1
Tokenizing Text and WordNet Basics

In this chapter, we will cover the following recipes:

- Tokenizing text into sentences
- Tokenizing sentences into words
- Tokenizing sentences using regular expressions
- Training a sentence tokenizer
- Filtering stopwords in a tokenized sentence
- Looking up Synsets for a word in WordNet
- Looking up lemmas and synonyms in WordNet
- Calculating WordNet Synset similarity
- Discovering word collocations

Introduction

Natural Language ToolKit (**NLTK**) is a comprehensive Python library for natural language processing and text analytics. Originally designed for teaching, it has been adopted in the industry for research and development due to its usefulness and breadth of coverage. NLTK is often used for rapid prototyping of text processing programs and can even be used in production applications. Demos of select NLTK functionality and production-ready APIs are available at `http://text-processing.com`.

This chapter will cover the basics of tokenizing text and using WordNet. **Tokenization** is a method of breaking up a piece of text into many pieces, such as sentences and words, and is an essential first step for recipes in the later chapters. **WordNet** is a dictionary designed for programmatic access by natural language processing systems. It has many different use cases, including:

- Looking up the definition of a word
- Finding synonyms and antonyms
- Exploring word relations and similarity
- Word sense disambiguation for words that have multiple uses and definitions

NLTK includes a WordNet corpus reader, which we will use to access and explore WordNet. A corpus is just a body of text, and corpus readers are designed to make accessing a corpus much easier than direct file access. We'll be using WordNet again in the later chapters, so it's important to familiarize yourself with the basics first.

Tokenizing text into sentences

Tokenization is the process of splitting a string into a list of pieces or tokens. A token is a piece of a whole, so a word is a token in a sentence, and a sentence is a token in a paragraph. We'll start with sentence tokenization, or splitting a paragraph into a list of sentences.

Getting ready

Installation instructions for NLTK are available at `http://nltk.org/install.html` and the latest version at the time of writing this is Version 3.0b1. This version of NLTK is built for Python 3.0 or higher, but it is backwards compatible with Python 2.6 and higher. In this book, we will be using Python 3.3.2. If you've used earlier versions of NLTK (such as version 2.0), note that some of the APIs have changed in Version 3 and are not backwards compatible.

Once you've installed NLTK, you'll also need to install the data following the instructions at `http://nltk.org/data.html`. I recommend installing everything, as we'll be using a number of corpora and pickled objects. The data is installed in a data directory, which on Mac and Linux/Unix is usually `/usr/share/nltk_data`, or on Windows is `C:\nltk_data`. Make sure that `tokenizers/punkt.zip` is in the data directory and has been unpacked so that there's a file at `tokenizers/punkt/PY3/english.pickle`.

Finally, to run the code examples, you'll need to start a Python console. Instructions on how to do so are available at `http://nltk.org/install.html`. For Mac and Linux/Unix users, you can open a terminal and type `python`.

How to do it...

Once NLTK is installed and you have a Python console running, we can start by creating a paragraph of text:

```
>>> para = "Hello World. It's good to see you. Thanks for buying this
book."
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Now we want to split the paragraph into sentences. First we need to import the sentence tokenization function, and then we can call it with the paragraph as an argument:

```
>>> from nltk.tokenize import sent_tokenize
>>> sent_tokenize(para)
['Hello World.', "It's good to see you.", 'Thanks for buying this
book.']
```

So now we have a list of sentences that we can use for further processing.

How it works...

The `sent_tokenize` function uses an instance of `PunktSentenceTokenizer` from the `nltk.tokenize.punkt` module. This instance has already been trained and works well for many European languages. So it knows what punctuation and characters mark the end of a sentence and the beginning of a new sentence.

There's more...

The instance used in `sent_tokenize()` is actually loaded on demand from a pickle file. So if you're going to be tokenizing a lot of sentences, it's more efficient to load the `PunktSentenceTokenizer` class once, and call its `tokenize()` method instead:

```
>>> import nltk.data
>>> tokenizer = nltk.data.load('tokenizers/punkt/PY3/english.pickle')
>>> tokenizer.tokenize(para)
['Hello World.', "It's good to see you.", 'Thanks for buying this
book.']
```

Tokenizing sentences in other languages

If you want to tokenize sentences in languages other than English, you can load one of the other pickle files in `tokenizers/punkt/PY3` and use it just like the English sentence tokenizer. Here's an example for Spanish:

```
>>> spanish_tokenizer = nltk.data.load('tokenizers/punkt/PY3/spanish.
pickle')
>>> spanish_tokenizer.tokenize('Hola amigo. Estoy bien.')
['Hola amigo.', 'Estoy bien.']
```

You can see a list of all the available language tokenizers in `/usr/share/nltk_data/tokenizers/punkt/PY3` (or `C:\nltk_data\tokenizers\punkt\PY3`).

See also

In the next recipe, we'll learn how to split sentences into individual words. After that, we'll cover how to use regular expressions to tokenize text. We'll cover how to train your own sentence tokenizer in an upcoming recipe, *Training a sentence tokenizer*.

Tokenizing sentences into words

In this recipe, we'll split a sentence into individual words. The simple task of creating a list of words from a string is an essential part of all text processing.

How to do it...

Basic word tokenization is very simple; use the `word_tokenize()` function:

```
>>> from nltk.tokenize import word_tokenize
>>> word_tokenize('Hello World.')
['Hello', 'World', '.']
```

How it works...

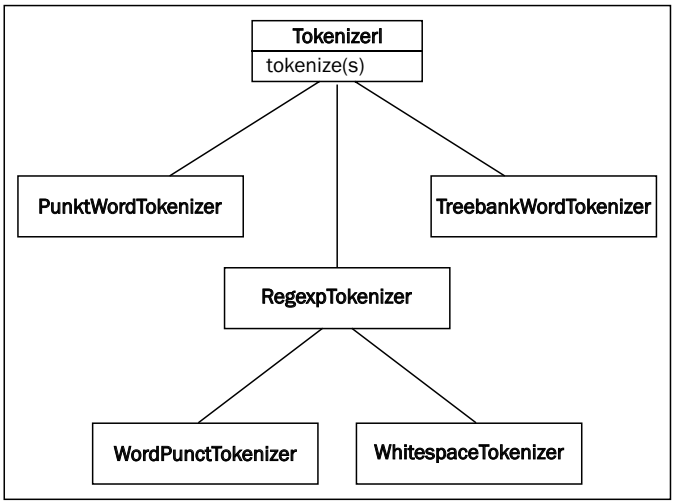
The `word_tokenize()` function is a wrapper function that calls `tokenize()` on an instance of the `TreebankWordTokenizer` class. It's equivalent to the following code:

```
>>> from nltk.tokenize import TreebankWordTokenizer
>>> tokenizer = TreebankWordTokenizer()
>>> tokenizer.tokenize('Hello World.')
['Hello', 'World', '.']
```

It works by separating words using spaces and punctuation. And as you can see, it does not discard the punctuation, allowing you to decide what to do with it.

There's more...

Ignoring the obviously named `WhitespaceTokenizer` and `SpaceTokenizer`, there are two other word tokenizers worth looking at: `PunktWordTokenizer` and `WordPunctTokenizer`. These differ from `TreebankWordTokenizer` by how they handle punctuation and contractions, but they all inherit from `TokenizerI`. The inheritance tree looks like what's shown in the following diagram:



Separating contractions

The `TreebankWordTokenizer` class uses conventions found in the Penn Treebank corpus. This corpus is one of the most used corpora for natural language processing, and was created in the 1980s by annotating articles from the *Wall Street Journal*. We'll be using this later in *Chapter 4*, *Part-of-speech Tagging*, and *Chapter 5*, *Extracting Chunks*.

One of the tokenizer's most significant conventions is to separate contractions. For example, consider the following code:

```
>>> word_tokenize("can't")
['ca', "n't"]
```

If you find this convention unacceptable, then read on for alternatives, and see the next recipe for tokenizing with regular expressions.

PunktWordTokenizer

An alternative word tokenizer is `PunktWordTokenizer`. It splits on punctuation, but keeps it with the word instead of creating separate tokens, as shown in the following code:

```
>>> from nltk.tokenize import PunktWordTokenizer
>>> tokenizer = PunktWordTokenizer()
>>> tokenizer.tokenize("Can't is a contraction.")
['Can', "'t", 'is', 'a', 'contraction.']
```

WordPunctTokenizer

Another alternative word tokenizer is `WordPunctTokenizer`. It splits all punctuation into separate tokens:

```
>>> from nltk.tokenize import WordPunctTokenizer
>>> tokenizer = WordPunctTokenizer()
>>> tokenizer.tokenize("Can't is a contraction.")
['Can', "'", 't', 'is', 'a', 'contraction', '.']
```

See also

For more control over word tokenization, you'll want to read the next recipe to learn how to use regular expressions and the `RegexpTokenizer` for tokenization. And for more on the Penn Treebank corpus, visit `http://www.cis.upenn.edu/~treebank/`.

Tokenizing sentences using regular expressions

Regular expressions can be used if you want complete control over how to tokenize text. As regular expressions can get complicated very quickly, I only recommend using them if the word tokenizers covered in the previous recipe are unacceptable.

Getting ready

First you need to decide how you want to tokenize a piece of text as this will determine how you construct your regular expression. The choices are:

- Match on the tokens
- Match on the separators or gaps

We'll start with an example of the first, matching alphanumeric tokens plus single quotes so that we don't split up contractions.

How to do it...

We'll create an instance of `RegexpTokenizer`, giving it a regular expression string to use for matching tokens:

```
>>> from nltk.tokenize import RegexpTokenizer
>>> tokenizer = RegexpTokenizer("[\w']+")
>>> tokenizer.tokenize("Can't is a contraction.")
["Can't", 'is', 'a', 'contraction']
```

There's also a simple helper function you can use if you don't want to instantiate the class, as shown in the following code:

```
>>> from nltk.tokenize import regexp_tokenize
>>> regexp_tokenize("Can't is a contraction.", "[\w']+")
["Can't", 'is', 'a', 'contraction']
```

Now we finally have something that can treat contractions as whole words, instead of splitting them into tokens.

How it works...

The `RegexpTokenizer` class works by compiling your pattern, then calling `re.findall()` on your text. You could do all this yourself using the `re` module, but `RegexpTokenizer` implements the `TokenizerI` interface, just like all the word tokenizers from the previous recipe. This means it can be used by other parts of the NLTK package, such as corpus readers, which we'll cover in detail in *Chapter 3*, *Creating Custom Corpora*. Many corpus readers need a way to tokenize the text they're reading, and can take optional keyword arguments specifying an instance of a `TokenizerI` subclass. This way, you have the ability to provide your own tokenizer instance if the default tokenizer is unsuitable.

There's more...

`RegexpTokenizer` can also work by matching the gaps, as opposed to the tokens. Instead of using `re.findall()`, the `RegexpTokenizer` class will use `re.split()`. This is how the `BlanklineTokenizer` class in `nltk.tokenize` is implemented.

Simple whitespace tokenizer

The following is a simple example of using `RegexpTokenizer` to tokenize on whitespace:

```
>>> tokenizer = RegexpTokenizer('\s+', gaps=True)
>>> tokenizer.tokenize("Can't is a contraction.")
["Can't", 'is', 'a', 'contraction.']
```

Notice that punctuation still remains in the tokens. The `gaps=True` parameter means that the pattern is used to identify gaps to tokenize on. If we used `gaps=False`, then the pattern would be used to identify tokens.

See also

For simpler word tokenization, see the previous recipe.

Training a sentence tokenizer

NLTK's default sentence tokenizer is general purpose, and usually works quite well. But sometimes it is not the best choice for your text. Perhaps your text uses nonstandard punctuation, or is formatted in a unique way. In such cases, training your own sentence tokenizer can result in much more accurate sentence tokenization.

Getting ready

For this example, we'll be using the `webtext` corpus, specifically the `overheard.txt` file, so make sure you've downloaded this corpus. The text in this file is formatted as dialog that looks like this:

```
White guy: So, do you have any plans for this evening?
Asian girl: Yeah, being angry!
White guy: Oh, that sounds good.
```

As you can see, this isn't your standard paragraph of sentences formatting, which makes it a perfect case for training a sentence tokenizer.

How to do it...

NLTK provides a `PunktSentenceTokenizer` class that you can train on raw text to produce a custom sentence tokenizer. You can get raw text either by reading in a file, or from an NLTK corpus using the `raw()` method. Here's an example of training a sentence tokenizer on dialog text, using `overheard.txt` from the `webtext` corpus:

```
>>> from nltk.tokenize import PunktSentenceTokenizer
>>> from nltk.corpus import webtext
>>> text = webtext.raw('overheard.txt')
>>> sent_tokenizer = PunktSentenceTokenizer(text)
```

Let's compare the results to the default sentence tokenizer, as follows:

```
>>> sents1 = sent_tokenizer.tokenize(text)
>>> sents1[0]
'White guy: So, do you have any plans for this evening?'

>>> from nltk.tokenize import sent_tokenize
>>> sents2 = sent_tokenize(text)
>>> sents2[0]
'White guy: So, do you have any plans for this evening?'
>>> sents1[678]
'Girl: But you already have a Big Mac...'
>>> sents2[678]
'Girl: But you already have a Big Mac...\\nHobo: Oh, this is all
theatrical.'
```

While the first sentence is the same, you can see that the tokenizers disagree on how to tokenize sentence `679` (this is the first sentence where the tokenizers diverge). The default tokenizer includes the next line of dialog, while our custom tokenizer correctly thinks that the next line is a separate sentence. This difference is a good demonstration of why it can be useful to train your own sentence tokenizer, especially when your text isn't in the typical paragraph-sentence structure.

How it works...

The `PunktSentenceTokenizer` class uses an unsupervised learning algorithm to learn what constitutes a sentence break. It is unsupervised because you don't have to give it any labeled training data, just raw text. You can read more about these kinds of algorithms at `https://en.wikipedia.org/wiki/Unsupervised_learning`. The specific technique used in this case is called sentence boundary detection and it works by counting punctuation and tokens that commonly end a sentence, such as a period or newline, then using the resulting frequencies to decide what the sentence boundaries should actually look like.

This is a simplified description of the algorithm—if you'd like more details, take a look at the source code of the `nltk.tokenize.punkt.PunktTrainer` class, which can be found online at `http://www.nltk.org/_modules/nltk/tokenize/punkt.html#PunktSentenceTokenizer`.

There's more...

The `PunktSentenceTokenizer` class learns from any string, which means you can open a text file and read its content. Here is an example of reading `overheard.txt` directly instead of using the `raw()` corpus method. This assumes that the `webtext` corpus is located in the standard directory at `/usr/share/nltk_data/corpora`. We also have to pass a specific encoding to the `open()` function, as follows, because the file is not in ASCII:

```
>>> with open('/usr/share/nltk_data/corpora/webtext/overheard.txt',
encoding='ISO-8859-2') as f:
...   text = f.read()
>>> sent_tokenizer = PunktSentenceTokenizer(text)
>>> sents = sent_tokenizer.tokenize(text)
>>> sents[0]
'White guy: So, do you have any plans for this evening?'
>>> sents[678]
'Girl: But you already have a Big Mac...'
```

Once you have a custom sentence tokenizer, you can use it for your own corpora. Many corpus readers accept a `sent_tokenizer` parameter, which lets you override the default sentence tokenizer object with your own sentence tokenizer. Corpus readers are covered in more detail in *Chapter 3*, *Creating Custom Corpora*.

See also

Most of the time, the default sentence tokenizer will be sufficient. This is covered in the first recipe, *Tokenizing text into sentences*.

Filtering stopwords in a tokenized sentence

Stopwords are common words that generally do not contribute to the meaning of a sentence, at least for the purposes of information retrieval and natural language processing. These are words such as *the* and *a*. Most search engines will filter out stopwords from search queries and documents in order to save space in their index.

Getting ready

NLTK comes with a `stopwords` corpus that contains word lists for many languages. Be sure to unzip the data file, so NLTK can find these word lists at `nltk_data/corpora/stopwords/`.

How to do it...

We're going to create a set of all English stopwords, then use it to filter stopwords from a sentence with the help of the following code:

```
>>> from nltk.corpus import stopwords
>>> english_stops = set(stopwords.words('english'))
>>> words = ["Can't", 'is', 'a', 'contraction']
>>> [word for word in words if word not in english_stops]
["Can't", 'contraction']
```

How it works...

The `stopwords` corpus is an instance `of nltk.corpus.reader.WordListCorpusReader`. As such, it has a `words()` method that can take a single argument for the file ID, which in this case is `'english'`, referring to a file containing a list of English stopwords. You could also call `stopwords.words()` with no argument to get a list of all stopwords in every language available.

There's more...

You can see the list of all English stopwords using `stopwords.words('english')` or by examining the word list file at `nltk_data/corpora/stopwords/english`. There are also stopword lists for many other languages. You can see the complete list of languages using the `fileids` method as follows:

```
>>> stopwords.fileids()
['danish', 'dutch', 'english', 'finnish', 'french', 'german',
'hungarian', 'italian', 'norwegian', 'portuguese', 'russian',
'spanish', 'swedish', 'turkish']
```

Any of these `fileids` can be used as an argument to the `words()` method to get a list of stopwords for that language. For example:

```
>>> stopwords.words('dutch')
['de', 'en', 'van', 'ik', 'te', 'dat', 'die', 'in', 'een', 'hij',
'het', 'niet', 'zijn', 'is', 'was', 'op', 'aan', 'met', 'als', 'voor',
'had', 'er', 'maar', 'om', 'hem', 'dan', 'zou', 'of', 'wat', 'mijn',
'men', 'dit', 'zo', 'door', 'over', 'ze', 'zich', 'bij', 'ook', 'tot',
'je', 'mij', 'uit', 'der', 'daar', 'haar', 'naar', 'heb', 'hoe',
'heeft', 'hebben', 'deze', 'u', 'want', 'nog', 'zal', 'me', 'zij',
'nu', 'ge', 'geen', 'omdat', 'iets', 'worden', 'toch', 'al', 'waren',
'veel', 'meer', 'doen', 'toen', 'moet', 'ben', 'zonder', 'kan',
'hun', 'dus', 'alles', 'onder', 'ja', 'eens', 'hier', 'wie', 'werd',
'altijd', 'doch', 'wordt', 'wezen', 'kunnen', 'ons', 'zelf', 'tegen',
'na', 'reeds', 'wil', 'kon', 'niets', 'uw', 'iemand', 'geweest',
'andere']
```

See also

If you'd like to create your own `stopwords` corpus, see the *Creating a wordlist corpus* recipe in *Chapter 3*, *Creating Custom Corpora*, to learn how to use `WordListCorpusReader`. We'll also be using stopwords in the *Discovering word collocations* recipe later in this chapter.

Looking up Synsets for a word in WordNet

WordNet is a lexical database for the English language. In other words, it's a dictionary designed specifically for natural language processing.

NLTK comes with a simple interface to look up words in WordNet. What you get is a list of **Synset** instances, which are groupings of synonymous words that express the same concept. Many words have only one Synset, but some have several. In this recipe, we'll explore a single Synset, and in the next recipe, we'll look at several in more detail.

Getting ready

Be sure you've unzipped the `wordnet` corpus at `nltk_data/corpora/wordnet`. This will allow `WordNetCorpusReader` to access it.

How to do it...

Now we're going to look up the Synset for `cookbook`, and explore some of the properties and methods of a Synset using the following code:

```
>>> from nltk.corpus import wordnet
>>> syn = wordnet.synsets('cookbook')[0]
>>> syn.name()
'cookbook.n.01'
>>> syn.definition()
'a book of recipes and cooking directions'
```

How it works...

You can look up any word in WordNet using `wordnet.synsets(word)` to get a list of Synsets. The list may be empty if the word is not found. The list may also have quite a few elements, as some words can have many possible meanings, and, therefore, many Synsets.

There's more...

Each Synset in the list has a number of methods you can use to learn more about it. The `name()` method will give you a unique name for the Synset, which you can use to get the Synset directly:

```
>>> wordnet.synset('cookbook.n.01')
Synset('cookbook.n.01')
```

The `definition()` method should be self-explanatory. Some Synsets also have an `examples()` method, which contains a list of phrases that use the word in context:

```
>>> wordnet.synsets('cooking')[0].examples()
['cooking can be a great art', 'people are needed who have experience
in cookery', 'he left the preparation of meals to his wife']
```

Working with hypernyms

Synsets are organized in a structure similar to that of an inheritance tree. More abstract terms are known as **hypernyms** and more specific terms are **hyponyms**. This tree can be traced all the way up to a root hypernym.

Hypernyms provide a way to categorize and group words based on their similarity to each other. The *Calculating WordNet Synset similarity* recipe details the functions used to calculate the similarity based on the distance between two words in the hypernym tree:

```
>>> syn.hypernyms()
[Synset('reference_book.n.01')]
>>> syn.hypernyms()[0].hyponyms()
[Synset('annual.n.02'), Synset('atlas.n.02'), Synset('cookbook.n.01'),
Synset('directory.n.01'), Synset('encyclopedia.n.01'),
Synset('handbook.n.01'), Synset('instruction_book.n.01'),
Synset('source_book.n.01'), Synset('wordbook.n.01')]
>>> syn.root_hypernyms()
[Synset('entity.n.01')]
```

As you can see, `reference_book` is a hypernym of `cookbook`, but `cookbook` is only one of the many hyponyms of `reference_book`. And all these types of books have the same root hypernym, which is `entity`, one of the most abstract terms in the English language. You can trace the entire path from entity down to `cookbook` using the `hypernym_paths()` method, as follows:

```
>>> syn.hypernym_paths()
[[Synset('entity.n.01'), Synset('physical_entity.n.01'),
Synset('object.n.01'), Synset('whole.n.02'), Synset('artifact.n.01'),
Synset('creation.n.02'), Synset('product.n.02'), Synset('work.n.02'),
Synset('publication.n.01'), Synset('book.n.01'), Synset('reference_
book.n.01'), Synset('cookbook.n.01')]]
```

The `hypernym_paths()` method returns a list of lists, where each list starts at the root hypernym and ends with the original Synset. Most of the time, you'll only get one nested list of Synsets.

Part of speech (POS)

You can also look up a simplified part-of-speech tag as follows:

```
>>> syn.pos()
'n'
```

There are four common part-of-speech tags (or POS tags) found in WordNet, as shown in the following table:

Part of speech	Tag
Noun	n
Adjective	a
Adverb	r
Verb	v

These POS tags can be used to look up specific Synsets for a word. For example, the word `'great'` can be used as a noun or an adjective. In WordNet, `'great'` has `1` noun Synset and `6` adjective Synsets, as shown in the following code:

```
>>> len(wordnet.synsets('great'))
7
>>> len(wordnet.synsets('great', pos='n'))
1
>>> len(wordnet.synsets('great', pos='a'))
6
```

These POS tags will be referenced more in the *Using WordNet for tagging* recipe in *Chapter 4*, *Part-of-speech Tagging*.

See also

In the next two recipes, we'll explore lemmas and how to calculate Synset similarity. And in *Chapter 2*, *Replacing and Correcting Words*, we'll use WordNet for lemmatization, synonym replacement, and then explore the use of antonyms.

Looking up lemmas and synonyms in WordNet

Building on the previous recipe, we can also look up lemmas in WordNet to find synonyms of a word. A **lemma** (in linguistics), is the canonical form or morphological form of a word.

How to do it...

In the following code, we'll find that there are two lemmas for the `cookbook` Synset using the `lemmas()` method:

```
>>> from nltk.corpus import wordnet
>>> syn = wordnet.synsets('cookbook')[0]
>>> lemmas = syn.lemmas()
>>> len(lemmas)
2
>>> lemmas[0].name()
'cookbook'
>>> lemmas[1].name()
'cookery_book'
>>> lemmas[0].synset() == lemmas[1].synset()
True
```

How it works...

As you can see, `cookery_book` and `cookbook` are two distinct lemmas in the same Synset. In fact, a lemma can only belong to a single Synset. In this way, a Synset represents a group of lemmas that all have the same meaning, while a lemma represents a distinct word form.

There's more...

Since all the lemmas in a Synset have the same meaning, they can be treated as synonyms. So if you wanted to get all synonyms for a Synset, you could do the following:

```
>>> [lemma.name() for lemma in syn.lemmas()]
['cookbook', 'cookery_book']
```

All possible synonyms

As mentioned earlier, many words have multiple Synsets because the word can have different meanings depending on the context. But, let's say you didn't care about the context, and wanted to get all the possible synonyms for a word:

```
>>> synonyms = []
>>> for syn in wordnet.synsets('book'):
...     for lemma in syn.lemmas():
...         synonyms.append(lemma.name())
>>> len(synonyms)
38
```

As you can see, there appears to be `38` possible synonyms for the word `'book'`. But in fact, some synonyms are verb forms, and many synonyms are just different usages of `'book'`. If, instead, we take the set of synonyms, there are fewer unique words, as shown in the following code:

```
>>> len(set(synonyms))
25
```

Antonyms

Some lemmas also have antonyms. The word `good`, for example, has 27 Synsets, five of which have lemmas with antonyms, as shown in the following code:

```
>>> gn2 = wordnet.synset('good.n.02')
>>> gn2.definition()
'moral excellence or admirableness'
>>> evil = gn2.lemmas()[0].antonyms()[0]
>>> evil.name
'evil'
>>> evil.synset().definition()
'the quality of being morally wrong in principle or practice'
>>> ga1 = wordnet.synset('good.a.01')
>>> ga1.definition()
'having desirable or positive qualities especially those suitable for
a thing specified'
>>> bad = ga1.lemmas()[0].antonyms()[0]
>>> bad.name()
'bad'
>>> bad.synset().definition()
'having undesirable or negative qualities'
```

The `antonyms()` method returns a list of lemmas. In the first case, as we can see in the previous code, the second Synset for `good` as a noun is defined as `moral excellence`, and its first antonym is `evil`, defined as `morally wrong`. In the second case, when `good` is used as an adjective to describe positive qualities, the first antonym is `bad`, which describes negative qualities.

See also

In the next recipe, we'll learn how to calculate Synset similarity. Then in *Chapter 2*, *Replacing and Correcting Words*, we'll revisit lemmas for lemmatization, synonym replacement, and antonym replacement.

Calculating WordNet Synset similarity

Synsets are organized in a *hypernym* tree. This tree can be used for reasoning about the similarity between the Synsets it contains. The closer the two Synsets are in the tree, the more similar they are.

How to do it...

If you were to look at all the hyponyms of `reference_book` (which is the hypernym of `cookbook`), you'd see that one of them is `instruction_book`. This seems intuitively very similar to a `cookbook`, so let's see what WordNet similarity has to say about it with the help of the following code:

```
>>> from nltk.corpus import wordnet
>>> cb = wordnet.synset('cookbook.n.01')
>>> ib = wordnet.synset('instruction_book.n.01')
>>> cb.wup_similarity(ib)
0.9166666666666666
```

So they are over 91% similar!

How it works...

The `wup_similarity` method is short for **Wu-Palmer Similarity**, which is a scoring method based on how similar the word senses are and where the Synsets occur relative to each other in the hypernym tree. One of the core metrics used to calculate similarity is the shortest path distance between the two Synsets and their common hypernym:

```
>>> ref = cb.hypernyms()[0]
>>> cb.shortest_path_distance(ref)
1
>>> ib.shortest_path_distance(ref)
1
>>> cb.shortest_path_distance(ib)
2
```

So `cookbook` and `instruction_book` must be very similar, because they are only one step away from the same `reference_book` hypernym, and, therefore, only two steps away from each other.

There's more...

Let's look at two dissimilar words to see what kind of score we get. We'll compare `dog` with `cookbook`, two seemingly very different words.

```
>>> dog = wordnet.synsets('dog')[0]
>>> dog.wup_similarity(cb)
0.38095238095238093
```

Wow, `dog` and `cookbook` are apparently 38% similar! This is because they share common hypernyms further up the tree:

```
>>> sorted(dog.common_hypernyms(cb))
[Synset('entity.n.01'), Synset('object.n.01'), Synset('physical_
entity.n.01'), Synset('whole.n.02')]
```

Comparing verbs

The previous comparisons were all between nouns, but the same can be done for verbs as well:

```
>>> cook = wordnet.synset('cook.v.01')
>>> bake = wordnet.0('bake.v.02')
>>> cook.wup_similarity(bake)
00.6666666666666666
```

The previous Synsets were obviously handpicked for demonstration, and the reason is that the hypernym tree for verbs has a lot more breadth and a lot less depth. While most nouns can be traced up to the hypernym `object`, thereby providing a basis for similarity, many verbs do not share common hypernyms, making WordNet unable to calculate the similarity. For example, if you were to use the Synset for `bake.v.01` in the previous code, instead of `bake.v.02`, the return value would be `None`. This is because the root hypernyms of both the Synsets are different, with no overlapping paths. For this reason, you also cannot calculate the similarity between words with different parts of speech.

Path and Leacock Chordorow (LCH) similarity

Two other similarity comparisons are the path similarity and the LCH similarity, as shown in the following code:

```
>>> cb.path_similarity(ib)
0.3333333333333333
>>> cb.path_similarity(dog)
0.07142857142857142
>>> cb.lch_similarity(ib)
2.538973871058276
>>> cb.lch_similarity(dog)
0.9985288301111273
```

As you can see, the number ranges are very different for these scoring methods, which is why I prefer the `wup_similarity` method.

See also

The recipe on *Looking up Synsets for a word in WordNet* has more details about hypernyms and the hypernym tree.

Discovering word collocations

Collocations are two or more words that tend to appear frequently together, such as United States. Of course, there are many other words that can come after United, such as United Kingdom and United Airlines. As with many aspects of natural language processing, context is very important. And for collocations, context is everything!

In the case of collocations, the context will be a document in the form of a list of words. Discovering collocations in this list of words means that we'll find common phrases that occur frequently throughout the text. For fun, we'll start with the script for *Monty Python and the Holy Grail*.

Getting ready

The script for *Monty Python and the Holy Grail* is found in the `webtext` corpus, so be sure that it's unzipped at `nltk_data/corpora/webtext/`.

How to do it...

We're going to create a list of all lowercased words in the text, and then produce `BigramCollocationFinder`, which we can use to find bigrams, which are pairs of words. These bigrams are found using association measurement functions in the `nltk.metrics` package, as follows:

```
>>> from nltk.corpus import webtext
>>> from nltk.collocations import BigramCollocationFinder
>>> from nltk.metrics import BigramAssocMeasures
>>> words = [w.lower() for w in webtext.words('grail.txt')]
>>> bcf = BigramCollocationFinder.from_words(words)
>>> bcf.nbest(BigramAssocMeasures.likelihood_ratio, 4)
[("'", 's'), ('arthur', ':'), ('#', '1'), ("'", 't')]
```

Well, that's not very useful! Let's refine it a bit by adding a word filter to remove punctuation and stopwords:

```
>>> from nltk.corpus import stopwords
>>> stopset = set(stopwords.words('english'))
>>> filter_stops = lambda w: len(w) < 3 or w in stopset
>>> bcf.apply_word_filter(filter_stops)
>>> bcf.nbest(BigramAssocMeasures.likelihood_ratio, 4)
[('black', 'knight'), ('clop', 'clop'), ('head', 'knight'), ('mumble',
'mumble')]
```

Much better, we can clearly see four of the most common bigrams in *Monty Python and the Holy Grail*. If you'd like to see more than four, simply increase the number to whatever you want, and the collocation finder will do its best.

How it works...

`BigramCollocationFinder` constructs two frequency distributions: one for each word, and another for bigrams. A frequency distribution, or `FreqDist` in NLTK, is basically an enhanced Python dictionary where the keys are what's being counted, and the values are the counts. Any filtering functions that are applied reduce the size of these two `FreqDists` by eliminating any words that don't pass the filter. By using a filtering function to eliminate all words that are one or two characters, and all English stopwords, we can get a much cleaner result. After filtering, the collocation finder is ready to accept a generic scoring function for finding collocations.

There's more...

In addition to `BigramCollocationFinder`, there's also `TrigramCollocationFinder`, which finds triplets instead of pairs. This time, we'll look for trigrams in Australian singles advertisements with the help of the following code:

```
>>> from nltk.collocations import TrigramCollocationFinder
>>> from nltk.metrics import TrigramAssocMeasures
>>> words = [w.lower() for w in webtext.words('singles.txt')]
>>> tcf = TrigramCollocationFinder.from_words(words)
>>> tcf.apply_word_filter(filter_stops)
>>> tcf.apply_freq_filter(3)
>>> tcf.nbest(TrigramAssocMeasures.likelihood_ratio, 4)
[('long', 'term', 'relationship')]
```

Now, we don't know whether people are looking for a long-term relationship or not, but clearly it's an important topic. In addition to the stopword filter, I also applied a frequency filter, which removed any trigrams that occurred less than three times. This is why only one result was returned when we asked for four because there was only one result that occurred more than two times.

Scoring functions

There are many more scoring functions available besides `likelihood_ratio()`. But other than `raw_freq()`, you may need a bit of a statistics background to understand how they work. Consult the NLTK API documentation for `NgramAssocMeasures` in the `nltk.metrics` package to see all the possible scoring functions.

Scoring ngrams

In addition to the `nbest()` method, there are two other ways to get ngrams (a generic term used for describing bigrams and trigrams) from a collocation finder:

- `above_score(score_fn, min_score)`: This can be used to get all ngrams with scores that are at least `min_score`. The `min_score` value that you choose will depend heavily on the `score_fn` you use.
- `score_ngrams(score_fn)`: This will return a list with tuple pairs of (ngram, score). This can be used to inform your choice for `min_score`.

See also

The `nltk.metrics` module will be used again in the *Measuring precision and recall of a classifier* and *Calculating high information words* recipes in *Chapter 7*, *Text Classification*.

2
Replacing and Correcting Words

In this chapter, we will cover the following recipes:

- Stemming words
- Lemmatizing words with WordNet
- Replacing words matching regular expressions
- Removing repeating characters
- Spelling correction with Enchant
- Replacing synonyms
- Replacing negations with antonyms

Introduction

In this chapter, we will go over various word replacement and correction techniques. The recipes cover the gamut of linguistic compression, spelling correction, and text normalization. All of these methods can be very useful for preprocessing text before search indexing, document classification, and text analysis.

Stemming words

Stemming is a technique to remove affixes from a word, ending up with the stem. For example, the stem of `cooking` is `cook`, and a good stemming algorithm knows that the `ing` suffix can be removed. Stemming is most commonly used by search engines for indexing words. Instead of storing all forms of a word, a search engine can store only the stems, greatly reducing the size of index while increasing retrieval accuracy.

One of the most common stemming algorithms is the **Porter stemming algorithm** by Martin Porter. It is designed to remove and replace well-known suffixes of English words, and its usage in NLTK will be covered in the next section.

The resulting stem is not always a valid word. For example, the stem of `cookery` is `cookeri`. This is a feature, not a bug.

How to do it...

NLTK comes with an implementation of the Porter stemming algorithm, which is very easy to use. Simply instantiate the `PorterStemmer` class and call the `stem()` method with the word you want to stem:

```
>>> from nltk.stem import PorterStemmer
>>> stemmer = PorterStemmer()
>>> stemmer.stem('cooking')
'cook'
>>> stemmer.stem('cookery')
'cookeri'
```

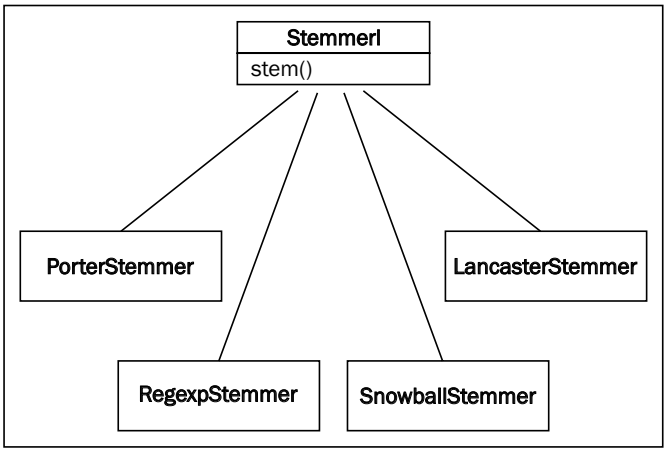
How it works...

The `PorterStemmer` class knows a number of regular word forms and suffixes and uses this knowledge to transform your input word to a final stem through a series of steps. The resulting stem is often a shorter word, or at least a common form of the word, which has the same root meaning.

There's more...

There are other stemming algorithms out there besides the Porter stemming algorithm, such as the **Lancaster stemming algorithm**, developed at Lancaster University. NLTK includes it as the `LancasterStemmer` class. At the time of writing this book, there is no definitive research demonstrating the superiority of one algorithm over the other. However, Porter stemming algorithm is generally the default choice.

All the stemmers covered next inherit from the `StemmerI` interface, which defines the `stem()` method. The following is an inheritance diagram that explains this:



The LancasterStemmer class

The functions of the `LancasterStemmer` class are just like the functions of the `PorterStemmer` class, but can produce slightly different results. It is known to be slightly more aggressive than the `PorterStemmer` functions:

```
>>> from nltk.stem import LancasterStemmer
>>> stemmer = LancasterStemmer()
>>> stemmer.stem('cooking')
'cook'
>>> stemmer.stem('cookery')
'cookery'
```

The RegexpStemmer class

You can also construct your own stemmer using the `RegexpStemmer` class. It takes a single regular expression (either compiled or as a string) and removes any prefix or suffix that matches the expression:

```
>>> from nltk.stem import RegexpStemmer
>>> stemmer = RegexpStemmer('ing')
>>> stemmer.stem('cooking')
'cook'
>>> stemmer.stem('cookery')
'cookery'
>>> stemmer.stem('ingleside')
'leside'
```

A `RegexpStemmer` class should only be used in very specific cases that are not covered by the `PorterStemmer` or the `LancasterStemmer` class because it can only handle very specific patterns and is not a general-purpose algorithm.

The SnowballStemmer class

The `SnowballStemmer` class supports 13 non-English languages. It also provides two English stemmers: the original porter algorithm as well as the new English stemming algorithm. To use the `SnowballStemmer` class, create an instance with the name of the language you are using and then call the `stem()` method. Here is a list of all the supported languages and an example using the Spanish `SnowballStemmer` class:

```
>>> from nltk.stem import SnowballStemmer
>>> SnowballStemmer.languages('danish', 'dutch', 'english', 'finnish',
'french', 'german', 'hungarian', 'italian', 'norwegian', 'porter',
'portuguese', 'romanian', 'russian', 'spanish', 'swedish')
>>> spanish_stemmer = SnowballStemmer('spanish')
>>> spanish_stemmer.stem('hola')
u'hol'
```

See also

In the next recipe, we will cover Lemmatization, which is quite similar to stemming, but subtly different.

Lemmatizing words with WordNet

Lemmatization is very similar to stemming, but is more akin to synonym replacement. A lemma is a root word, as opposed to the root stem. So unlike stemming, you are always left with a valid word that means the same thing. However, the word you end up with can be completely different. A few examples will explain this.

Getting ready

Make sure that you have unzipped the `wordnet` corpus in `nltk_data/corpora/wordnet`. This will allow the `WordNetLemmatizer` class to access WordNet. You should also be familiar with the part-of-speech tags covered in the *Looking up Synsets for a word in WordNet* recipe of *Chapter 1*, *Tokenizing Text and WordNet Basics*.

How to do it...

We will use the `WordNetLemmatizer` class to find lemmas:

```
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()
>>> lemmatizer.lemmatize('cooking')
'cooking'
>>> lemmatizer.lemmatize('cooking', pos='v')
'cook'
>>> lemmatizer.lemmatize('cookbooks')
'cookbook'
```

How it works...

The `WordNetLemmatizer` class is a thin wrapper around the `wordnet` corpus and uses the `morphy()` function of the `WordNetCorpusReader` class to find a lemma. If no lemma is found, or the word itself is a lemma, the word is returned as is. Unlike with stemming, knowing the part of speech of the word is important. As demonstrated previously, `cooking` does not return a different lemma unless you specify that the POS is a verb. This is because the default POS is a noun, and as a noun, `cooking` is its own lemma. On the other hand, `cookbooks` is a noun with its singular form, `cookbook`, as its lemma.

There's more...

Here's an example that illustrates one of the major differences between stemming and lemmatization:

```
>>> from nltk.stem import PorterStemmer
>>> stemmer = PorterStemmer()
>>> stemmer.stem('believes')
'believ'
>>> lemmatizer.lemmatize('believes')
'belief'
```

Instead of just chopping off the `es` like the `PorterStemmer` class, the `WordNetLemmatizer` class finds a valid root word. Where a stemmer only looks at the form of the word, the lemmatizer looks at the meaning of the word. By returning a lemma, you will always get a valid word.

Combining stemming with lemmatization

Stemming and lemmatization can be combined to compress words more than either process can by itself. These cases are somewhat rare, but they do exist:

```
>>> stemmer.stem('buses')
'buse'
>>> lemmatizer.lemmatize('buses')
'bus'
>>> stemmer.stem('bus')
'bu'
```

In this example, stemming saves one character, lemmatization saves two characters, and stemming the lemma saves a total of three characters out of five characters. That is nearly a 60% compression rate! This level of word compression over many thousands of words, while unlikely to always produce such high gains, can still make a huge difference.

See also

In the previous recipe, we covered the basics of stemming and WordNet was introduced in the *Looking up Synsets for a word in WordNet* and *Looking up lemmas and synonyms in WordNet* recipes of *Chapter 1*, *Tokenizing Text and WordNet Basics*. Looking forward, we will cover the *Using WordNet for tagging* recipe in *Chapter 4*, *Part-of-speech Tagging*.

Replacing words matching regular expressions

Now, we are going to get into the process of replacing words. If stemming and lemmatization are a kind of linguistic compression, then word replacement can be thought of as error correction or text normalization.

In this recipe, we will replace words based on regular expressions, with a focus on expanding contractions. Remember when we were tokenizing words in *Chapter 1*, *Tokenizing Text and WordNet Basics*, and it was clear that most tokenizers had trouble with contractions? This recipe aims to fix this by replacing contractions with their expanded forms, for example, by replacing "can't" with "cannot" or "would've" with "would have".

Getting ready

Understanding how this recipe works will require a basic knowledge of regular expressions and the `re` module. The key things to know are matching patterns and the `re.sub()` function.

How to do it...

First, we need to define a number of replacement patterns. This will be a list of tuple pairs, where the first element is the pattern to match with and the second element is the replacement.

Next, we will create a `RegexpReplacer` class that will compile the patterns and provide a `replace()` method to substitute all the found patterns with their replacements.

The following code can be found in the `replacers.py` module in the book's code bundle and is meant to be imported, not typed into the console:

```
import re

replacement_patterns = [
  (r'won\'t', 'will not'),
  (r'can\'t', 'cannot'),
  (r'i\'m', 'i am'),
  (r'ain\'t', 'is not'),
  (r'(\w+)\'ll', '\g<1> will'),
  (r'(\w+)n\'t', '\g<1> not'),
  (r'(\w+)\'ve', '\g<1> have'),
  (r'(\w+)\'s', '\g<1> is'),
  (r'(\w+)\'re', '\g<1> are'),
  (r'(\w+)\'d', '\g<1> would')
]

class RegexpReplacer(object):
  def __init__(self, patterns=replacement_patterns):
    self.patterns = [(re.compile(regex), repl) for (regex, repl) in
      patterns]

  def replace(self, text):
    s = text
    for (pattern, repl) in self.patterns:
      s = re.sub(pattern, repl, s)
    return s
```

How it works...

Here is a simple usage example:

```
>>> from replacers import RegexpReplacer
>>> replacer = RegexpReplacer()
>>> replacer.replace("can't is a contraction")
'cannot is a contraction'
>>> replacer.replace("I should've done that thing I didn't do")
'I should have done that thing I did not do'
```

The `RegexpReplacer.replace()` function works by replacing every instance of a replacement pattern with its corresponding substitution pattern. In replacement patterns, we have defined tuples such as `r'(\w+)\'ve'` and `'\g<1> have'`. The first element matches a group of ASCII characters followed by `'ve`. By grouping the characters before `'ve` in parenthesis, a match group is found and can be used in the substitution pattern with the `\g<1>` reference. So, we keep everything before `'ve`, then replace `'ve` with the word `h`ave. This is how `should've` can become `should have`.

There's more...

This replacement technique can work with any kind of regular expression, not just contractions. So, you can replace any occurrence of `&` with `and`, or eliminate all occurrences of `-` by replacing it with an empty string. The `RegexpReplacer` class can take any list of replacement patterns for whatever purpose.

Replacement before tokenization

Let's try using the `RegexpReplacer` class as a preliminary step before tokenization:

```
>>> from nltk.tokenize import word_tokenize
>>> from replacers import RegexpReplacer
>>> replacer = RegexpReplacer()
>>> word_tokenize("can't is a contraction")
['ca', "n't", 'is', 'a', 'contraction']
>>> word_tokenize(replacer.replace("can't is a contraction"))
['can', 'not', 'is', 'a', 'contraction']
```

Much better! By eliminating the contractions in the first place, the tokenizer will produce cleaner results. Cleaning up the text before processing is a common pattern in natural language processing.

See also

For more information on tokenization, see the first three recipes in *Chapter 1*, *Tokenizing Text and WordNet Basics*. For more replacement techniques, continue reading the rest of this chapter.

Removing repeating characters

In everyday language, people are often not strictly grammatical. They will write things such as `I looooooove it` in order to emphasize the word `love`. However, computers don't know that "looooooove" is a variation of "love" unless they are told. This recipe presents a method to remove these annoying repeating characters in order to end up with a *proper* English word.

Getting ready

As in the previous recipe, we will be making use of the `re` module, and more specifically, backreferences. A **backreference** is a way to refer to a previously matched group in a regular expression. This will allow us to match and remove repeating characters.

How to do it...

We will create a class that has the same form as the `RegexpReplacer` class from the previous recipe. It will have a `replace()` method that takes a single word and returns a more correct version of that word, with the dubious repeating characters removed. This code can be found in `replacers.py` in the book's code bundle and is meant to be imported:

```
import re

class RepeatReplacer(object):
  def __init__(self):
    self.repeat_regexp = re.compile(r'(\w*)(\w)\2(\w*)')
    self.repl = r'\1\2\3'

  def replace(self, word):
    repl_word = self.repeat_regexp.sub(self.repl, word)

    if repl_word != word:
      return self.replace(repl_word)
    else:
      return repl_word
```

And now some example use cases:

```
>>> from replacers import RepeatReplacer
>>> replacer = RepeatReplacer()
>>> replacer.replace('looooove')
'love'
>>> replacer.replace('oooooh')
'oh'
>>> replacer.replace('goose')
'gose'
```

How it works...

The `RepeatReplacer` class starts by compiling a regular expression to match and define a replacement string with backreferences. The `repeat_regexp` pattern matches three groups:

- `0` or more starting characters `(\w*)`
- A single character (`\w`) that is followed by another instance of that character `(\2)`
- `0` or more ending characters `(\w*)`

The replacement string is then used to keep all the matched groups, while discarding the backreference to the second group. So, the word `looooove` gets split into `(looo)(o)o(ve)` and then recombined as `loooove`, discarding the last `o`. This continues until only one `o` remains, when `repeat_regexp` no longer matches the string and no more characters are removed.

There's more...

In the preceding examples, you can see that the `RepeatReplacer` class is a bit too greedy and ends up changing `goose` into `gose`. To correct this issue, we can augment the `replace()` function with a WordNet lookup. If WordNet recognizes the word, then we can stop replacing characters. Here is the WordNet-augmented version:

```
import re
from nltk.corpus import wordnet

class RepeatReplacer(object):
  def __init__(self):
    self.repeat_regexp = re.compile(r'(\w*)(\w)\2(\w*)')
    self.repl = r'\1\2\3'
```

```
def replace(self, word):
  if wordnet.synsets(word):
    return word
  repl_word = self.repeat_regexp.sub(self.repl, word)

  if repl_word != word:
    return self.replace(repl_word)
  else:
    return repl_word
```

Now, `goose` will be found in WordNet, and no character replacement will take place. Also, `oooooh` will become `ooh` instead of `oh` because `ooh` is actually a word in WordNet, defined as an expression of admiration or pleasure.

See also

Read the next recipe to learn how to correct misspellings. For more information on WordNet, refer to the WordNet recipes in *Chapter 1*, *Tokenizing Text and WordNet Basics*. We will also be using WordNet for antonym replacement later in this chapter.

Spelling correction with Enchant

Replacing repeating characters is actually an extreme form of spelling correction. In this recipe, we will take on the less extreme case of correcting minor spelling issues using **Enchant**—a spelling correction API.

Getting ready

You will need to install Enchant and a dictionary for it to use. Enchant is an offshoot of the AbiWord open source word processor, and more information on it can be found at `http://www.abisource.com/projects/enchant/`.

For dictionaries, **Aspell** is a good open source spellchecker and dictionary that can be found at `http://aspell.net/`.

Finally, you will need the **PyEnchant** library, which can be found at the following link: `http://pythonhosted.org/pyenchant/`

You should be able to install it with the `easy_install` command that comes with Python setuptools, such as by typing `sudo easy_install pyenchant` on Linux or Unix. On a Mac machine, PyEnchant may be difficult to install. If you have difficulties, consult `http://pythonhosted.org/pyenchant/download.html`.

How to do it...

We will create a new class called `SpellingReplacer` in `replacers.py`, and this time, the `replace()` method will check Enchant to see whether the word is valid. If not, we will look up the suggested alternatives and return the best match using `nltk.metrics.edit_distance()`:

```
import enchant
from nltk.metrics import edit_distance

class SpellingReplacer(object):
  def __init__(self, dict_name='en', max_dist=2):
    self.spell_dict = enchant.Dict(dict_name)
    self.max_dist = max_dist

  def replace(self, word):
    if self.spell_dict.check(word):
      return word
    suggestions = self.spell_dict.suggest(word)

    if suggestions and edit_distance(word, suggestions[0]) <=
      self.max_dist:
      return suggestions[0]
    else:
      return word
```

The preceding class can be used to correct English spellings, as follows:

```
>>> from replacers import SpellingReplacer
>>> replacer = SpellingReplacer()
>>> replacer.replace('cookbok')
'cookbook'
```

How it works...

The `SpellingReplacer` class starts by creating a reference to an Enchant dictionary. Then, in the `replace()` method, it first checks whether the given word is present in the dictionary. If it is, no spelling correction is necessary and the word is returned. If the word is not found, it looks up a list of suggestions and returns the first suggestion, as long as its edit distance is less than or equal to `max_dist`. The edit distance is the number of character changes necessary to transform the given word into the suggested word. The `max_dist` value then acts as a constraint on the Enchant `suggest` function to ensure that no unlikely replacement words are returned. Here is an example showing all the suggestions for `languege`, a misspelling of `language`:

```
>>> import enchant
>>> d = enchant.Dict('en')
>>> d.suggest('languege')
['language', 'languages', 'languor', "language's"]
```

Except for the correct suggestion, `language`, all the other words have an edit distance of three or greater. You can try this yourself with the following code:

```
>>> from nltk.metrics import edit_distance
>>> edit_distance('language', 'languege')
1
>>> edit_distance('language', 'languo')
3
```

There's more...

You can use language dictionaries other than `en`, such as `en_GB`, assuming the dictionary has already been installed. To check which other languages are available, use `enchant.list_languages()`:

```
>>> enchant.list_languages()
['en', 'en_CA', 'en_GB', 'en_US']
```

If you try to use a dictionary that doesn't exist, you will get `enchant.DictNotFoundError`. You can first check whether the dictionary exists using `enchant.dict_exists()`, which will return `True` if the named dictionary exists, or `False` otherwise.

The en_GB dictionary

Always ensure that you use the correct dictionary for whichever language you are performing spelling correction on. The `en_US` dictionary can give you different results than `en_GB`, such as for the word `theater`. The word `theater` is the American English spelling whereas the British English spelling is `theatre`:

```
>>> import enchant
>>> dUS = enchant.Dict('en_US')
>>> dUS.check('theater')
True
>>> dGB = enchant.Dict('en_GB')
>>> dGB.check('theater')
False
```

```
>>> from replacers import SpellingReplacer
>>> us_replacer = SpellingReplacer('en_US')
>>> us_replacer.replace('theater')
'theater'
>>> gb_replacer = SpellingReplacer('en_GB')
>>> gb_replacer.replace('theater')
'theatre'
```

Personal word lists

Enchant also supports personal word lists. These can be combined with an existing dictionary, allowing you to augment the dictionary with your own words. So, let's say you had a file named `mywords.txt` that had `nltk` on one line. You could then create a dictionary augmented with your personal word list as follows:

```
>>> d = enchant.Dict('en_US')
>>> d.check('nltk')
False
>>> d = enchant.DictWithPWL('en_US', 'mywords.txt')
>>> d.check('nltk')
True
```

To use an augmented dictionary with our `SpellingReplacer` class, we can create a subclass in `replacers.py` that takes an existing spelling dictionary:

```
class CustomSpellingReplacer(SpellingReplacer):
  def __init__(self, spell_dict, max_dist=2):
    self.spell_dict = spell_dict
    self.max_dist = max_dist
```

This `CustomSpellingReplacer` class will not replace any words that you put into `mywords.txt`:

```
>>> from replacers import CustomSpellingReplacer
>>> d = enchant.DictWithPWL('en_US', 'mywords.txt')
>>> replacer = CustomSpellingReplacer(d)
>>> replacer.replace('nltk')
'nltk'
```

See also

The previous recipe covered an extreme form of spelling correction by replacing repeating characters. You can also perform spelling correction by simple word replacement as discussed in the next recipe.

Replacing synonyms

It is often useful to reduce the vocabulary of a text by replacing words with common synonyms. By compressing the vocabulary without losing meaning, you can save memory in cases such as *frequency analysis* and *text indexing*. More details about these topics are available at `https://en.wikipedia.org/wiki/Frequency_analysis` and `https://en.wikipedia.org/wiki/Full_text_search`. Vocabulary reduction can also increase the occurrence of significant collocations, which was covered in the *Discovering word collocations* recipe of *Chapter 1*, *Tokenizing Text and WordNet Basics*.

Getting ready

You will need a defined mapping of a word to its synonym. This is a simple controlled vocabulary. We will start by hardcoding the synonyms as a Python dictionary, and then explore other options to store synonym maps.

How to do it...

We'll first create a `WordReplacer` class in `replacers.py` that takes a word replacement mapping:

```
class WordReplacer(object):
  def __init__(self, word_map):
    self.word_map = word_map

  def replace(self, word):
    return self.word_map.get(word, word)
```

Then, we can demonstrate its usage for simple word replacement:

```
>>> from replacers import WordReplacer
>>> replacer = WordReplacer({'bday': 'birthday'})
>>> replacer.replace('bday')
'birthday'
>>> replacer.replace('happy')
'happy'
```

How it works...

The `WordReplacer` class is simply a class wrapper around a Python dictionary. The `replace()` method looks up the given word in its `word_map` dictionary and returns the replacement synonym if it exists. Otherwise, the given word is returned as is.

If you were only using the `word_map` dictionary, you wouldn't need the `WordReplacer` class and could instead call `word_map.get()` directly. However, `WordReplacer` can act as a base class for other classes that construct the `word_map` dictionary from various file formats. Read on for more information.

There's more...

Hardcoding synonyms in a Python dictionary is not a good long-term solution. Two better alternatives are to store the synonyms in a CSV file or in a YAML file. Choose whichever format is easiest for those who maintain your synonym vocabulary. Both of the classes outlined in the following section inherit the `replace()` method from `WordReplacer`.

CSV synonym replacement

The `CsvWordReplacer` class extends `WordReplacer` in `replacers.py` in order to construct the `word_map` dictionary from a CSV file:

```
import csv

class CsvWordReplacer(WordReplacer):
  def __init__(self, fname):
    word_map = {}
    for line in csv.reader(open(fname)):
      word, syn = line
      word_map[word] = syn
    super(CsvWordReplacer, self).__init__(word_map)
```

Your CSV file should consist of two columns, where the first column is the word and the second column is the synonym meant to replace it. If this file is called `synonyms.csv` and the first line is `bday,birthday`, then you can perform the following:

```
>>> from replacers import CsvWordReplacer
>>> replacer = CsvWordReplacer('synonyms.csv')
>>> replacer.replace('bday')
'birthday'
>>> replacer.replace('happy')
'happy'
```

YAML synonym replacement

If you have PyYAML installed, you can create `YamlWordReplacer` in `replacers.py` as shown in the following:

```
import yaml

class YamlWordReplacer(WordReplacer):
  def __init__(self, fname):
    word_map = yaml.load(open(fname))
    super(YamlWordReplacer, self).__init__(word_map)
```

Download and installation instructions for PyYAML are located at `http://pyyaml.org/wiki/PyYAML`. You can also type `pip install pyyaml` on the command prompt

Your YAML file should be a simple mapping of `word: synonym`, such as `bday: birthday`. Note that the YAML syntax is very particular, and the space after the colon is required. If the file is named `synonyms.yaml`, then you can perform the following:

```
>>> from replacers import YamlWordReplacer
>>> replacer = YamlWordReplacer('synonyms.yaml')
>>> replacer.replace('bday')
'birthday'
>>> replacer.replace('happy')
'happy'
```

See also

You can use the `WordReplacer` class to perform any kind of word replacement, even spelling correction for more complicated words that can't be automatically corrected, as we did in the previous recipe. In the next recipe, we will cover antonym replacement.

Replacing negations with antonyms

The opposite of synonym replacement is **antonym replacement**. An **antonym** is a word that has the opposite meaning of another word. This time, instead of creating custom word mappings, we can use WordNet to replace words with unambiguous antonyms. Refer to the *Looking up lemmas and synonyms in WordNet* recipe in *Chapter 1*, *Tokenizing Text and WordNet Basics*, for more details on antonym lookups.

How to do it...

Let's say you have a sentence like `let's not uglify our code`. With antonym replacement, you can replace `not uglify` with `beautify`, resulting in the sentence `let's beautify our code`. To do this, we will create an `AntonymReplacer` class in `replacers.py` as follows:

```
from nltk.corpus import wordnet

class AntonymReplacer(object):
  def replace(self, word, pos=None):
    antonyms = set()
    for syn in wordnet.synsets(word, pos=pos):
      for lemma in syn.lemmas():
        for antonym in lemma.antonyms():
          antonyms.add(antonym.name())
    if len(antonyms) == 1:
      return antonyms.pop()
    else:
      return None

  def replace_negations(self, sent):
    i, l = 0, len(sent)
    words = []
    while i < l:
      word = sent[i]
      if word == 'not' and i+1 < l:
        ant = self.replace(sent[i+1])
        if ant:
          words.append(ant)
          i += 2
          continue
      words.append(word)
      i += 1
    return words
```

Now, we can tokenize the original sentence into `["let's", 'not', 'uglify', 'our', 'code']` and pass this to the `replace_negations()` function. Here are some examples:

```
>>> from replacers import AntonymReplacer
>>> replacer = AntonymReplacer()
>>> replacer.replace('good')
>>> replacer.replace('uglify')
'beautify'
>>> sent = ["let's", 'not', 'uglify', 'our', 'code']
>>> replacer.replace_negations(sent)
["let's", 'beautify', 'our', 'code']
```

How it works...

The `AntonymReplacer` class has two methods: `replace()` and `replace_negations()`. The `replace()` method takes a single word and an optional part-of-speech tag, then looks up the Synsets for the word in WordNet. Going through all the Synsets and every lemma of each Synset, it creates a set of all antonyms found. If only one antonym is found, then it is an unambiguous replacement. If there is more than one antonym, which can happen quite often, then we don't know for sure which antonym is correct. In the case of multiple antonyms (or no antonyms), `replace()` returns `None` as it cannot make a decision.

In `replace_negations()`, we look through a tokenized sentence for the word `not`. If `not` is found, then we try to find an antonym for the next word using `replace()`. If we find an antonym, then it is appended to the list of words, replacing `not` and the original word. All other words are appended as is, resulting in a tokenized sentence with unambiguous negations replaced by their antonyms.

There's more...

As unambiguous antonyms aren't very common in WordNet, you might want to create a custom antonym mapping in the same way we did for synonyms. This `AntonymWordReplacer` can be constructed by inheriting from both `WordReplacer` and `AntonymReplacer`:

```
class AntonymWordReplacer(WordReplacer, AntonymReplacer):
  pass
```

The order of inheritance is very important, as we want the initialization and replace function of `WordReplacer` combined with the `replace_negations` function from `AntonymReplacer`. The result is a replacer that can perform the following:

```
>>> from replacers import AntonymWordReplacer
>>> replacer = AntonymWordReplacer({'evil': 'good'})
>>> replacer.replace_negations(['good', 'is', 'not', 'evil'])
['good', 'is', 'good']
```

Of course, you can also inherit from `CsvWordReplacer` or `YamlWordReplacer` instead of `WordReplacer` if you want to load the antonym word mappings from a file.

See also

The previous recipe covers the `WordReplacer` from the perspective of synonym replacement. In *Chapter 1*, *Tokenizing Text and WordNet Basics*, WordNet usage is covered in detail in the *Looking up Synsets for a word in WordNet* and *Looking up lemmas and synonyms in WordNet* recipes.