CHAPTER 13

Textual Data Analysis with NLTK

In this book, you have seen various analysis techniques and numerous examples that worked on data in numerical or tabular form, which is easily processed through mathematical expressions and statistical techniques. But most of the data is composed of text, which responds to grammatical rules (or sometimes not even that :)) that differ from language to language. In text, the words and the meanings attributable to the words (as well as the emotions they transmit) can be a very useful source of information.

In this chapter, you will learn about some text analysis techniques using the NLTK (Natural Language Toolkit) library, which allows you to perform otherwise complex operations. Furthermore, the topics covered will help you understand this important part of data analysis.

Text Analysis Techniques

In recent years, with the advent of Big Data and the immense amount of textual data coming from the Internet, a lot of text analysis techniques have been developed by necessity. In fact, this form of data can be very difficult to analyze, but at the same time represents a source of a lot of useful information, also given the enormous availability of data. Just think of all the literature produced—the numerous posts published on the Internet, for example. Comments on social networks and chats can also be a great source of data, especially to understand the degree of approval or disapproval of a particular topic.

Analyzing these texts has therefore become a source of enormous interest, and there are many techniques that have been introduced for this purpose, creating a real discipline in itself. Some of the more important techniques are listed here.

For preprocessing:

- Lowercase conversion
- Word and sentence tokenization
- Punctuation mark removal
- Stopword removal
- Stemming
- Lemmatization

For text analysis:

- Analysis of the frequency distribution of words
- Pattern recognition

F. Nelli, *Python Data Analytics*, https://doi.org/10.1007/978-1-4842-9532-8_13

- Tagging
- Analysis of links and associations
- Sentiment analysis

The Natural Language Toolkit (NLTK)

If you program in Python and want to analyze data in text form, one of the most commonly used tools at the moment is the Python Natural Language Toolkit (NLTK).

NLTK is nothing more than a Python library (`www.nltk.org`) in which there are many tools specialized in processing and text data analysis. NLTK was created in 2001 for educational purposes, then over time it developed to such an extent that it became a real analysis tool.

Within the NLTK library, there is also a large collection of sample texts, called *corpora*. This collection of texts is taken largely from literature and is very useful as a basis for the application of the techniques developed with the NLTK library. In particular, it's used to perform tests (a role similar to the MNIST dataset present in TensorFlow, which is discussed in Chapter 9).

Installing NLTK on your computer is a very simple operation.

If you are not currently using an Anaconda platform, you can install it using the PyPI system.

```
pip install nltk
```

If, on the other hand, you have an Anaconda platform to develop your projects in Python, on the virtual environment you want to use, install the `nltk` package graphically via Anaconda Navigator, or via the command console:

```
conda install nltk
```

Import the NLTK Library and the NLTK Downloader Tool

In order to be more confident with NLTK, there is no better method than working directly with the Python code. This way, you can see and gradually understand the operation of this library.

The first thing you need to do is open a Jupyter Notebook. The first command imports the NLTK library.

```
import nltk
```

Then you need to import text from the corpora collection. To do this, there is a function called `nltk.download_shell()`, which opens a tool called NLTK Downloader. The downloader allows you to make selections through a guided choice of options.

If you enter this command on the terminal:

```
nltk.download_shell()
```

You will see in output the NLTK Downloader suggesting various options in text format, as shown in Figure 13-1.

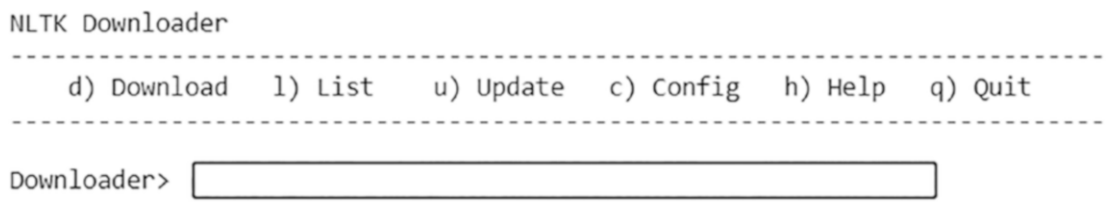

Figure 13-1. *The NTLK Downloader in a Jupyter Notebook*

Now the tool is waiting for an option. If you want to see a list of possible NLTK extensions, enter `l` for list and press Enter. You will immediately see a list of all the possible packages belonging to NLTK that you can download to extend the functionality of NLTK, including the texts of the corpora collection.

```
Packages:
  [ ] abc..................... Australian Broadcasting Commission 2006
  [ ] alpino.................. Alpino Dutch Treebank
  [ ] averaged_perceptron_tagger Averaged Perceptron Tagger
  [ ] averaged_perceptron_tagger_ru Averaged Perceptron Tagger (Russian)
  [ ] basque_grammars..... Grammars for Basque
  [ ] biocreative_ppi..... BioCreAtIvE (Critical Assessment of Information
                           Extraction Systems in Biology)
  [ ] bllip_wsj_no_aux.... BLLIP Parser: WSJ Model
  [ ] book_grammars....... Grammars from NLTK Book
  [ ] brown............... Brown Corpus
  [ ] brown_tei........... Brown Corpus (TEI XML Version)
  [ ] cess_cat............ CESS-CAT Treebank
  [ ] cess_esp............ CESS-ESP Treebank
  [ ] chat80.............. Chat-80 Data Files
  [ ] city_database....... City Database
  [ ] cmudict............. The Carnegie Mellon Pronouncing Dictionary (0.6)
  [ ] comparative_sentences Comparative Sentence Dataset
  [ ] comtrans............ ComTrans Corpus Sample
  [ ] conll2000........... CONLL 2000 Chunking Corpus
  [ ] conll2002........... CONLL 2002 Named Entity Recognition Corpus
Hit Enter to continue:
```

Pressing Enter again will continue displaying the list by showing other packages in alphabetical order. Press Enter until the list is finished to see all the possible packages. At the end of the list, the different initial options of the NLTK Downloader will reappear.

To create a series of examples to learn about the library, you need a series of texts to work on. An excellent source of texts suitable for this purpose is the Gutenberg corpus, present in the corpora collection. The Gutenberg corpus is a small selection of texts extracted from the electronic archive called the Project Gutenberg (`www.gutenberg.org`). There are over 25,000 e-books in this archive.

■ **Note** Attention, in some countries such as Italy, this site is not accessible.

To download this package, first enter the **d** option to download it. The tool will ask you for the package name, so you then enter the name gutenberg.

```
---------------------------------------------------------------------------
    d) Download   l) List    u) Update   c) Config   h) Help   q) Quit
---------------------------------------------------------------------------
Downloader> d
Download which package (l=list; x=cancel)?
  Identifier> gutenberg
```

At this point the package will start to download.

When you already know the name of the package you want to download, just enter the command `nltk.download()` with the package name as an argument. This will not open the NLTK Downloader tool, but will directly download the required package. So the previous operation is equivalent to writing:

```
nltk.download ('gutenberg')
```

Once it's completed, you can see the contents of the package thanks to the `fileids()` function, which shows the names of the files contained in it.

```
gb = nltk.corpus.gutenberg
print ("Gutenberg files:", gb.fileids ())
```

An array will appear on the terminal with all the text files contained in the gutenberg package.

```
Out [ ]:
Gutenberg files :  ['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', 'bible-
kjv.txt', 'blake-poems.txt', 'bryant-stories.txt', 'burgess-busterbrown.txt', 'carroll-
alice.txt', 'chesterton-ball.txt', 'chesterton-brown.txt', 'chesterton-thursday.txt',
'edgeworth-parents.txt', 'melville-moby_dick.txt', 'milton-paradise.txt', 'shakespeare-
caesar.txt', 'shakespeare-hamlet.txt', 'shakespeare-macbeth.txt', 'whitman-leaves.txt']
```

To access the internal content of one of these files, you first select one, for example Shakespeare's Macbeth (`shakespeare-macbeth.txt`), and then assign it to a variable of convenience. An extraction mode is for words, that is, you want to create an array containing words as elements. In this regard, you need to use the `words()` function.

```
macbeth = nltk.corpus.gutenberg.words ('shakespeare-macbeth.txt')
```

If you want to see the length of this text (in words), you can use the `len()` function.

```
len (macbeth)
Out [ ]:
23140
```

The text used for these examples is therefore composed of 23140 words.

The `macbeth` variable is a long array containing the words of the text. If you want to see the first ten words of the text, you can write the following command.

```
macbeth [:10]
Out [ ]:
['[',
```

```
 'The',
 'Tragedie',
 'of',
 'Macbeth',
 'by',
 'William',
 'Shakespeare',
 '1603',
 ']']
```

As you can see, the first ten words contain the title of the work, but also the square brackets, which indicate the beginning and end of a sentence. If you had used the sentence extraction mode with the `sents()` function, you would have obtained a more structured array, with each sentence as an element. These elements, in turn, would be arrays with words for elements.

```
macbeth_sents = nltk.corpus.gutenberg.sents ('shakespeare-macbeth.txt')
macbeth_sents [: 5]
Out [ ]:
[['[',
  'The',
  'Tragedie',
  'of',
  'Macbeth',
  'by',
  'William',
  'Shakespeare',
  '1603',
  ']'],
 ['Actus', 'Primus', '.'],
 ['Scoena', 'Prima', '.'],
 ['Thunder', 'and', 'Lightning', '.'],
 ['Enter', 'three', 'Witches', '.']]
```

Search for a Word with NLTK

One of the most basic things you need to do when you have an NLTK corpus (that is, an array of words extracted from a text) is to do research inside it. The concept of research is slightly different than what you are used to.

The `concordance()` function looks for all occurrences of a word passed as an argument within a corpus.

The first time you run this command, the system will take several seconds to return a result. The subsequent times will be faster. In fact, the first time this command is executed on a corpus, it creates an indexing of the content to perform the search, which once created will be used in subsequent calls. This explains why the system takes longer the first time.

First, make sure that the corpus is an object `nltk.Text`, and then search internally for the word `'Stage'`.

```
text = nltk.Text(macbeth)
text.concordance('Stage')
Out [ ]:
Displaying 3 of 3 matches:
nts with Dishes and Seruice ouer the Stage . Then enter Macbeth Macb . If it we
with mans Act , Threatens his bloody Stage : byth ' Clock ' tis Day , And yet d
 struts and frets his houre vpon the Stage , And then is heard no more . It is
```

You have obtained three different occurrences of the text.

Another form of searching for a word present in NLTK is that of context. That is, the previous word and the word next to the one you are looking for. To do this, you must use the `common_contexts()` function.

```
text.common_contexts(['Stage'])
Out [ ]:
the_ bloody_: the_,
```

If you look at the results of the previous research, you can see that the three results correspond to what has been said.

Once you understand how NLTK conceives the concept of the word and its context during the search, it is easy to understand the concept of a synonym. That is, it is assumed that all words that have the same context can be possible synonyms. To search for all words that have the same context as the searched one, you must use the `similar()` function.

```
text.similar('Stage')
Out [ ]:
fogge ayre bleeding reuolt good shew heeles skie other sea feare
consequence heart braine seruice herbenger lady round deed doore
```

These methods of research may seem rather strange for those who are not used to processing and analyzing text, but you will soon understand that these methods of research are perfectly suited to the words and their meaning in relation to the text in which they are present.

Analyze the Frequency of Words

One of the simplest and most basic examples for the analysis of a text is to calculate the frequency of the words contained in it. This operation is so common that it has been incorporated into a single `nltk.FreqDist()` function to which the variable containing the word array is passed as an argument.

So to get a statistical distribution of all the words in the text, you enter a simple command.

```
fd = nltk.FreqDist(macbeth)
```

If you want to see the first ten most common words in the text, you can use the `most_common()` function.

```
fd.most_common(10)
Out [ ]:
[(',', 1962),
 ('.', 1235),
 ("'", 637),
 ('the', 531),
 (':', 477),
 ('and', 376),
 ('I', 333),
 ('of', 315),
 ('to', 311),
 ('?', 241)]
```

From the result obtained, you can see that the most common elements are punctuation, prepositions, and articles, and this applies to many languages, including English. Because these have little meaning during text analysis, it is often necessary to eliminate them. These are called *stopwords*.

Stopwords are words that have little meaning in the analysis and must be filtered. There is no general rule to determine whether a word is a stopword (to be deleted) or not. However, the NLTK library comes to the rescue by providing you with an array of pre-selected stopwords. To download these stopwords, you can use the `nltk.download()` command.

```
nltk.download('stopwords')
Out [ ]:
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\nelli\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
True
```

Once you have downloaded all the stopwords, you can select only those related to English, saving them in a variable `sw`.

```
sw = set(nltk.corpus.stopwords.words ('english'))
print(len(sw))
list(sw) [:10]
Out [ ]:
179
['through',
 'are',
 'than',
 'nor',
 'ain',
 "didn't",
 'didn',
 "shan't",
 'down',
 'our']
```

There are 179 stopwords in the English vocabulary according to NLTK. Now you can use these stopwords to filter the `macbeth` variable.

```
macbeth_filtered = [w for w in macbeth if w.lower() not in sw]
fd = nltk.FreqDist (macbeth_filtered)
fd.most_common(10)
Out [ ]:
[(',', 1962),
 ('.', 1235),
 ("'", 637),
 (':', 477),
 ('?', 241),
 ('Macb', 137),
 ('haue', 117),
 ('-', 100),
 ('Enter', 80),
 ('thou', 63)]
```

Now that the first ten most common words are returned, you can see that the stopwords have been eliminated, but the result is still not satisfactory. In fact, punctuation is still present in the words. To eliminate all punctuation, you can change the previous code by inserting in the filter an array of punctuation containing the punctuation symbols. This punctuation array can be obtained by importing the `string` function.

```
import string
punctuation = set (string.punctuation)
macbeth_filtered2 = [w.lower () for w in macbeth if w.lower () not in sw and w.lower () not
in punctuation]
```

Now you can recalculate the frequency distribution of words.

```
fd = nltk.FreqDist (macbeth_filtered2)
fd.most_common(10)
Out [ ]:
[('macb', 137),
 ('haue', 122),
 ('thou', 90),
 ('enter', 81),
 ('shall', 68),
 ('macbeth', 62),
 ('vpon', 62),
 ('thee', 61),
 ('macd', 58),
 ('vs', 57)]
```

Finally, the result is what you were looking for.

Select Words from Text

Another form of processing and data analysis is the process of selecting words contained in a body of text based on particular characteristics. For example, you might be interested in extracting words based on their length.

To get all the longest words, for example words that are longer than 12 characters, you enter the following command.

```
long_words = [w for w in macbeth if len(w)> 12]
```

All words longer than 12 characters have now been entered in the `long_words` variable. You can list them in alphabetical order by using the `sort()` function.

```
sorted(long_words)
Out [ ]:
['Assassination',
 'Chamberlaines',
 'Distinguishes',
 'Gallowgrosses',
 'Metaphysicall',
 'Northumberland',
 'Voluptuousnesse',
```

```
 'commendations',
 'multitudinous',
 'supernaturall',
 'vnaccompanied']
```

As you can see, there are 11 words that meet this criteria.

Another example is to look for all the words that contain a certain sequence of characters, such as `'ious'`. You only have to change the condition in the `for in` loop to get the desired selection.

```
ious_words = [w for w in macbeth if 'ious' in w]
ious_words = set(ious_words)
sorted(ious_words)
Out [ ]:
['Auaricious',
 'Gracious',
 'Industrious',
 'Iudicious',
 'Luxurious',
 'Malicious',
 'Obliuious',
 'Pious',
 'Rebellious',
 'compunctious',
 'furious',
 'gracious',
 'pernicious',
 'pernitious',
 'pious',
 'precious',
 'rebellious',
 'sacrilegious',
 'serious',
 'spacious',
 'tedious']
```

This example uses `sort()` to make a list casting, so that it did not contain duplicate words.

These two examples are just a starting point to show you the potential of this tool and the ease with which you can filter words.

Bigrams and Collocations

Another basic element of text analysis is to consider pairs of words (*bigrams*) instead of single words. The words “is” and “yellow” are for example a bigram, since their combination is possible and meaningful. So “is yellow” can be found in textual data. We all know that some of these bigrams are so common in our literature that they are almost always used together. Examples include “fast food,” “pay attention,” “good morning,” and so on. These bigrams are called *collocations*.

Textual analysis can also involve the search for any bigrams within the text under examination. To find them, simply use the `bigrams()` function. In order to exclude stopwords and punctuation from the bigrams, you must use the set of words already filtered, such as `macbeth_filtered2`.

```
bgrms = nltk.FreqDist(nltk.bigrams(macbeth_filtered2))
bgrms.most_common(15)
Out [ ]:
[(('enter', 'macbeth'), 16),
 (('exeunt', 'scena'), 15),
 (('thane', 'cawdor'), 13),
 (('knock', 'knock'), 10),
 (('st', 'thou'), 9),
 (('thou', 'art'), 9),
 (('lord', 'macb'), 9),
 (('haue', 'done'), 8),
 (('macb', 'haue'), 8),
 (('good', 'lord'), 8),
 (('let', 'vs'), 7),
 (('enter', 'lady'), 7),
 (('wee', 'l'), 7),
 (('would', 'st'), 6),
 (('macbeth', 'macb'), 6)]
```

By displaying the most common bigrams in the text, linguistic locations can be found.

In addition to the bigrams, there can also be placements based on trigrams, which are combinations of three words. In this case, the `trigrams()` function is used.

```
tgrms = nltk.FreqDist(nltk.trigrams (macbeth_filtered2))
tgrms.most_common(10)
Out [ ]:
[(('knock', 'knock', 'knock'), 6),
 (('enter', 'macbeth', 'macb'), 5),
 (('enter', 'three', 'witches'), 4),
 (('exeunt', 'scena', 'secunda'), 4),
 (('good', 'lord', 'macb'), 4),
 (('three', 'witches', '1'), 3),
 (('exeunt', 'scena', 'tertia'), 3),
 (('thunder', 'enter', 'three'), 3),
 (('exeunt', 'scena', 'quarta'), 3),
 (('scena', 'prima', 'enter'), 3)]
```

Preprocessing Steps

Text preprocessing is one of the most important and fundamental phases of text analysis. After collecting the text to be analyzed from various available sources, you will soon realize that in order to use the text in the various NLP techniques, it is necessary to clean it, transform it, and then prepare it specifically to be usable. This section looks at some of the more common preprocessing operations.

The *lower conversion* is perhaps the most frequent and common operation, not only in the preprocessing phase, but also in the following phases of the analysis. In fact, the text contains many words in which some characters are capitalized. Most parsing techniques require that all words be lowercased, so that "word" and "Word" are considered the same word.

In Python, the operation is very simple (there is no need to use the `nltk` library), since string variables have the `lower()` method that applies the conversion.

```
text = 'This is a Demo Sentence'
lower_text = text.lower()
lower_text
Out [ ]:
'this is a demo sentence'
```

Word tokenization is another very common operation in NLP. Text consists of words with spaces between them. Therefore, the operation of converting text into a list of words is fundamental in order to be able to process the text computationally. NLTK provides the `word_tokenize()` function for this purpose. But first you need to download the `'punkt'` resource from `nltk`.

```
nltk.download('punkt')

text = 'This is a Demo Sentence'
tokens = nltk.word_tokenize(text)
tokens
Out [ ]:
['This', 'is', 'a', 'Demo', 'Sentence']
```

Tokenization can also be performed at a higher level, by separating the sentences that make up the text and converting them into elements of a list, instead of single words. In this case, the `sent_tokenize()` function is used.

```
text = 'This is a Demo Sentence. This is another sentence'
tokens = nltk.sent_tokenize(text)
tokens
Out [ ]:
['This is a Demo Sentence.', 'This is another sentence']
```

Another common preprocessing operation is *punctuation mark removal*. Often you have to submit comments taken from social networks or product reviews on the web for analysis. Many of these texts are rich in punctuation marks, which compromise correct tokenization of the words contained in them. For this operation, NLTK provides a particular tokenizer object called `RegexpTokenizer`, which allows you to define the tokenization criteria through regular expressions. The following example sets `RegexpTokenizer` to remove all punctuation marks present in the text.

```
from nltk.tokenize import RegexpTokenizer

text = 'This% is a #!!@ Sentence full of punctuation marks :-) '
regexpt = RegexpTokenizer(r'[a-zA-Z0-9]+')
tokens = regexpt.tokenize(text)
tokens
Out [ ]:
['This', 'is', 'a', 'Sentence', 'full', 'of', 'punctuation', 'marks']
```

The *stopword removal* operation is instead more complex. In fact, stopwords are not particular characters that can be discarded using regular expressions, but are real words that "do not provide information to the text." These words are clearly related to each individual language, and they vary in each language. In English, words like "the," "a," "on," and "in" are basically stopwords. To remove them from the text being analyzed, you can operate as follows.

First, you load the stopwords from `nltk` and then import them into the code. You do the normal word tokenization and then later remove the English stopwords.

```
nltk.download('stopwords')

from nltk.corpus import stopwords

text = 'This is a Demo Sentence. This is another sentence'
eng_sw = stopwords.words('english')
tokens = nltk.word_tokenize(text)
clean_tokens = [word for word in tokens if word not in eng_sw]
clean_tokens
Out [ ]:
['This', 'Demo', 'Sentence', '.', 'This', 'another', 'sentence']
```

Another type of preprocessing operation involves linguistics. Operations such as *stemming* and *lemmatization* operate on individual words by evaluating their linguistic root (in the first case) and lemma (in the second case). Stemming then groups all words having the same root, considering them the single, same word. Lemmatization instead looks for all the inflected forms of a word or a verb and groups them under the same lemma, considering them all a single word.

For stemming, you therefore have all the roots of the words contained in the text in the tokens. You import a stemmer available in NLTK as `SnowballStemmer` and set it to English. Then a classic tokenization is performed on the words. Only at this point are they cleaned up by converting them into their linguistic roots.

```
from nltk.stem import SnowballStemmer

text = 'This operation operates for the operator curiosity. A decisive decision'
stemmer = SnowballStemmer('english')
tokens = nltk.word_tokenize(text)
stemmed_tokens = [stemmer.stem(word) for word in tokens]
print(stemmed_tokens)
Out [ ]:
['this', 'oper', 'oper', 'for', 'the', 'oper', 'curios', '.', 'a', 'decis', 'decis']
```

As far as lemmatization is concerned, the operation is very similar. But first you need to download from `nltk` two components, like `WordNet` and `Omw`. A classic word tokenization is performed on the text and then a lemmatizer is defined. At this point this is applied to the tokens to perform the lemmatization of the single words. All inflected forms are lumped together, including singular and plural words and verb conjugations.

```
nltk.download('omw-1.4')
nltk.download('wordnet')

from nltk.stem import WordNetLemmatizer

text = 'A verb: I split, it splits. Splitted verbs.'
tokens = nltk.word_tokenize(text)
```

```
lmtzr = WordNetLemmatizer()
lemma_tokens = [lmtzr.lemmatize(word) for word in tokens]
print(lemma_tokens)
Out [ ]:
['A', 'verb', ':', 'I', 'split', ',', 'it', 'split', '.', 'Splitted', 'verb', '.']
```

Use Text on the Network

So far you have seen a series of examples that use ordered and included text (called a corpus) within the NLTK library as gutenberg. But in reality, you will need to access the Internet to extract the text and collect it as a corpus to be used for analysis with NLTK.

In this section, you see how simple this kind of operation is. First, you need to import a library that allows you to connect to the contents of web pages. The urllib library is an excellent candidate for this purpose, as it allows you to download the text content from the Internet, including HTML pages.

So first you import the request() function, which specializes in this kind of operation, from the urllib library.

```
from urllib import request
```

Then you have to write the URL of the page that contains the text to be extracted. Still referring to the gutenberg project, you can choose, for example, a book written by Dostoevsky (www.gutenberg.org). On the site, there is text in different formats; this example uses the one in the raw format (.txt).

```
url = "http://www.gutenberg.org/files/2554/2554-0.txt"
response = request.urlopen(url)
raw = response.read().decode('utf8')
```

Within the raw text is all the textual content of the book, downloaded from the Internet. Always check the contents of what you downloaded. To do this, the first 75 characters are enough.

```
raw[:75]
Out [ ]:
'\ufeffThe Project Gutenberg EBook of Crime and Punishment, by Fyodor Dostoevsky\r'
```

As you can see, these characters correspond to the title of the text. You can see that there is also an error in the first word of the text. In fact there is the Unicode character BOM \ufeff. This happened because this example used the utf8 decoding system, which is valid in most cases, but not in this case. The most suitable system in this case is utf-8-sig. Replace the incorrect value with the correct one.

```
raw = response.read().decode('utf8-sig')
raw[:75]
Out [ ]:
'The Project Gutenberg EBook of Crime and Punishment, by Fyodor Dostoevsky\r\n'
```

To be able to work on it, you have to convert it into a corpus compatible with NLTK. To do this, enter the following conversion commands.

```
tokens = nltk.word_tokenize (raw)
webtext = nltk.Text (tokens)
```

These commands do nothing more than split the character text into tokens (that is, words) using the `nltk.word_tokenize()` function and then convert the tokens into a textual body suitable for NLTK using `nltk.Text()`.

You can see the title by entering this command:

```
webtext[:12]
Out [ ]:
['The',
 'Project',
 'Gutenberg',
 'EBook',
 'of',
 'Crime',
 'and',
 'Punishment',
 ',',
 'by',
 'Fyodor',
 'Dostoevsky']
```

Now you have a correct corpus on which to carry out your analysis.

Extract the Text from the HTML Pages

In the previous example, you created a NLTK corpus from text downloaded from the Internet. But most of the documentation on the Internet is in the form of HTML pages. In this section, you see how to extract text from HTML pages.

You always use the `request()` function of the `urllib` library to download the HTML content of a web page.

```
url = "https://news.bbc.co.uk/2/hi/health/2284783.stm"
html = request.urlopen(url).read().decode('utf8')
html[:120]
Out [ ]:
'<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/TR/REC-
html40/loose.dtd">\r\n<html>\r\n<hea'
```

Now, however, the conversion into NLTK corpus requires an additional library, `bs4 (BeautifulSoup)`, which provides you with suitable parsers that can recognize HTML tags and extract the text contained in them.

```
from bs4 import BeautifulSoup
raw = BeautifulSoup(html, "lxml").get_text()
tokens = nltk.word_tokenize(raw)
text = nltk.Text(tokens)
```

Now you also have a corpus in this case, even if you often have to perform more complex cleaning operations than the previous case to eliminate the words that do not interest you.

Sentiment Analysis

Sentiment analysis is a new field of research that has developed very recently in order to evaluate people's opinions about a particular topic. This discipline is based on different techniques that use text analysis and its field of work in the world of social media and forums (*opinion mining*).

Thanks to comments and reviews by users, sentiment analysis algorithms can evaluate the degree of appreciation or evaluation based on certain keywords. This degree of appreciation is called *opinion* and has three possible values: positive, neutral, or negative. The assessment of this opinion thus becomes a form of classification.

So many sentiment analysis techniques are actually classification algorithms, similar to those you saw in previous chapters covering machine learning and deep learning (see Chapters 8 and 9).

As an example to better understand this methodology, I reference a classification tutorial using the Naïve Bayes algorithm on the official website (`www.nltk.org/book/ch06.html`), where it is possible to find many other useful examples to better understand this library.

As a training set, this example uses another corpus present in NLTK, which is very useful for these types of classification problems: `movie_reviews`. This corpus contains numerous film reviews in which there is text of a discrete length together with another field that specifies whether the critique is positive or negative. Therefore, it serves as great learning material.

The purpose of this tutorial is to find the words that recur most in negative documents, or words that recur more in positive ones, so as to focus on the keywords related to an opinion. This evaluation is carried out through a Naïve Bayes classification integrated into NLTK.

First of all, the corpus called `movie_reviews` is important.

```
nltk.download('movie_reviews')
Out [ ]:
[nltk_data] Downloading package movie_reviews to
[nltk_data]     C:\Users\nelli\AppData\Roaming\nltk_data...
[nltk_data]   Package movie_reviews is already up-to-date!
True
```

Then you build the training set from the corpus obtained, creating an array of element pairs called `documents`. This array contains in the first field the text of the single review, and in the second field the negative or positive evaluation. At the end, you mix all the elements of the array in random order.

```
import random
reviews = nltk.corpus.movie_reviews
documents = [(list(reviews.words(fileid)), category)
                for category in reviews.categories()
          for fileid in reviews.fileids(category)]
random.shuffle(documents)
```

To better understand this, take a look at the contents of the documents in detail. The first element contains two fields; the first is the review containing all the words used.

```
first_review = ' '.join(documents[0][0])
print(first_review)
Out [ ]:
topless women talk about their lives falls into that category that i mentioned in the
devil ' s advocate : movies that have a brilliant beginning but don ' t know how to end .
it begins by introducing us to a selection of characters who all know each other . there
is liz , who oversleeps and so is running late for her appointment , prue who is getting
married ,...
```

The second field instead contains the evaluation of the review:

```
documents[0][1]
Out [ ]:
'neg'
```

But the training set is not yet ready; in fact you have to create a frequency distribution of all the words in the corpus. This distribution is converted into a casting list with the `list()` function.

```
all_words = nltk.FreqDist(w.lower() for w in reviews.words())
word_features = list(all_words)
```

Then the next step is to define a function for the calculation of the features, that is, words that are important enough to establish the opinion of a review.

```
def document_features(document, word_features):
     document_words = set(document)
     features = {}
     for word in word_features:
         features ['{}'.format(word)] = (word in document_words)
     return features
```

Once you have defined the `document_features()` function, you can create feature sets from documents.

```
featuresets = [(document_features(d,word_features), c) for (d,c) in documents]
```

The aim is to create a set of all the words contained in the whole movie corpus, analyze whether they are present (True or False) in each single review, and see how much they contribute to the positive or negative judgment of the review. The more often a word is present in the negative reviews and the less often it's present in the positive ones, the more it's evaluated as a "bad" word. The opposite is true for a "good" word evaluation.

To determine how to subdivide this feature set for the training set and the testing set, you must first determine how many elements it contains.

```
len (featuresets)
Out [ ]:
2000
```

To evaluate the accuracy of the model, you use the first 1,500 elements of the set for the training set, and the last 500 items for the testing set.

```
train_set, test_set = featuresets[1500:], featuresets[:500]
```

Finally, you apply the Naïve Bayes classifier provided by the NLTK library to classify this problem. Then you calculate its accuracy, submitting the test set to the model.

```
classifier = nltk.NaiveBayesClassifier.train(train_set)
print (nltk.classify.accuracy(classifier, test_set))
Out [ ]:
0.85
```

The accuracy is not as high as in the examples from the previous chapters, but you are working with words contained in text, and therefore it is very difficult to create accurate models relative to numerical problems.

Now that you have completed the analysis, you can see which words have the most weight in evaluating the negative or positive opinion of a review.

```
classifier.show_most_informative_features(10)
Out [ ]:
Most Informative Features
              compelling = True              pos : neg    =     11.9 : 1.0
             outstanding = True              pos : neg    =     11.2 : 1.0
                    lame = True              neg : pos    =     10.2 : 1.0
           extraordinary = True              pos : neg    =      9.7 : 1.0
                   lucas = True              pos : neg    =      9.7 : 1.0
                    bore = True              neg : pos    =      8.3 : 1.0
                   catch = True              neg : pos    =      8.3 : 1.0
                 journey = True              pos : neg    =      8.3 : 1.0
             magnificent = True              pos : neg    =      8.3 : 1.0
                 triumph = True              pos : neg    =      8.3 : 1.0
```

Looking at the results, you will not be surprised to find that the word “badly” is a bad opinion word and that “finest” is a good opinion word. The interesting thing here is that “julie” is a bad opinion word.

Conclusions

In this chapter, you took a small glimpse of the text analysis world. In fact, there are many other techniques and examples that could be discussed. However, at the end of this chapter, you should be familiar with this branch of analysis and especially have begun to learn about the NLTK (Natural Language Toolkit) library, a powerful tool for text analysis.