

4. Categorizing and Tagging Words

In the last chapter 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 abstractions over groups of words. We also 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**). Here's a list of the top eight, ordered by frequency, along with explanations of each tag. As we can see, the majority of words following *often* are verbs.

Table 1: Part of Speech Tags Following often in the Brown Corpus

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

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 developing tools to tag text automatically.

Automatic tagging can bring a number of benefits. It helps predict the behavior of a previously unseen word. 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*.

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.

4.1 Exploring a Tagged Corpus using Python

4.1.1 Representing Tags and Reading Tagged Corpora

By convention in NLTK, a tagged token is represented using a Python tuple as follows:

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

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

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

Several large corpora, such as the Brown Corpus and portions of the Wall Street Journal, have already been tagged, 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 ''/'' ./.
```

We can construct tagged tokens directly from a string, with the help of two NLTK functions, `tokenize.whitespace()` and `tag2tuple`:

```
>>> from nltk_lite import tokenize
>>> from nltk_lite.tag import tag2tuple
>>> 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 ''/'' ./.
... '''
>>> for t in tokenize.whitespace(sent):
...     print tag2tuple(t),
('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'), ('.', '.'), ... ('.', '.')]

```

4.1.2 Brown Corpus Tags and other 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 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.

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.

So far, we have only looked at tags as capturing information about word class. However, common tag sets often capture a certain amount of **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:

3. *Go* away!
4. He sometimes *goes* to the cafe.
5. All the cakes have *gone*.
6. We *went* on the excursion.

It is apparent that each of these forms is morphologically distinct from the others. What do we mean by saying that the morphological markings are correlated with syntactic role? 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.

7. *They sometimes *goes* to the cafe.
8. *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.

9. *All the cakes have *goes*.
10. *He sometimes *gone* to the cafe.

You should be able to satisfy yourself that there are also restrictions on the distribution of *go* and *went* in the sense that they cannot be freely interchanged in the kinds of contexts illustrated by (3)-(6).

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 in fact noted at the beginning of this chapter, the Brown tag set does in fact capture these distinctions, as summarized here:

Table 2: Some morpho-syntactic distinctions in the Brown tag set

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

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.

In the rest of this chapter, we will use the following tags: **at** (article) **nn** (Noun), **vb** (Verb), **jj** (Adjective), **in** (Preposition), **cd** (Number), and **.** (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. More details can be found in the [Appendix](#).

4.1.3 Exercises

1. **Ambiguity resolved by part-of-speech tags:** 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. **Explorations with part-of-speech tagged corpora:** Tokenize the Brown Corpus and build one or more suitable data structures so that you can answer the following questions.
 - a) What is the most frequent tag? (This is the tag we would want to assign with `tag.Default`.)
 - b) Which word has the greatest number of distinct tags?
 - c) What proportion of word types are always assigned the same part-of-speech tag?
 - d) What is the ratio of masculine to feminine pronouns?
 - e) How many words are ambiguous, in the sense that they appear with at least two tags?
 - f) What percentage of word *occurrences* in the Brown Corpus involve these ambiguous words?
 - g) Which nouns are more common in their plural form, rather than their singular form? (Only consider regular plurals, formed with the *-s* suffix.)

- h) Produce an alphabetically sorted list of the distinct words tagged as **md**.
 - i) Identify words which can be plural nouns or third person singular verbs (e.g. *deals, flies*).
 - j) Identify three-word prepositional phrases of the form IN + DET + NN (eg. *in the lab*).
 - k) There are 264 distinct words having exactly three possible tags. 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.
 - l) For the word with the greatest number of distinct tags, print out sentences from the corpus containing the word, one for each possible tag.
3. **Competition:** 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; check the opponents prediction, and tally the score over several turns.
 4. 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*?
 5. In the introduction we saw a table involving frequency counts for the adjectives *adore*, *love:lx*, *like*, *prefer* and preceding qualifiers such as *really*. Investigate the full range of qualifiers (Brown tag **q1**) which appear before these four adjectives.

4.2 English Word Classes

Linguists recognize four major categories of open class words in English: nouns, verbs, adjectives and adverbs. 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:

Table 3: Syntactic Patterns involving some Nouns

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

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*.

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 the context of a sentence, verbs express a relation involving the referents of one or more noun phrases.

Table 4: Syntactic Patterns involving some Verbs

Word	Simple	With modifiers and adjuncts (<i>italicized</i>)
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>

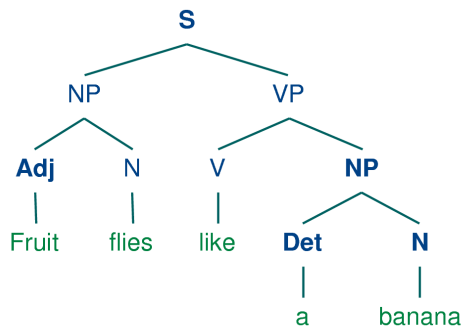
Verbs can be classified according to the number of arguments (usually noun phrases) that they require. The word *fall* is **intransitive**, requiring exactly one argument (the entity which falls). The word *eat* is **transitive**, requiring two arguments (the eater and the eaten). Other verbs are more complex; for instance *put* requires three arguments, the agent doing the putting, the entity being put somewhere, and a location. 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*_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. The following table shows the correspondence:

Table 5: Word Class Labels and Brown Corpus Tags

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

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:

1. the near window
2. 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 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.

4.3 Simple Taggers

In this section we consider three simple taggers. They all process the input tokens one by one, adding a tag to each token. In each case they begin with tokenized text. We can easily create a sample of tokenized text as follows:

```
>>> from nltk_lite import tokenize
>>> text = "John saw 3 polar bears ."
>>> tokens = list(tokenize.whitespace(text))
>>> print tokens
['John', 'saw', '3', 'polar', 'bears', '.']
```

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.

4.3.1 The Default Tagger

The simplest possible tagger assigns the same tag to each token. Here we create a tagger called `my_tagger` which tags everything as a noun.

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

This is a simple algorithm, and it performs poorly when used on its own. On a typical corpus, it will tag only 10%-20% of the tokens correctly.

Default taggers assign their tag to every single word, even words that have never been encountered before. Thus, 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*.

4.3.2 The Regular Expression Tagger

The regular expression tagger assigns tags to tokens on the basis of matching patterns in the token's text. For instance, the following tagger assigns `cd` to cardinal numbers, and `nn` to everything else:

```
>>> patterns = [(r'^-?[0-9]+(.[0-9]+)?$', 'cd'), (r'.*', 'nn')]
>>> nn_cd_tagger = tag.Regexp(patterns)
>>> list(nn_cd_tagger.tag(tokens))
[('John', 'nn'), ('saw', 'nn'), ('3', 'cd'), ('polar', 'nn'),
 ('bears', 'nn'), ('.', 'nn')]
```

We can generalize this method to guess the correct tag for words based on the presence of certain prefix or suffix strings. For instance, English words beginning with *un-* are likely to be adjectives, and words ending with *'s* are likely to be possessive nouns. Here is a more sophisticated regular expression tagger:

```
>>> patterns = [
...     (r'^-?[0-9]+(.[0-9]+)?$', 'cd'), # cardinal numbers
...     (r'(The|the|A|a|An|an|)$', 'at'), # articles
...     (r'un.*', 'jj'), # adjectives
...     (r'.*\s$', 'nn$'), # possessive nouns
...     (r'.*s$', 'nns'), # plural nouns
...     (r'.*ing$', 'vbg'), # gerunds
...     (r'.*ed$', 'vbd'), # past tense verbs
...     (r'.*', 'nn') # nouns (default)
... ]
>>> regexp_tagger = tag.Regexp(patterns)
```

4.3.3 The Unigram Tagger

The `UnigramTagger` class implements a simple statistical tagging algorithm: for each token, it assigns the tag that is most likely for that token's text. 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 `UnigramTagger` can be used to tag data, it must be trained on a *training corpus*. It uses this corpus to determine which tags are most common for each word. `UnigramTaggers` 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 `UnigramTagger` 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)]
```

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

4.3.4 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', 'vbd'),
 ('on', None), ('the', 'at'), ('table', 'jj')]
```

4.3.5 Exercises

1. **Regular Expression Tagging:** We defined the `nn_cd_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.)
2. **Unigram Tagging:** Train a unigram tagger and run it on some new text. Observe that some words are not assigned a tag. Why not?
3. **Affix Tagging:** 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.

4.4 Evaluating Taggers

As we experiment with different taggers, it is important to have an objective performance measure. Fortunately, we already have manually verified training data (the original tagged corpus), so we can use that to score the accuracy of a tagger, and to perform systematic error analysis.

4.4.1 Scoring Accuracy

Consider the following sentence from the Brown Corpus. The 'Gold Standard' tags from the corpus are given in the second column, while the tags assigned by a unigram tagger appear in the third column. Two mistakes made by the unigram tagger are italicized.

Table 6: Evaluating Taggers

Sentence	Gold Standard	Unigram Tagger
The	at	at
President	nn-tl	<i>nn-tl</i>

Table 6: Evaluating Taggers

Sentence	Gold Standard	Unigram Tagger
said	vbd	vbd
he	pps	pps
will	md	md
ask	vb	vb
Congress	np	np
to	to	to
increase	vb	<i>nn</i>
grants	nns	nns
to	in	<i>to</i>
states	nns	nns
for	in	in
vocational	jj	jj
rehabilitation	nn	nn
.	.	.

The tagger correctly tagged 14 out of 16 words, so it gets a score of 14/16, or 87.5%. Of course, accuracy should be judged on the basis of a larger sample of data. NLTK provides a function called `tag.accuracy` to automate the task. In the simplest case, we can test the tagger using the same data it was trained on:

```
>>> acc = tag.accuracy(unigram_tagger, train_sents)
>>> print 'Accuracy = %4.1f%%' % (100 * acc)
Accuracy = 81.8%
```

However, testing a language processing system over its training data is unwise. A system which simply memorized the training data would get a perfect score without doing any linguistic modeling. Instead, we would like to reward systems that make good generalizations, so we should test against *unseen data*, and replace `train_sents` above with `unseen_sents`. We can then define the two sets of data as follows:

```
>>> train_sents = list(brown.tagged('a'))[:500]
>>> unseen_sents = list(brown.tagged('a'))[500:600] # sentcs 500-599
```

Now we train the tagger using `train_sents` and evaluate it using `unseen_sents`, as follows:

```
>>> unigram_tagger = tag.Unigram(backoff=nn_cd_tagger)
>>> unigram_tagger.train(train_sents)
>>> acc = tag.accuracy(unigram_tagger, unseen_sents)
>>> print 'Accuracy = %4.1f%%' % (100 * acc)
Accuracy = 74.7%
```

The accuracy scores produced by this evaluation method are lower, but they give a more realistic picture of the performance of the tagger. Note that the performance of any statistical tagger is highly dependent on the quality of its training set. In particular, if the training set is too small, it will not be

able to reliably estimate the most likely tag for each word. Performance will also suffer if the training set is significantly different from the texts we wish to tag.

In the process of developing a tagger, we can use the accuracy score as an objective measure of the improvements made to the system. Initially, the accuracy score will go up quickly as we fix obvious shortcomings of the tagger. After a while, however, it becomes more difficult and improvements are small.

4.4.2 Baseline Performance

It is difficult to interpret an accuracy score in isolation. For example, is a person who scores 25% in a test likely to know a quarter of the course material? If the test is made up of 4-way multiple choice questions, then this person has not performed any better than chance. Thus, it is clear that we should *interpret* an accuracy score relative to a *baseline*. The choice of baseline is somewhat arbitrary, but it usually corresponds to minimal knowledge about the domain.

In the case of tagging, a possible baseline score can be found by tagging every word with **NN**, the most likely tag.

```
>>> baseline_tagger = tag.Default('nn')
>>> acc = tag.accuracy(baseline_tagger, brown.tagged('a'))
>>> print 'Accuracy = %4.1f%%' % (100 * acc)
Accuracy = 13.1%
```

Unfortunately this is not a very good baseline. There are many high-frequency words which are not nouns. Instead we could use the standard unigram tagger to get a baseline of 75%. However, this does not seem fully legitimate: the unigram's model covers all words seen during training, which hardly seems like 'minimal knowledge'. Instead, let's only permit ourselves to store tags for the most frequent words.

The first step is to identify the most frequent words in the corpus, and for each of these words, identify the most likely tag:

```
>>> from nltk_lite.probability import *
>>> wordcounts = FreqDist()
>>> wordtags = ConditionalFreqDist()
>>> for sent in brown.tagged('a'):
...     for (w,t) in sent:
...         wordcounts.inc(w)      # count the word
...         wordtags[w].inc(t)    # count the word's tag
>>> frequent_words = wordcounts.sorted_samples()[:1000]
```

Now we can create a lookup table (a dictionary) which maps words to likely tags, just for these high-frequency words. We can then define a new baseline tagger which uses this lookup table:

```
>>> table = dict((word, wordtags[word].max()) for word in frequent_words)
>>> baseline_tagger = tag.Lookup(table, tag.Default('nn'))
>>> acc = tag.accuracy(baseline_tagger, brown.tagged('a'))
>>> print 'Accuracy = %4.1f%%' % (100 * acc)
Accuracy = 72.5%
```

This, then, would seem to be a reasonable baseline score for a tagger. When we build new taggers, we will only credit ourselves for performance exceeding this baseline.

Note

`tag.Lookup()` is defined in NLTK-Lite version 0.6.5.

4.4.3 Error Analysis

While the accuracy score is certainly useful, it does not tell us how to improve the tagger. For this we need to undertake error analysis. For instance, we could construct a *confusion matrix*, with a row and a column for every possible tag, and entries that record how often a word with tag T_i is incorrectly tagged as T_j . Another approach is to analyze the context of the errors, which is what we do now.

Consider the following program, which catalogs all errors along with the tag on the left and their frequency of occurrence.

```
>>> errors = {}
>>> for i in range(len(unseen_sents)):
...     raw_sent = tag.untag(unseen_sents[i])
...     test_sent = list(unigram_tagger.tag(raw_sent))
...     unseen_sent = unseen_sents[i]
...     for j in range(len(test_sent)):
...         if test_sent[j][1] != unseen_sent[j][1]:
...             test_context = test_sent[j-1:j+1]
...             gold_context = unseen_sent[j-1:j+1]
...             if None not in test_context:
...                 pair = (tuple(test_context), tuple(gold_context))
...                 errors[pair] = errors.get(pair, 0) + 1
```

The `errors` dictionary has keys of the form `((t1,t2), (g1,g2))`, where `(t1,t2)` are the test tags, and `(g1,g2)` are the gold-standard tags. The values in the `errors` dictionary are simple counts of how often the error occurred. With some further processing, we construct the list `counted_errors` containing tuples consisting of counts and errors, and then do a reverse sort to get the most significant errors first:

```
>>> counted_errors = [(errors[k], k) for k in errors.keys()]
>>> counted_errors.sort()
>>> counted_errors.reverse()
>>> for err in counted_errors[:5]:
...     print err
(32, (((), ()))
(5, (((('the', 'at'), ('Rev.', 'nn')),
        (('the', 'at'), ('Rev.', 'np')))))
(5, (((('Assemblies', 'nn'), ('of', 'in')),
        (('Assemblies', 'nns-tl'), ('of', 'in-tl')))))
(4, (((('of', 'in'), ('God', 'nn')),
        (('of', 'in-tl'), ('God', 'np-tl')))))
(3, (((('to', 'to'), ('form', 'nn')),
        (('to', 'to'), ('form', 'vb')))))
```

The fifth line of output records the fact that there were 3 cases where the unigram tagger mistakenly tagged a verb as a noun, following the word *to*. (We encountered the inverse of this mistake for the word *increase* in the above evaluation table, where the unigram tagger tagged *increase* as a verb instead of a noun since it occurred more often in the training data as a verb.) Here, when *form* appears after the word *to*, it is invariably a verb. Evidently, the performance of the tagger would improve if it was modified to consider not just the word being tagged, but also the tag of the word on the left. Such taggers are known as bigram taggers, and we consider them next.

4.4.4 Exercises

1. **Evaluating a Regular Expression Tagger:** 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?
2. **Evaluating a Unigram Tagger:** Apply our evaluation methodology to the unigram tagger developed in the previous section. Discuss your findings.
3. **Affix Tagging:** 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?

4.5 N-Gram Taggers

Earlier we encountered the **UnigramTagger**, 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.

4.5.1 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. We can visualize this process with the help of the following bigram table, a tiny fragment of the internal data structure built by a bigram tagger.

Table 7: Fragment of Bigram Table

at				nn			
tl			to			to	
bd			to		nns	to	
md	<i>vb</i>			vb			
vb		<i>np</i>	to		<i>nns</i>	to	nns
np			<i>to</i>			to	
to	vb		<i>vb</i>				
nn		np	to	nn	nns	to	
nns			to			<i>to</i>	
in		np	in			in	<i>nns</i>
jj			to		nns	to	nns

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.

4.5.2 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 this example of an n-gram tagger, we have $n=3$; that is, we consider the tags of the two preceding words in addition to the current word.

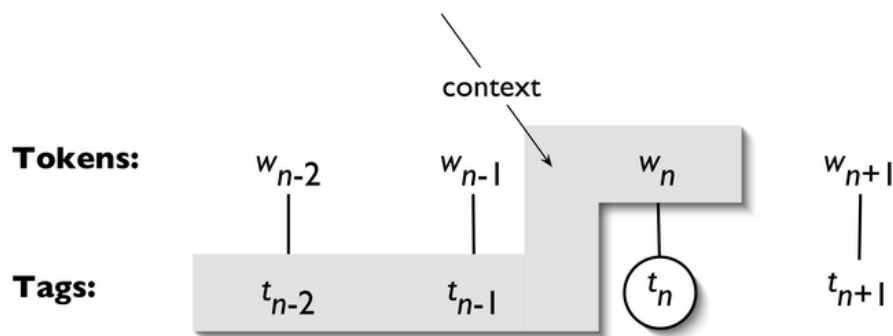


Figure 1: Tagger Context :scale:80

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', 'np'), ('saw', 'vbd'), ('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* 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`.

4.5.3 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 `nn_cd_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 = 80.2%
>>> print 'Bigram Accuracy = %4.1f%%' % (100 * accuracy2)
Bigram Accuracy = 78.3%
```

4.5.4 Exercises

1. **Bigram Tagging:** 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. **Combining taggers:** 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?
3. **Tagger context (advanced):**

N-gram taggers choose a tag for a token based on its text and the tags of the $n-1$ preceding tokens. This is a common context to use for tagging, but certainly not the only possible context.

 - a) Construct a new tagger, sub-classed from `SequentialTagger`, that uses a different context. If your tagger's context contains multiple elements, then you should combine them in a tuple. Some possibilities for elements to include are: (i) the current word or a previous word; (ii) the length of the current word text or of the previous word; (iii) the first letter of the current word or the previous word; or (iv) the previous tag. Try to choose context elements that you believe will help the tagger decide which tag is appropriate. Keep in mind the trade-off between more specific taggers with accurate results; and more general taggers with broader coverage. Combine your tagger with other taggers using the backoff method.
 - b) How does the combined tagger's accuracy compare to the basic tagger?
 - c) How does the combined tagger's accuracy compare to the combined taggers you created in the previous exercise?
4. **Reverse sequential taggers (advanced):** Since sequential taggers tag tokens in order, one at a time, they can only use the predicted tags to the *left* of the current token to decide what tag to assign to a token. But in some cases, the *right* context may provide more useful information than the left context. A reverse sequential tagger starts with the last word of the sentence and, proceeding in right-to-left order, assigns tags to words on the basis of the tags it has already predicted to the right. By reversing texts at appropriate times, we can use NLTK's existing sequential tagging classes to perform reverse sequential tagging: reverse the training text before training the tagger; and reverse the text being tagged both before and after.

- a) Use this technique to create a bigram reverse sequential tagger.
 - b) Measure its accuracy on a tagged section of the Brown corpus. Be sure to use a different section of the corpus for testing than you used for training.
 - c) How does its accuracy compare to a left-to-right bigram tagger, using the same training data and test data?
5. **Alternatives to backoff:** Create a new kind of tagger that combines several taggers using a new mechanism other than backoff (e.g. voting). For robustness in the face of unknown words, include a regexp tagger, a unigram tagger that removes a small number of prefix or suffix characters until it recognizes a word, or an n-gram tagger that does not consider the text of the token being tagged.
6. **Sparse Data Problem:** 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 .

4.6 The Brill Tagger

A potential issue with n-gram taggers is the size of their n-gram table (or language model). If tagging is to be employed in a variety of language technologies deployed on mobile computing devices, it is important to strike a balance between model size and tagger performance. An n-gram tagger with backoff may store trigram and bigram tables, large sparse arrays which may have hundreds of millions of entries.

A second issue concerns context. The only information an n-gram tagger considers from prior context is tags, even though words themselves might be a useful source of information. It is simply impractical for n-gram models to be conditioned on the identities of words in the context. In this section we examine Brill tagging, a statistical tagging method which performs very well using models that are only a tiny fraction of the size of n-gram taggers.

Brill tagging is a kind of *transformation-based learning*. The general idea is very simple: guess the tag of each word, then go back and fix the mistakes. In this way, a Brill tagger successively transforms a bad tagging of a text into a better one. As with n-gram tagging, this is a *supervised learning* method, since we need annotated training data. However, unlike n-gram tagging, it does not count observations but compiles a list of transformational correction rules.

The process of Brill tagging is usually explained by analogy with painting. Suppose we were painting a tree, with all its details of boughs, branches, twigs and leaves, against a uniform sky-blue background. Instead of painting the tree first then trying to paint blue in the gaps, it is simpler to paint the whole canvas blue, then “correct” the tree section by over-painting the blue background. In the same fashion we might paint the trunk a uniform brown before going back to over-paint further details with even finer brushes. Brill tagging uses the same idea: begin with broad brush strokes then fix up the details, with successively finer changes. The following table illustrates this process, first tagging with the unigram tagger, then fixing the errors.

Table 8: Steps in Brill Tagging

Sentence:	Gold:	Uni-gram:	Replace <i>nn</i> with <i>vb</i> when the previous word is <i>to</i>	Replace <i>to</i> with <i>in</i> when the next tag is <i>nns</i>
The	at	at		
President	nn-tl	nn-tl		
said	vbd	vbd		
he	pps	pps		
will	md	md		
ask	vb	vb		
Congress	np	np		
to	to	to		
increase	vb	<i>nn</i>	<i>vb</i>	
grants	nns	nns		
to	in	<i>to</i>	<i>to</i>	<i>in</i>
states	nns	nns		
for	in	in		
vocational	jj	jj		
rehabilitation	nn	nn		

In this table we see two rules. All such rules are generated from a template of the following form: “replace T_1 with T_2 in the context C ”. Typical contexts are the identity or the tag of the preceding or following word, or the appearance of a specific tag within 2-3 words of of the current word. During its training phase, the tagger guesses values for T_1 , T_2 and C , to create thousands of candidate rules. Each rule is scored according to its net benefit: the number of incorrect tags that it corrects, less the number of correct tags it incorrectly modifies. This process is best illustrated by a listing of the output from the NLTK Brill tagger (here run on tagged Wall Street Journal text from the Penn Treebank).

```

Loading tagged data...
Training unigram tagger: [accuracy: 0.820940]
Training Brill tagger on 37168 tokens...

Iteration 1: 1482 errors; ranking 23989 rules;
  Found: "Replace POS with VBZ if the preceding word is tagged PRP"
  Apply: [changed 39 tags: 39 correct; 0 incorrect]

Iteration 2: 1443 errors; ranking 23662 rules;
  Found: "Replace VBP with VB if one of the 3 preceding words is tagged MD"
  Apply: [changed 36 tags: 36 correct; 0 incorrect]

Iteration 3: 1407 errors; ranking 23308 rules;
  Found: "Replace VBP with VB if the preceding word is tagged TO"
  Apply: [changed 24 tags: 23 correct; 1 incorrect]

Iteration 4: 1384 errors; ranking 23057 rules;
  Found: "Replace NN with VB if the preceding word is to"
  Apply: [changed 67 tags: 22 correct; 45 incorrect]

```

```

...
Iteration 20: 1138 errors; ranking 20717 rules;
  Found: "Replace RBR with JJR if one of the 2 following words is tagged NNS"
  Apply: [changed 14 tags: 10 correct; 4 incorrect]

Iteration 21: 1128 errors; ranking 20569 rules;
  Found: "Replace VBD with VBN if the preceding word is tagged VBD"
  [insufficient improvement; stopping]

Brill accuracy: 0.835145

```

Brill taggers have another interesting property: the rules are linguistically interpretable. Compare this with the n-gram taggers, which employ a potentially massive table of n-grams. We cannot learn much from direct inspection of such a table, in comparison to the rules learned by the Brill tagger.

4.6.1 Exercises

1. Try the Brill tagger demonstration, as follows:

```

from nltk_lite.tag import brill
brill.demo()

```

2. Consult the documentation for the demo function, using `help(brill.demo)`. Experiment with the tagger by setting different values for the parameters. Is there any trade-off between training time (corpus size) and performance?
3. (Advanced) Inspect the diagnostic files created by the tagger `rules.out` and `errors.out`. Obtain the demonstration code (`nltk_lite/tag/brill.py`) and create your own version of the Brill tagger.
 - a) Delete some of the rule templates, based on what you learned from inspecting `rules.out`.
 - b) Add some new rule templates which employ contexts that might help to correct the errors you saw in `errors.out`.

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, n-gram taggers, and the Brill tagger. We also described some objective evaluation methods. In the process, the reader has been introduced to two important paradigms in language processing, namely *language modeling* and *transformation-based learning*. The former is extremely general, and we will encounter it again later. The latter had to be specially tailored to the tagging task, but resulted in smaller, linguistically-interpretable models.

There are several other important approaches to tagging involving *Hidden Markov Models* (see `nltk_lite.tag.hmm`) and *Finite State Transducers*, though a discussion of these approaches falls outside the scope of this chapter. Later 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.

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
0.0509036201939
```

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.

Sometimes more context will resolve the ambiguity. In other cases however, as noted by Abney (1996), the ambiguity can only be resolved with reference to syntax, or to world knowledge. Despite these imperfections, part-of-speech tagging has played a crucial 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 the next chapter, on chunk parsing, we shall see that it can.

4.8 Further Reading

Tagging: Jurafsky and Martin, Chapter 8

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

HMM tagging: Manning and Schütze 345ff

Abney, Steven (1996). Tagging and Partial Parsing. In: Ken Church, Steve Young, and Gerrit Bloothoof (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

The following table 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.)

Some English Closed Class Words, with Brown Tag		
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, coordi- nating	and or but plus & either neither nor yet 'n' and/or minus an'
cs	conjunction, subor- dinating	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 every- one anybody anything someone no-one nothin'
ppl	pronoun, singular, reflexive	itself himself myself yourself herself oneself ownself
pp\$	determiner, posses- sive	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, nomi- native	that who whoever whosoever what whatsoever

4.10.1 Acknowledgments

We are grateful to Christopher Maloof for developing NLTK's Brill tagger, and Trevor Cohn for developing NLTK's HMM tagger.

About this document...

This chapter is a draft from *Introduction to Natural Language Processing*, by [Steven Bird](#), [James Curran](#), [Ewan Klein](#) and [Edward Loper](#), Copyright © 2006 the authors. It is distributed with the *Natural Language Toolkit* [<http://nltk.sourceforge.net>], under the terms of the *Creative Commons Attribution-ShareAlike License* [<http://creativecommons.org/licenses/by-sa/2.5/>].