vbg` (present continuous; gerund) `gave/vbd` (simple past), and `given/vbn` (past participle). We will discuss these tags in more detail in a later section.

4.2.3 Nouns and verbs in tagged corpora

Now that we are able to access tagged corpora, we can write simple simple programs to garner statistics about the tags. In this section we will focus on the nouns and verbs.

What are the 10 most common verbs? We can write a program to find all words tagged with VB, VBZ, VBG, VBD or VBN.

```
>>> from nltk_lite.probability import FreqDist
>>> fd = FreqDist()
>>> for sent in brown.tagged():
...     for word, tag in sent:
...         if tag[:2] == 'vb':
...             fd.inc(word+"/"+tag)
>>> fd.sorted_samples()[:20]
['said/vbd', 'make/vb', 'see/vb', 'get/vb', 'know/vb', 'made/vbn',
'came/vbd', 'go/vb', 'take/vb', 'went/vbd', 'say/vb', 'used/vbn',
'made/vbd', 'United/vbn-tl', 'think/vb', 'took/vbd', 'come/vb',
'knew/vbd', 'find/vb', 'going/vbg']
```

Let's study nouns, and find the most frequent nouns of each noun part-of-speech type. The program in Listing 4.1 finds all tags starting with `nn`, and provides a few example words for each one. Observe that there are many noun tags; the most important of these have `$` for possessive nouns, `s` for plural nouns (since plural nouns typically end in *s*), `p` for proper nouns.

Some tags contain a plus sign; these are compound tags, and are assigned to words that contain two parts normally treated separately. Some tags contain a minus sign; this indicates disjunction [MORE].

4.2.4 The Default Tagger

The simplest possible tagger assigns the same tag to each token. This may seem to be a rather banal step, but it establishes an important baseline for tagger performance. In order to get the best result, we tag each word with the most likely word. (This kind of tagger is known as a **majority class classifier**). What then, is the most frequent tag? We can find out using a simple program:

```
>>> fd = FreqDist()
>>> for sent in brown.tagged('a'):
...     for word, tag in sent:
...         fd.inc(tag)
>>> fd.max()
'nn'
```

Now we can create a tagger, called `default_tagger`, which tags everything as `nn`.

```
>>> from nltk_lite import tag
>>> tokens = tokenize.whitespace('John saw 3 polar bears .')
>>> default_tagger = tag.Default('nn')
>>> list(default_tagger.tag(tokens))
[('John', 'nn'), ('saw', 'nn'), ('3', 'nn'), ('polar', 'nn'),
('bears', 'nn'), ('.', 'nn')]
```

> **Note**
>
> The tokenizer is a *generator* over tokens. We cannot print it directly, but we can convert it to a list for printing, as shown in the above program. Note that we can only use a generator once, but if we save it as a list, the list can be used many times over.

Listing 1 Program to Find the Most Frequent Noun Tags

```
from nltk_lite.probability import ConditionalFreqDist
def findtags(tag_prefix, tagged_text):
    cfd = ConditionalFreqDist()
    for sent in tagged_text:
        for word, tag in sent:
            if tag.startswith(tag_prefix):
                cfd[tag].inc(word)
    tagdict = {}
    for tag in cfd.conditions():
        tagdict[tag] = cfd[tag].sorted_samples()[:5]
    return tagdict

>>> tagdict = findtags('nn', brown.tagged('a'))
>>> for tag in sorted(tagdict):
...     print tag, tagdict[tag]
nn ['year', 'time', 'state', 'week', 'home']
nn$ ["year's", "world's", "state's", "city's", "company's"]
nn$-hl ["Golf's", "Navy's"]
nn$-tl ["President's", "Administration's", "Army's", "Gallery's", "League's"]
nn-hl ['Question', 'Salary', 'business', 'condition', 'cut']
nn-nc ['aya', 'eva', 'ova']
nn-tl ['President', 'House', 'State', 'University', 'City']
nn-tl-hl ['Fort', 'Basin', 'Beat', 'City', 'Commissioner']
nns ['years', 'members', 'people', 'sales', 'men']
nns$ ["children's", "women's", "janitors'", "men's", "builders'"]
nns$-hl ["Dealers'", "Idols'"]
nns$-tl ["Women's", "States'", "Giants'", "Bombers'", "Braves'"]
nns-hl ['$12,500', '$14', '$37', 'A135', 'Arms']
nns-tl ['States', 'Nations', 'Masters', 'Bears', 'Communists']
nns-tl-hl ['Nations']
```

This is a simple algorithm, and it performs poorly when used on its own. On a typical corpus, it will tag only about an eighth of the tokens correctly:

```
>>> tag.accuracy(default_tagger, brown.tagged('a'))
0.13089484257215028
```

Default taggers assign their tag to every single word, even words that have never been encountered before. As it happens, most new words are nouns. Thus, default taggers they help to improve the robustness of a language processing system. We will return to them later, in the context of our discussion of *backoff*.

4.2.5 Exercises

1. ☼ Working with someone else, take turns to pick a word which can be either a noun or a verb (e.g. *contest*); the opponent has to predict which one is likely to be the most frequent in the Brown corpus; check the opponents prediction, and tally the score over several turns.

2. ◑ Write programs to process the Brown Corpus and find answers to the following questions:

    1) Which nouns are more common in their plural form, rather than their singular form? (Only consider regular plurals, formed with the *-s* suffix.)
    2) Which word has the greatest number of distinct tags. What are they, and what do they represent?
    3) List tags in order of decreasing frequency. What do the 20 most frequent tags represent?
    4) Which tags are nouns most commonly found after? What do these tags represent?

3. ◑ Generate some statistics for tagged data to answer the following questions:

    a) What proportion of word types are always assigned the same part-of-speech tag?
    b) How many words are ambiguous, in the sense that they appear with at least two tags?
    c) What percentage of word *occurrences* in the Brown Corpus involve these ambiguous words?

4. ◑ Above we gave an example of the `tag.accuracy()` function. It has two arguments, a tagger and some tagged text, and it works out how accurately the tagger performs on this text. For example, if the supplied tagged text was `[('the', 'dt'), ('dog', 'nn')]` and the tagger produced the output `[('the', 'nn'), ('dog', 'nn')]`, then the accuracy score would be `0.5`. Can you figure out how the `tag.accuracy()` function works?

    a) A tagger takes a list of words as input, and produces a list of tagged words as output. However, `tag.accuracy()` is given correctly tagged text as its input. What must the `tag.accuracy()` function do with this input before performing the tagging?

b) Once the supplied tagger has created newly tagged text, how would `tag.accuracy()` go about comparing it with the original tagged text and computing the accuracy score?

4.3 Looking for Patterns in Words

4.3.1 Some morphology

English nouns can be morphologically complex. For example, words like *books* and *women* are plural. 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*.

English verbs can also 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 *-ing* suffix also appears on nouns derived from verbs, e.g. *the falling of the leaves* (this is known as the **gerund**). In the Brown corpus, these are tagged `vbg`.

The **past participle** of a verb often ends in *-ed*, and expresses the idea of a completed action (e.g. *fell, ate*). These are tagged `vbd`.

[MORE: Modal verbs, e.g. *would* ...]

Common tag sets often capture some **morpho-syntactic** information; that is, information about the kind of morphological markings which words receive by virtue of their syntactic role. Consider, for example, the selection of distinct grammatical forms of the word *go* illustrated in the following sentences:

(1a) *Go* away!

(1b) He sometimes *goes* to the cafe.

(1c) All the cakes have *gone*.

(1d) We *went* on the excursion.

Each of these forms — *go*, *goes*, *gone*, and *went* — is morphologically distinct from the others. Consider the form, *goes*. This cannot occur in all grammatical contexts, but requires, for instance, a third person singular subject. Thus, the following sentences are ungrammatical.

(2a) *They sometimes *goes* to the cafe.

(2b) *I sometimes *goes* to the cafe.

By contrast, *gone* is the past participle form; it is required after *have* (and cannot be replaced in this context by *goes*), and cannot occur as the main verb of a clause.

(3a) *All the cakes have *goes*.

(3b) *He sometimes *gone* to the cafe.

We can easily imagine a tag set in which the four distinct grammatical forms just discussed were all tagged as VB. Although this would be adequate for some purposes, a more fine-grained tag set will provide useful information about these forms that can be of value to other processors which try to detect syntactic patterns from tag sequences. As we noted at the beginning of this chapter, the Brown tag set does in fact capture these distinctions, as summarized in Table 4.4.

Form	Category	Tag
go	base	vb
goes	3rd singular present	vbz
gone	past participle	vbn
went	simple past	vbd

Table 4.4: Some morphosyntactic distinctions in the Brown tag set

These differences between the forms are encoded in their Brown Corpus tags: `be/be`, `being/beg`, `am/bem`, `been/ben` and `was/bedz`. This means that an automatic tagger which uses this tag set is in effect carrying out a limited amount of morphological analysis.

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 just tagged as a verb in one tag set; but as a distinct form of the lexeme *BE* in another tag set (as in the Brown Corpus). 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. More details about the Brown corpus tag set can be found in the Appendix.

4.3.2 The Regular Expression Tagger

The regular expression tagger assigns tags to tokens on the basis of matching patterns. For instance, we might guess that any word ending in *ed* is the past participle of a verb, and any word ending with *'s* is a possessive noun. We can express these as a list of regular expressions:

```
>>> patterns = [
...     (r'.*ing$', 'vbg'),               # gerunds
...     (r'.*ed$', 'vbd'),                # simple past
...     (r'.*es$', 'vbz'),                # 3rd singular present
...     (r'.*ould$', 'md'),               # modals
...     (r'.*\'s$', 'nn$'),               # possessive nouns
...     (r'.*s$', 'nns'),                 # plural nouns
...     (r'^-?[0-9]+(.[0-9]+)?$', 'cd'),  # cardinal numbers
...     (r'.*', 'nn')                     # nouns (default)
... ]
```

Note that these are processed in order, and the first one that matches is applied.

Now we can set up a tagger and use it to tag some text.

```
>>> regexp_tagger = tag.Regexp(patterns)
>>> list(regexp_tagger.tag(brown.raw('a')))[3]
[('``', 'nn'), ('Only', 'nn'), ('a', 'nn'), ('relative', 'nn'),
('handful', 'nn'), ('of', 'nn'), ('such', 'nn'), ('reports', 'nns'),
('was', 'nns'), ('received', 'vbd'), ("''", 'nn'), (',', 'nn'),
('the', 'nn'), ('jury', 'nn'), ('said', 'nn'), (',', 'nn'), ('``', 'nn'),
('considering', 'vbg'), ('the', 'nn'), ('widespread', 'nn'), ..., ('.', 'nn')]
```

How well does this do?

```
>>> tag.accuracy(regexp_tagger, brown.tagged('a'))
0.20326391789486245
```

The regular expression is a catch-all, which tags everything as a noun. This is equivalent to the default tagger (only much less efficient). Instead of re-specifying this as part of the regular expression tagger, is there a way to combine this tagger with the default tagger? We will see how to do this later, under the heading of backoff taggers.

4.3.3 Exercises

1. ☼ Search the web for "spoof newspaper headlines", to find such gems as: *British Left Waffles on Falkland Islands*, and *Juvenile Court to Try Shooting Defendant*. Manually tag these headlines to see if knowledge of the part-of-speech tags removes the ambiguity.

2. ☼ Satisfy yourself that there are restrictions on the distribution of *go* and *went*, in the sense that they cannot be freely interchanged in the kinds of contexts illustrated in (1).

3. ◑ Write code to search for particular words and phrases according to tags, to answer the following questions:

   a) Produce an alphabetically sorted list of the distinct words tagged as `md`.

   b) Identify words which can be plural nouns or third person singular verbs (e.g. *deals*, *flies*).

   c) Identify three-word prepositional phrases of the form IN + DET + NN (eg. *in the lab*).

   d) What is the ratio of masculine to feminine pronouns?

4. ◑ In the introduction we saw a table involving frequency counts for the adjectives *adore*, *love*, *like*, *prefer* and preceding qualifiers such as *really*. Investigate the full range of qualifiers (Brown tag `ql`) which appear before these four adjectives.

5. ◑ We defined the `regexp_tagger`, which can be used as a fall-back tagger for unknown words. This tagger only checks for cardinal numbers. By testing for particular prefix or suffix strings, it should be possible to guess other tags. For example, we could tag any word that ends with *-s* as a plural noun. Define a regular expression tagger (using `tag.Regexp` which tests for at least five other patterns in the spelling of words. (Use inline documentation to explain the rules.)

6. ◑ Consider the regular expression tagger developed in the exercises in the previous section. Evaluate the tagger using `tag.accuracy()`, and try to come up with ways to improve its performance. Discuss your findings. How does objective evaluation help in the development process?

7. ★ There are 264 distinct words in the Brown Corpus having exactly three possible tags.

   a) Print a table with the integers 1..10 in one column, and the number of distinct words in the corpus having 1..10 distinct tags.

   b) For the word with the greatest number of distinct tags, print out sentences from the corpus containing the word, one for each possible tag.

8. ★ Write a program to classify contexts involving the word *must* according to the tag of the following word. Can this be used to discriminate between the epistemic and deontic uses of *must*?

4.4 Baselines and Backoff

So far the performance of our simple taggers has been disappointing. Before we embark on a process to get 90+% performance, we need to do two more things. First, we need to establish a more principled baseline performance than the default tagger, which was too simplistic, and the regular expression tagger, which was too arbitrary. Second, we need a way to connect multiple taggers together, so that if a more specialized tagger is unable to assign a tag, we can "back off" to a more generalized tagger.

4.4.1 The Lookup Tagger

A lot of high-frequency words do not have the `nn` tag. Let's find some of these words and their tags. The function in Listing 4.2 takes a list of sentences and counts up the words, and returns the *n* most frequent words. We'll test out this function for the 100 most frequent words:

Listing 2 Program to Find the N Most Frequent Words

```
def wordcounts(sents, n):
    "find the n most frequent words"
    fd = FreqDist()
    for sent in sents:
        for word in sent:
            fd.inc(word)    # count the word
    return fd.sorted_samples()[:n]

>>> frequent_words = wordcounts(brown.raw('a'), 100)
>>> frequent_words
['the', ',', '.', 'of', 'and', 'to', 'a', 'in', 'for', 'The',
'that', '``', 'is', 'was', "''", 'on', 'at', 'with', 'be', 'by',
'as', 'he', 'said', 'his', 'will', 'it', 'from', 'are', ';', '--',
'an', 'has', 'had', 'who', 'have', 'not', 'Mrs.', 'were', 'this',
'which', 'would', 'their', 'been', 'they', 'He', 'one', ..., 'now']
```

Next, let's inspect the tags that these words have. First we will do this in the most obvious (but highly inefficient) way:

```
>>> [(w,t) for sent in brown.tagged('a')
...        for (w,t) in sent
...        if w in frequent_words]
[('The', 'at'), ('said', 'vbd'), ('an', 'at'), ('of', 'in'),
('``', '``'), ('no', 'at'), ("''", "''"), ('that', 'cs'),
('any', 'dti'), ('.', '.'), ..., ("''", "''")]
```

A much better approach is to set up a dictionary which maps each of the 100 most frequent words to its most likely tag. We can do this by setting up a frequency distribution `cfd` over the tags, conditioned on each of the frequent words, as shown in Listing 4.3. This gives us, for each word, a count of the frequency of different tags that occur with the word.

Now for any word that appears in this section of the corpus, we can look up its most likely tag. For example, to find the tag for the word *The* we can access the corresponding frequency distribution, and ask for its most frequent event:

Listing 3 Program to Find the Most Likely Tags for the Specified Words

```
from nltk_lite.probability import ConditionalFreqDist
def wordtags(tagged_sents, words):
    "Find the most likely tag for these words in the tagged sentences"
    cfd = ConditionalFreqDist()
    for sent in tagged_sents:
        for (w,t) in sent:
            if w in words:
                cfd[w].inc(t)    # count the word's tag
    return dict((word, cfd[word].max()) for word in words)
```

```
>>> table = wordtags(brown.tagged('a'), frequent_words)
>>> table['The']
'at'
```

Now we can create and evaluate a simple tagger that assigns tags to words based on this table:

```
>>> baseline_tagger = tag.Lookup(table)
>>> tag.accuracy(baseline_tagger, brown.tagged('a'))
0.45578495136941344
```

This is surprisingly good; just knowing the tags for the 100 most frequent words enables us to tag nearly half the words correctly! Let's see how it does on some untagged input text:

```
>>> list(baseline_tagger.tag(brown.raw('a')))[3]
[('``', '``'), ('Only', None), ('a', 'at'), ('relative', None),
('handful', None), ('of', 'in'), ('such', None), ('reports', None),
('was', 'bedz'), ('received', None), ("''", "''"), (',', ','),
('the', 'at'), ('jury', None), ('said', 'vbd'), (',', ','),
('``', '``'), ('considering', None), ('the', 'at'), ('widespread', None),
('interest', None), ('in', 'in'), ('the', 'at'), ('election', None),
(',', ','), ('the', 'at'), ('number', None), ('of', 'in'),
('voters', None), ('and', 'cc'), ('the', 'at'), ('size', None),
('of', 'in'), ('this', 'dt'), ('city', None), ("''", "''"), ('.', '.')]
```

Notice that a lot of these words have been assigned a tag of `None`. That is because they were not among the 100 most frequent words. In these cases we would like to assign the default tag of `nn`, a process known as backoff.

4.4.2 Backoff

How do we combine these taggers? We want to use the lookup table first, and if it is unable to assign a tag, then use the default tagger. We do this by specifying the default tagger as an argument to the lookup tagger. The lookup tagger will call the default tagger just in case it can't assign a tag itself.

```
>>> baseline_tagger = tag.Lookup(table, backoff=tag.Default('nn'))
>>> tag.accuracy(baseline_tagger, brown.tagged('a'))
0.58177695566561249
```

4.4.3 Choosing a good baseline

We can write a simple (but somewhat inefficient) program to create and evaluate lookup taggers having a range of sizes, as shown in Listing 4.4. We include a backoff tagger that tags everything as a noun. A consequence of using this backoff tagger is that the lookup tagger only has to store word/tag pairs for words other than nouns.

Listing 4 Lookup Tagger Performace with Varying Model Size

```
def performance(size):
    frequent_words = wordcounts(brown.raw('a'), size)
    table = wordtags(brown.tagged('a'), frequent_words)
    baseline_tagger = tag.Lookup(table, backoff=tag.Default('nn'))
    return tag.accuracy(baseline_tagger, brown.tagged('a'))

>>> from pylab import *
>>> sizes = 2**arange(15)
>>> perfs = [performance(size) for size in sizes]
>>> plot(sizes, perfs, '-bo')
>>> title('Lookup Tagger Performance with Varying Model Size')
>>> xlabel('Model Size')
>>> ylabel('Performance')
>>> show()
```

Observe that performance initially increases rapidly as the model size grows, eventually reaching a plateau, when large increases in model size yield little improvement in performance. (This example used the `pylab` plotting package; we will return to this later in Section 6.3.4).

4.4.4 Exercises

1. ◑ Explore the following issues that arise in connection with the lookup tagger:

   a) What happens to the tagger performance for the various model sizes when a backoff tagger is omitted?

   b) Consider the curve in Figure 4.2; suggest a good size for a lookup tagger that balances memory and performance. Can you come up with scenarios where it would be preferable to minimize memory usage, or to maximize performance with no regard for memory usage?

2. ◑ What is the upper limit of performance for a lookup tagger, assuming no limit to the size of its table? (Hint: write a program to work out what percentage of tokens of a word are assigned the most likely tag for that word, on average.)

4.5 Getting Better Coverage

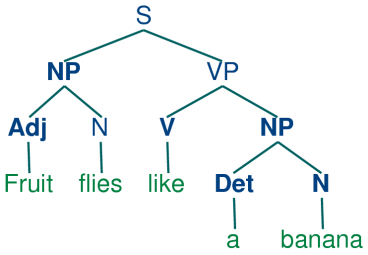
4.5.1 More English Word Classes

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*$_V$*+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, such as **articles** (also often called **determiners**) (e.g., *the*, *a*), **modals** (e.g., *should*, *may*), and **personal pronouns** (e.g., *she*, *they*). Each dictionary and grammar classifies these words differently.

Part-of-speech tags are closely related to the notion of word class used in syntax. The assumption in linguistics is that every distinct word type will be listed in a lexicon (or dictionary), with information about its pronunciation, syntactic properties and meaning. A key component of the word's properties will be its class. When we carry out a syntactic analysis of an example like *fruit flies like a banana*, we will look up each word in the lexicon, determine its word class, and then group it into a hierarchy of phrases, as illustrated in the following parse tree.



Syntactic analysis will be dealt with in more detail in Part II. For now, we simply want to make the connection between the labels used in syntactic parse trees and part-of-speech tags. Table 4.5 shows the correspondence:

Word Class Label	Brown Tag	Word Class
Det	AT	article
N	NN	noun
V	VB	verb
Adj	JJ	adjective
P	IN	preposition
Card	CD	cardinal number
--	.	Sentence-ending punctuation

Table 4.5: Word Class Labels and Brown Corpus Tags

4.5.2 Some diagnostics

Now that we have examined word classes in detail, we turn to a more basic question: how do we decide what category a word belongs to in the first place? In general, linguists use three criteria: morphological (or formal); syntactic (or distributional); semantic (or notional). A **morphological** 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*. So if we encounter a word which ends in *-ness*, this is very likely to be a noun.

A **syntactic** criterion refers to the 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 categorized as an adjective:

(4a) the near window

(4b) The end is (very) near.

A familiar example of a **semantic** criterion is that a noun is "the name of a person, place or thing". Within modern linguistics, semantic criteria for word classes are treated with suspicion, mainly because they are hard to formalize. Nevertheless, semantic criteria underpin many of our intuitions about word classes, and enable us to make a good guess about the categorization of words in languages that we are unfamiliar with. For example, if we 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 vandaag 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.

With this background we are now ready to embark on our main task for this chapter, automatically assigning part-of-speech tags to words.

4.5.3 Unigram Tagging

The `tag.Unigram()` class implements a simple statistical tagging algorithm: for each token, it assigns the tag that is most likely for that particular token. For example, it will assign the tag `jj` to any occurrence of the word *frequent*, since *frequent* is used as an adjective (e.g. *a frequent word*) more often than it is used as a verb (e.g. *I frequent this cafe*).

Before a unigram tagger can be used to tag data, it must be trained on a tagged corpus. It uses this corpus to determine which tags are most common for each word. Unigram taggers are trained using the `train()` method, which takes a tagged corpus:

```
>>> from nltk_lite.corpora import brown
>>> from itertools import islice
>>> train_sents = list(islice(brown.tagged(), 500))  # sents 0..499
>>> unigram_tagger = tag.Unigram()
>>> unigram_tagger.train(train_sents)
```

Once a unigram tagger has been trained, the `tag()` method can be used to tag new text:

```
>>> text = "John saw the book on the table"
>>> tokens = list(tokenize.whitespace(text))
>>> list(unigram_tagger.tag(tokens))
[('John', 'np'), ('saw', 'vbd'), ('the', 'at'), ('book', None),
('on', 'in'), ('the', 'at'), ('table', None)]
```

The unigram tagger will assign the special tag `None` to any token that was not encountered in the training data.

4.5.4 Affix Taggers

Affix taggers are like unigram taggers, except they are trained on word prefixes or suffixes of a specified length. (NB. Here we use *prefix* and *suffix* in the string sense, not the morphological sense.) For example, the following tagger will consider suffixes of length 3 (e.g. *-ize*, *-ion*), for words having at least 5 characters.

```
>>> affix_tagger = tag.Affix(-2, 3)
>>> affix_tagger.train(train_sents)
>>> list(affix_tagger.tag(tokens))
[('John', 'np'), ('saw', 'nn'), ('the', 'at'), ('book', 'np'),
('on', None), ('the', 'at'), ('table', 'jj')]
```

4.5.5 Exercises

1. ☼ Train a unigram tagger and run it on some new text. Observe that some words are not assigned a tag. Why not?

2. ☼ Train an affix tagger `tag.Affix()` and run it on some new text. Experiment with different settings for the affix length and the minimum word length. Can you find a setting which seems to perform better than the one described above? Discuss your findings.

3. ◑ Write a program which calls `tag.Affix()` repeatedly, using different settings for the affix length and the minimum word length. What parameter values give the best overall performance? Why do you think this is the case?

4.6 N-Gram Taggers

Earlier we encountered the unigram tagger, which assigns a tag to a word based on the identity of that word. In this section we will look at taggers that exploit a larger amount of context when assigning a tag.

4.6.1 Bigram Taggers

Bigram taggers use two pieces of contextual information for each tagging decision, typically the current word together with the tag of the previous word. Given the context, the tagger assigns the most likely tag. In order to do this, the tagger uses a bigram table, a fragment of which is shown in Table 4.6. Given the tag of the previous word (down the left), and the current word (across the top), it can look up the preferred tag.

tag	ask	Congress	to	increase	grants	to	states
at				nn			
tl			to			to	
bd			to		nns	to	
md	*vb*			vb			
vb		*np*	to		*nns*	to	nns
np			*to*			to	
to	vb		*vb*				
nn		np	to	nn	nns	to	

tag	ask	Congress	to	increase	grants	to	states
nns			to			*to*	
in		np	in			in	*nns*
jj			to		nns	to	nns

Table 4.6: Fragment of Bigram Table

The best way to understand the table is to work through an example. Suppose we are processing the sentence *The President will ask Congress to increase grants to states for vocational rehabilitation* . and that we have got as far as `will/md`. We can use the table to simply read off the tags that should be assigned to the remainder of the sentence. When preceded by `md`, the tagger guesses that *ask* has the tag `vb` (italicized in the table). Moving to the next word, we know it is preceded by `vb`, and looking across this row we see that *Congress* is assigned the tag `np`. The process continues through the rest of the sentence. When we encounter the word *increase*, we correctly assign it the tag `vb` (unlike the unigram tagger which assigned it `nn`). However, the bigram tagger mistakenly assigns the infinitival tag to the word *to* immediately preceding *states*, and not the preposition tag. This suggests that we may need to consider even more context in order to get the correct tag.

4.6.2 N-Gram Taggers

As we have just seen, it may be desirable to look at more than just the preceding word's tag when making a tagging decision. An **n-gram tagger** is a generalization of a bigram tagger whose context is the current word together with the part-of-speech tags of the *n-1*: preceding tokens, as shown in the following diagram. It then picks the tag which is most likely for that context. The tag to be chosen, t_n, is circled, and the context is shaded in grey. In the example of an n-gram tagger shown in Figure 4.3, we have *n=3*; that is, we consider the tags of the two preceding words in addition to the current word.

> **Note**
>
> A 1-gram tagger is another term for a unigram tagger: i.e., the context used to tag a token is just the text of the token itself. 2-gram taggers are also called *bigram taggers*, and 3-gram taggers are called *trigram taggers*.

The `tag.Ngram` class uses a tagged training corpus to determine which part-of-speech tag is most likely for each context. Here we see a special case of an n-gram tagger, namely a bigram tagger:

```
>>> bigram_tagger = tag.Bigram()
>>> bigram_tagger.train(brown.tagged(['a','b']))
```

Once a bigram tagger has been trained, it can be used to tag untagged corpora:

```
>>> text = "John saw the book on the table"
>>> tokens = list(tokenize.whitespace(text))
>>> list(bigram_tagger.tag(tokens))
[('John', None), ('saw', None), ('the', 'at'), ('book', 'nn'),
('on', 'in'), ('the', 'at'), ('table', None)]
```

As with the other taggers, n-gram taggers assign the tag NONE to any token whose context was not seen during training.

As *n* gets larger, the specificity of the contexts increases, as does the chance that the data we wish to tag contains contexts that were not present in the training data. This is known as the *sparse data*

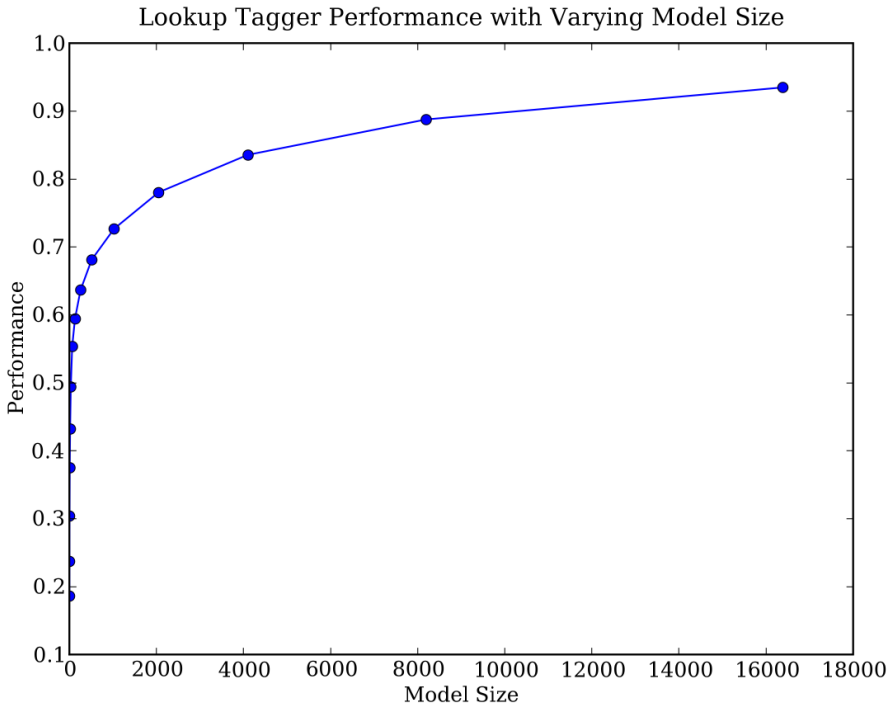


Figure 4.2: Lookup Tagger

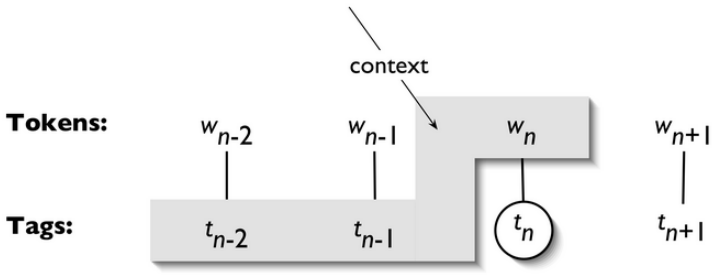


Figure 4.3: Tagger Context

problem, and is quite pervasive in NLP. Thus, there is a trade-off between the accuracy and the coverage of our results (and this is related to the **precision/recall trade-off** in information retrieval.)

> **Note**
>
> n-gram taggers should not consider context that crosses a sentence boundary. Accordingly, NLTK taggers are designed to work with lists of sentences, where each sentence is a list of words. At the start of a sentence, t_{n-1} and preceding tags are set to `None`.

4.6.3 Combining Taggers

One way to address the trade-off between accuracy and coverage is to use the more accurate algorithms when we can, but to fall back on algorithms with wider coverage when necessary. For example, we could combine the results of a bigram tagger, a unigram tagger, and a `regexp_tagger`, as follows:

1. Try tagging the token with the bigram tagger.
2. If the bigram tagger is unable to find a tag for the token, try the unigram tagger.
3. If the unigram tagger is also unable to find a tag, use a default tagger.

Each NLTK tagger other than `tag.Default` permits a backoff-tagger to be specified. The backoff-tagger may itself have a backoff tagger:

```
>>> t0 = tag.Default('nn')
>>> t1 = tag.Unigram(backoff=t0)
>>> t2 = tag.Bigram(backoff=t1)
>>> t1.train(brown.tagged('a'))    # section a: press-reportage
>>> t2.train(brown.tagged('a'))
```

> **Note**
>
> We specify the backoff tagger when the tagger is initialized, so that training can take advantage of the backing off. Thus, if the bigram tagger would assign the same tag as its unigram backoff tagger in a certain context, the bigram tagger discards the training instance. This keeps the bigram tagger model as small as possible. We can further specify that a tagger needs to see more than one instance of a context in order to retain it, e.g. `Bigram(cutoff=2, backoff=t1)` will discard contexts which have only been seen once or twice.

As before we test the taggers against unseen data. Here we will use a different segment of the corpus.

```
>>> accuracy0 = tag.accuracy(t0, brown.tagged('b')) # section b: press-editorial
>>> accuracy1 = tag.accuracy(t1, brown.tagged('b'))
>>> accuracy2 = tag.accuracy(t2, brown.tagged('b'))

>>> print 'Default Accuracy = %4.1f%%' % (100 * accuracy0)
Default Accuracy = 12.5%
>>> print 'Unigram Accuracy  = %4.1f%%' % (100 * accuracy1)
Unigram Accuracy  = 81.0%
>>> print 'Bigram Accuracy = %4.1f%%' % (100 * accuracy2)
Bigram Accuracy = 81.9%
```

4.6.4 Investigating tagger performance

What is the upper limit to tagger performance? Unfortunately perfect tagging is impossible. Consider the case of a trigram tagger. How many cases of part-of-speech ambiguity does it encounter? We can determine the answer to this question empirically:

```
>>> from nltk_lite.corpora import brown
>>> from nltk_lite.probability import ConditionalFreqDist
>>> cfdist = ConditionalFreqDist()
>>> for sent in brown.tagged('a'):
...     p = [(None, None)] # empty token/tag pair
...     trigrams = zip(p+p+sent, p+sent+p, sent+p+p)
...     for (pair1,pair2,pair3) in trigrams:
...         context = (pair1[1], pair2[1], pair3[0]) # last 2 tags, this word
...         cfdist[context].inc(pair3[1])            # current tag
>>> total = ambiguous = 0
>>> for cond in cfdist.conditions():
...     if cfdist[cond].B() > 1:
...         ambiguous += cfdist[cond].N()
...     total += cfdist[cond].N()
>>> print float(ambiguous) / total
0.0560190160471
```

Thus, one out of twenty trigrams is ambiguous. Given the current word and the previous two tags, there is more than one tag that could be legitimately assigned to the current word according to the training data. Assuming we always pick the most likely tag in such ambiguous contexts, we can derive an empirical upper bound on the performance of a trigram tagger.

Another way to investigate the performance of a tagger is to study its mistakes. Some tags may be harder than others to assign, and it might be possible to treat them specially by pre- or post-processing the data. A convenient way to look at tagging errors is the **confusion matrix**. It charts expected tags (the gold standard) against actual tags generated by a tagger:

```
>>> from nltk_lite.evaluate import ConfusionMatrix
>>> from nltk_lite.tag import untag
>>> def tag_list(tagged_sents):
...     return [tag for (word, tag) in sent for sent in tagged_sents]
>>> def apply_tagger(tagger, corpus):
...     return [tagger.tag(untag(sent)) for sent in corpus]
>>> gold = tag_list(brown.tagged('b'))
>>> test = tag_list(apply_tagger(t2, brown.tagged('b')))
>>> print ConfusionMatrix(gold, test)
```

Based on such analysis we may decide to modify the tagset. Perhaps a distinction between tags that is difficult to make can be dropped, since it is not important in the context of some larger processing task. We could collapse distinctions with the aid of a dictionary that maps the existing tagset to a simpler tagset:

```
>>> mapping = {'pps': 'nn', 'ppss': 'nn'}
>>> data = [(word, mapping.get(tag, tag))
            for (word, tag) in sent for sent in tagged_sents]
```

We can do this for both the training and test data.

4.6.5 Storing Taggers

Training a tagger on a large corpus may take several minutes. Instead of training a tagger every time we need one, it is convenient to save a trained tagger in a file for later re-use. Let's save our tagger `t2` to a file `t2.pkl`.

```
>>> from cPickle import dump
>>> output = open('t2.pkl', 'wb')
>>> dump(t2, output, -1)
>>> output.close()
```

Now, in a separate Python process, we can load our saved tagger.

```
>>> from cPickle import load
>>> input = open('t2.pkl', 'rb')
>>> tagger = load(input)
>>> input.close()
```

Now let's check that it can be used for tagging.

```
>>> text = """The board's action shows what free enterprise
...     is up against in our complex maze of regulatory laws ."""
>>> tokens = list(tokenize.whitespace(text))
>>> list(tagger.tag(tokens))
[('The', 'at'), ("board's", 'nn$'), ('action', 'nn'), ('shows', 'nns'),
('what', 'wdt'), ('free', 'jj'), ('enterprise', 'nn'), ('is', 'bez'),
('up', 'rp'), ('against', 'in'), ('in', 'in'), ('our', 'pp$'), ('complex', 'jj'),
('maze', 'nn'), ('of', 'in'), ('regulatory', 'nn'), ('laws', 'nns'), ('.', '.')]
```

4.6.6 Smoothing

[Brief discussion of NLTK's smoothing classes, for another approach to handling unknown words: Lidstone, Laplace, Expected Likelihood, Heldout, Witten-Bell, Good-Turing.]

4.6.7 Exercises

1. ☼ Train a bigram tagger with no backoff tagger, and run it on some of the training data. Next, run it on some new data. What happens to the performance of the tagger? Why?

2. ◑ Inspect the confusion matrix for the bigram tagger `t2` defined above, and identify one or more sets of tags to collapse. Define a dictionary to do the mapping, and evaluate the tagger on the simplified data.

3. ◑ How serious is the sparse data problem? Investigate the performance of n-gram taggers as n increases from 1 to 6. Tabulate the accuracy score. Estimate the training data required for these taggers, assuming a vocabulary size of 10^5 and a tagset size of 10^2.

4. ★ Create a default tagger and various unigram and n-gram taggers, incorporating backoff, and train them on part of the Brown corpus.

   a) Create three different combinations of the taggers. Test the accuracy of each combined tagger. Which combination works best?

   b) Try varying the size of the training corpus. How does it affect your results?

5. ★ Our approach for tagging an unknown word has been to consider the letters of the word (using `tag.Regexp()` and `tag.Affix()`), or to ignore the word altogether and tag it as a noun (using `tag.Default()`). These methods will not do well for texts having new words that are not nouns. Consider the sentence *I like to blog on Kim's blog*. If `blog`:lx: is a new word, then looking at the previous tag (`to` vs `np$`) would probably be helpful. I.e. we need a default tagger that is sensitive to the preceding tag.

   a) Create a new kind of unigram tagger that looks at the tag of the previous word, and ignores the current word. (The best way to do this is to modify the source code for `tag.Unigram()`, which presumes knowledge of Python classes discussed in Section 10.1.)

   b) Add this tagger to the sequence of backoff taggers (including ordinary trigram and bigram taggers that look at words), right before the usual default tagger.

   c) Evaluate the contribution of this new unigram tagger.

4.7 Conclusion

This chapter has introduced the language processing task known as tagging, with an emphasis on part-of-speech tagging. English word classes and their corresponding tags were introduced. We showed how tagged tokens and tagged corpora can be represented, then discussed a variety of taggers: default tagger, regular expression tagger, unigram tagger and n-gram taggers. We also described some objective evaluation methods. In the process, the reader has been introduced to an important paradigm in language processing, namely *language modeling*. This paradigm former is extremely general, and we will encounter it again later.

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. 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.

There are several other important approaches to tagging involving *Transformation-Based Learning*, *Markov Modeling*, and *Finite State Methods*. We will discuss these in a later chapter. In Chapter 5 we will see a generalization of tagging called *chunking* in which a contiguous sequence of words is assigned a single tag.

Part-of-speech tagging is just one kind of tagging, one that does not depend on deep linguistic analysis. There are many other kinds of tagging. Words can be tagged with directives to a speech synthesizer, indicating which words should be emphasized. Words can be tagged with sense numbers, indicating which sense of the word was used. Words can also be tagged with morphological features. Examples of each of these kinds of tags are shown below. For space reasons, we only show the tag for a single word. Note also that the first two examples use XML-style tags, where elements in angle brackets enclose the word that is tagged.

1. *Speech Synthesis Markup Language (W3C SSML):* `That is a <emphasis>big</emphasis> car!`

2. *SemCor: Brown Corpus tagged with WordNet senses:* `Space in any <wf pos="NN " lemma="form" wnsn="4">form</wf> is completely measured by the three dimensions.` (Wordnet form/nn sense 4: "shape, form, configuration, contour, conformation")

3. *Morphological tagging, from the Turin University Italian Treebank:* `E' italiano , come progetto e realizzazione , il primo (PRIMO ADJ ORDIN M SING ) porto turistico dell' Albania .`

Tagging exhibits several properties that are characteristic of natural language processing. First, tagging involves *classification*: words have properties; many words share the same property (e.g. `cat` and `dog` are both nouns), while some words can have multiple such properties (e.g. `wind` is a noun and a verb). Second, in tagging, disambiguation occurs via *representation*: we augment the representation of tokens with part-of-speech tags. Third, training a tagger involves *sequence learning from annotated corpora*. Finally, tagging uses *simple, general, methods* such as conditional frequency distributions and transformation-based learning.

We have seen that ambiguity in the training data leads to an upper limit in tagger performance. Sometimes more context will resolve the ambiguity. In other cases however, as noted by Abney (1996), the ambiguity can only resolved with reference to syntax, or to world knowledge. Despite these imperfections, part-of-speech tagging has played a central role in the rise of statistical approaches to natural language processing. In the early 1990s, the surprising accuracy of statistical taggers was a striking demonstration that it was possible to solve one small part of the language understanding problem, namely part-of-speech disambiguation, without reference to deeper sources of linguistic knowledge. Can this idea be pushed further? In Chapter 5, on chunk parsing, we shall see that it can.

4.8 Further Reading

Tagging: Jurafsky and Martin, Chapter 8

Brill tagging: Manning and Schutze 361ff; Jurafsky and Martin 307ff

HMM tagging: Manning and Schutze 345ff

Abney, Steven (1996). Tagging and Partial Parsing. In: Ken Church, Steve Young, and Gerrit Bloothooft (eds.), *Corpus-Based Methods in Language and Speech.* Kluwer Academic Publishers, Dordrecht. `http://www.vinartus.net/spa/95a.pdf`

Wikipedia: `http://en.wikipedia.org/wiki/Part-of-speech_tagging`

List of available taggers: `http://www-nlp.stanford.edu/links/statnlp.html`

4.9 Further Exercises

1. **Impossibility of exact tagging:** Write a program to determine the upper bound for accuracy of an n-gram tagger. Hint: how often is the context seen during training inadequate for uniquely determining the tag to assign to a word?

2. **Impossibility of exact tagging:** Consult the Abney reading and review his discussion of the impossibility of exact tagging. Explain why correct tagging of these examples requires access to other kinds of information than just words and tags. How might you estimate the scale of this problem?

3. **Application to other languages**: Obtain some tagged data for another language, and train and evaluate a variety of taggers on it. If the language is morphologically complex, or if there are any orthographic clues (e.g. capitalization) to word classes, consider developing a regular expression tagger for it (ordered after the unigram tagger, and before the default tagger). How does the accuracy of your tagger(s) compare with the same taggers run on English data? Discuss any issues you encounter in applying these methods to the language.

4. **Comparing n-gram taggers and Brill taggers** (advanced): Investigate the relative performance of n-gram taggers with backoff and Brill taggers as the size of the training data is increased. Consider the training time, running time, memory usage, and accuracy, for a range of different parameterizations of each technique.

5. **HMM taggers**: Explore the Hidden Markov Model tagger `nltk_lite.tag.hmm`.

6. (Advanced) **Estimation:** Use some of the estimation techniques in `nltk_lite.probability`, such as *Lidstone* or *Laplace* estimation, to develop a statistical tagger that does a better job than ngram backoff taggers in cases where contexts encountered during testing were not seen during training. Read up on the TnT tagger, since this provides useful technical background: `http://www.aclweb.org/anthology/A00-1031`

4.10 Appendix: Brown Tag Set

Table 4.7 gives a sample of closed class words, following the classification of the Brown Corpus. (Note that part-of-speech tags may be presented as either upper-case or lower-case strings -- the case difference is not significant.)

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

Table 4.7: Some English Closed Class Words, with Brown Tag

4.10.1 Acknowledgments