CHAPTER 7

Semantic and Sentiment Analysis

Natural language understanding has gained significant importance in the last decade with the advent of machine learning (ML) and further advances like *deep learning* and artificial intelligence. Computers and other machines can be programmed to learn things and perform specific operations. The key limitation is their inability to perceive, understand, and comprehend things like humans do. With the resurgence in popularity of neural networks and advances made in computer architecture, we now have deep learning and artificial intelligence evolving rapidly to make some efforts into trying to engineer machines into learning, perceiving, understanding, and performing actions on their own. You may have seen or heard several of these efforts, such as self-driving cars, computers beating experienced players in games like chess and Go, and the proliferation of chatbots on the Internet.

In Chapters 4–6, we have looked at various computational, language processing, and ML techniques to classify, cluster, and summarize text. Back in Chapter 3 we developed certain methods and programs to analyze and understand text syntax and structure. This chapter will deal with methods that try to answer the question *Can we analyze and understand the meaning and sentiment behind a body of text?*

Natural Language Processing (NLP) has a wide variety of applications that try to use natural language understanding to infer the meaning and context behind text and use it to solve various problems. We discussed several of these applications briefly in Chapter 1. To refresh your memory, the following applications require extensive understanding of text from the semantic perspective:

- Question Answering Systems
- Contextual recognition
- Speech recognition (for some applications)

Text semantics specifically deals with understanding the meaning of text or language. When combined into sentences, words have lexical relations and contextual relations between them lead to various types of relationships and hierarchies, and semantics sits at the heart of all this in trying to analyze and understand these relationships and infer meaning from them. We will be exploring various types of semantic relationships in natural language and look at some NLP-based techniques for inferring and extracting meaningful

D. Sarkar, *Text Analytics with Python*, DOI 10.1007/978-1-4842-2388-8_7

semantic information from text. Semantics is purely concerned with context and meaning, and the structure or format of text holds little significance here. But sometimes even the syntax or arrangement of words helps us in inferring the context of words and helps us differentiate things like *lead* as a metal from *lead* as in the lead of a movie.

Sentiment analysis is perhaps the most popular application of text analytics, with a vast number of tutorials, web sites, and applications that focus on analyzing sentiment of various text resources ranging from corporate surveys to movie reviews. The key aspect of sentiment analysis is to analyze a body of text for understanding the opinion expressed by it and other factors like mood and modality. Usually sentiment analysis works best on text that has a subjective context than on that with only an objective context. This is because when a body of text has an objective context or perspective to it, the text usually depicts some normal statements or facts without expressing any emotion, feelings, or mood. Subjective text contains text that is usually expressed by a human having typical moods, emotions, and feelings. Sentiment analysis is widely used, especially as a part of social media analysis for any domain, be it a business, a recent movie, or a product launch, to understand its reception by the people and what they think of it based on their opinions or, you guessed it, sentiment.

In this chapter, we will be covering several aspects from both semantic and sentiment analysis for textual data. We will start with exploring WordNet, a lexical database, and introduce a new concept called *synsets*. We will also explore various semantic relationships and representations in natural language and we will cover techniques such as *word sense disambiguation* and *named entity recognition*. In sentiment analysis, we will be looking at how to use supervised ML techniques to analyze sentiment and also at several unsupervised lexical techniques with more detailed insights into natural language sentiment, mood, and modality.

Semantic Analysis

We have seen how terms or words get grouped into phrases that further form clauses and finally sentences. Chapter 3 showed various structural components in natural language, including parts of speech (POS), chunking, and grammars. All these concepts fall under the syntactic and structural analysis of text data. Whereas we do explore relationships of words, phrases, and clauses, these are purely based on their position, syntax, and structure. Semantic analysis is more about understanding the actual context and meaning behind words in text and how they relate to other words to convey some information as a whole. As mentioned in Chapter 1, the definition of semantics itself is the study of meaning, and linguistic semantics is a complete branch under linguistics that deals with the study of meaning in natural language, including exploring various relationships between words, phrases and symbols. Besides this, there are also various ways to represent semantics associated with statements and propositions. We will be broadly covering the following topics under semantic analysis:

- Exploring WordNet and synsets
- Analyzing lexical semantic relations
- Word sense disambiguation
- Named entity recognition
- Analyzing semantic representations

The main objective of these topics is to give you a clear understanding of the resources you can leverage for semantic analysis as well as how to use these resources. We will explore various concepts related to semantic analysis, which was covered in Chapter 1, with actual examples. You can refresh your memory by revisiting the "Language Semantics" section in Chapter 1. Without any further delay, let's get started!

Exploring WordNet

WordNet is a huge lexical database for the English Language. The database is a part of Princeton University, and you can read more about it at `https://wordnet.princeton.edu`. It was originally created in around 1985, in Princeton University's Cognitive Science Laboratory under the direction of Professor G. A. Miller. This lexical database consists of nouns, adjective, verbs, and adverbs, and related lexical terms are grouped together based on some common concepts into sets, known as *cognitive synonym sets* or *synsets*. Each synset expresses a unique, distinct concept. At a high level, WordNet can be compared to a thesaurus or a dictionary that provides words and their synonyms. On a lower level, it is much more than that, with synsets and their corresponding terms having detailed relationships and hierarchies based on their semantic meaning and similar concepts. WordNet is used extensively as a lexical database, in text analytics, NLP, and artificial intelligence (AI)-based applications.

The WordNet database consists of over 155,000 words, represented in more than 117,000 synsets, and contains over 206,000 word-sense pairs. The database is roughly 12 MB in size and can be accessed through various interfaces and APIs. The official web site has a web application interface for accessing various details related to words, synsets, and concepts related to the entered word. You can access it at `http://wordnetweb.princeton.edu/perl/webwn` or download it from `https://wordnet.princeton.edu/wordnet/download/`. The download contains various packages, files, and tools related to WordNet. We will be accessing WordNet programmatically using the interface provided by the `nltk` package. We will start by exploring synsets and then various semantic relationships using synsets.

Understanding Synsets

We will start exploring WordNet by looking at synsets since they are perhaps one of the most important concepts and structures that tie everything together. In general, based on concepts from NLP and information retrieval, a synset is a collection or set of data entities that are considered to be semantically similar. This doesn't mean that they will be exactly the same, but they will be centered on similar context and concepts. Specifically in the context of WordNet, a synset is a set or collection of synonyms that are interchangeable and revolve around a specific concept. Synsets not only consist of simple words, but also collocations. *Polysemous* word forms (words that sound and look the same but have different but relatable meanings) are assigned to different synsets based on their meaning. Synsets are connected to other synsets using semantic relations, which we shall explore in a future section. Typically each synset has the term, a definition explaining the meaning of the term, and some optional examples and related lemmas (collection of synonyms) to the term. Some terms may have multiple synsets associated with them, where each synset has a particular context.

Let's look at a real example by using `nltk`'s WordNet interface to explore synsets associated with the term, `'fruit'`. We can do this using the following code snippet:

```
from nltk.corpus import wordnet as wn
import pandas as pd

term = 'fruit'
synsets = wn.synsets(term)
# display total synsets
In [75]: print 'Total Synsets:', len(synsets)
Total Synsets: 5
```

We can see that there are a total of five synsets associated with the term `'fruit'`. What can these synsets indicate? We can dig deeper into each synset and its components using the following code snippet:

```
In [76]: for synset in synsets:
    ...:     print 'Synset:', synset
    ...:     print 'Part of speech:', synset.lexname()
    ...:     print 'Definition:', synset.definition()
    ...:     print 'Lemmas:', synset.lemma_names()
    ...:     print 'Examples:', synset.examples()
    ...:     print
    ...:
    ...:
Synset: Synset('fruit.n.01')
Part of speech: noun.plant
Definition: the ripened reproductive body of a seed plant
Lemmas: [u'fruit']
Examples: []

Synset: Synset('yield.n.03')
Part of speech: noun.artifact
Definition: an amount of a product
Lemmas: [u'yield', u'fruit']
Examples: []

Synset: Synset('fruit.n.03')
Part of speech: noun.event
Definition: the consequence of some effort or action
Lemmas: [u'fruit']
Examples: [u'he lived long enough to see the fruit of his policies']

Synset: Synset('fruit.v.01')
Part of speech: verb.creation
Definition: cause to bear fruit
Lemmas: [u'fruit']
Examples: []
```

```
Synset: Synset('fruit.v.02')
Part of speech: verb.creation
Definition: bear fruit
Lemmas: [u'fruit']
Examples: [u'the trees fruited early this year']
```

The preceding output shows us details pertaining to each synset associated with the term `'fruit'`, and the definitions give us the sense of each synset and the lemma associated with it. The part of speech for each synset is also mentioned, which includes nouns and verbs. Some examples are also depicted in the preceding output that show how the term is used in actual sentences. Now that we understand synsets better, let's start exploring various semantic relationships as mentioned.

Analyzing Lexical Semantic Relations

Text semantics refers to the study of meaning and context. Synsets give a nice abstraction over various terms and provide useful information like definition, examples, POS, and lemmas. But can we explore semantic relationships among entities using synsets? The answer is definitely yes. We will be talking about many of the concepts related to semantic relations (covered in detail in the "Lexical Semantic Relations" subsection under the "Language Semantics" section in Chapter 1. It would be useful for you to review that section to better understand each of the concepts when we illustrate them with real-world examples here. We will be using `nltk`'s `wordnet` resource here, but you can use the same WordNet resource from the `pattern` package, which includes an interface similar to `nltk`.

Entailments

The term *entailment* usually refers to some event or action that logically involves or is associated with some other action or event that has taken place or will take place. Ideally this applies very well to verbs indicating some specific action. The following snippet shows how to get entailments:

```
# entailments
In [80]: for action in ['walk', 'eat', 'digest']:
    ...:     action_syn = wn.synsets(action, pos='v')[0]
    ...:     print action_syn, '-- entails -->', action_syn.entailments()
Synset('walk.v.01') -- entails --> [Synset('step.v.01')]
Synset('eat.v.01') -- entails --> [Synset('chew.v.01'),
Synset('swallow.v.01')]
Synset('digest.v.01') -- entails --> [Synset('consume.v.02')]
```

You can see how related synsets depict the concept of entailment in that output. Related actions are depicted in entailment, where actions like *walking* involve or entail *stepping*, and *eating* entails *chewing* and *swallowing*.

Homonyms and Homographs

On a high level, *homonyms* refer to words or terms having the same written form or pronunciation but different meanings. Homonyms are a superset of homographs, which are words with same spelling but may have different pronunciation and meaning. The following code snippet shows how we can get homonyms/homographs:

```
In [81]: for synset in wn.synsets('bank'):
    ...:     print synset.name(),'-',synset.definition()
    ...:
    ...:
bank.n.01 - sloping land (especially the slope beside a body of water)
depository_financial_institution.n.01 - a financial institution that accepts
deposits and channels the money into lending activities
bank.n.03 - a long ridge or pile
bank.n.04 - an arrangement of similar objects in a row or in tiers
...
...
deposit.v.02 - put into a bank account
bank.v.07 - cover with ashes so to control the rate of burning
trust.v.01 - have confidence or faith in
```

The preceding output shows a part of the result obtained for the various homographs for the term `'bank'`. You can see that there are various different meanings associated with the word `'bank'`, which is the core intuition behind homographs.

Synonyms and Antonyms

Synonyms are words having similar meaning and context, and *antonyms* are words having opposite or contrasting meaning, as you may know already. The following snippet depicts synonyms and antonyms:

```
In [82]: term = 'large'
    ...: synsets = wn.synsets(term)
    ...: adj_large = synsets[1]
    ...: adj_large = adj_large.lemmas()[0]
    ...: adj_large_synonym = adj_large.synset()
    ...: adj_large_antonym = adj_large.antonyms()[0].synset()
    ...: # print synonym and antonym
    ...: print 'Synonym:', adj_large_synonym.name()
    ...: print 'Definition:', adj_large_synonym.definition()
    ...: print 'Antonym:', adj_large_antonym.name()
    ...: print 'Definition:', adj_large_antonym.definition()
Synonym: large.a.01
Definition: above average in size or number or quantity or magnitude or
extent
Antonym: small.a.01
```

```
Definition: limited or below average in number or quantity or magnitude or
extent

In [83]: term = 'rich'
    ...: synsets = wn.synsets(term)[:3]
    ...: # print synonym and antonym for different synsets
    ...: for synset in synsets:
    ...:     rich = synset.lemmas()[0]
    ...:     rich_synonym = rich.synset()
    ...:     rich_antonym = rich.antonyms()[0].synset()
    ...:     print 'Synonym:', rich_synonym.name()
    ...:     print 'Definition:', rich_synonym.definition()
    ...:     print 'Antonym:', rich_antonym.name()
    ...:     print 'Definition:', rich_antonym.definition()
Synonym: rich_people.n.01
Definition: people who have possessions and wealth (considered as a group)
Antonym: poor_people.n.01
Definition: people without possessions or wealth (considered as a group)

Synonym: rich.a.01
Definition: possessing material wealth
Antonym: poor.a.02
Definition: having little money or few possessions

Synonym: rich.a.02
Definition: having an abundant supply of desirable qualities or substances
(especially natural resources)
Antonym: poor.a.04
Definition: lacking in specific resources, qualities or substances
```

The preceding outputs show sample synonyms and antonyms for the term `'large'` and the term `'rich'`. Additionally, we explore several synsets associated with the term or concept `'rich'`, which rightly give us distinct synonyms and their corresponding antonyms.

Hyponyms and Hypernyms

Synsets represent terms with unique semantics and concepts and are linked or related to each other based on some similarity and context. Several of these synsets represent abstract and generic concepts also besides concrete entities. Usually they are interlinked together in the form of a hierarchical structure representing *is-a* relationships. Hyponyms and hypernyms help us explore related concepts by navigating through this hierarchy. To be more specific, *hyponyms* refer to entities or concepts that are a subclass of a higher order concept or entity and have very specific sense or context compared to its superclass. The following snippet shows the hyponyms for the entity `'tree'`:

```
term = 'tree'
synsets = wn.synsets(term)
tree = synsets[0]
# print the entity and its meaning
In [86]: print 'Name:', tree.name()
    ...: print 'Definition:', tree.definition()
Name: tree.n.01
Definition: a tall perennial woody plant having a main trunk and branches
forming a distinct elevated crown; includes both gymnosperms and angiosperms
# print total hyponyms and some sample hyponyms for 'tree'
In [87]: hyponyms = tree.hyponyms()
    ...: print 'Total Hyponyms:', len(hyponyms)
    ...: print 'Sample Hyponyms'
    ...: for hyponym in hyponyms[:10]:
    ...:     print hyponym.name(), '-', hyponym.definition()

Total Hyponyms: 180
Sample Hyponyms
aalii.n.01 - a small Hawaiian tree with hard dark wood
acacia.n.01 - any of various spiny trees or shrubs of the genus Acacia
african_walnut.n.01 - tropical African timber tree with wood that resembles
mahogany
albizzia.n.01 - any of numerous trees of the genus Albizia
alder.n.02 - north temperate shrubs or trees having toothed leaves and
conelike fruit; bark is used in tanning and dyeing and the wood is rot-
resistant
angelim.n.01 - any of several tropical American trees of the genus Andira
angiospermous_tree.n.01 - any tree having seeds and ovules contained in the
ovary
anise_tree.n.01 - any of several evergreen shrubs and small trees of the
genus Illicium
arbor.n.01 - tree (as opposed to shrub)
aroeira_blanca.n.01 - small resinous tree or shrub of Brazil
```

The preceding output tells us that there are a total of 180 hyponyms for `'tree'`, and we see some of the sample hyponyms and their definitions. We can see that each hyponym is a specific type of tree, as expected. Hyponyms are entities or concepts that act as the superclass to hyponyms and have a more generic sense or context. The following snippet shows the immediate superclass hyponym for `'tree'`:

```
In [88]: hypernyms = tree.hypernyms()
    ...: print hypernyms
[Synset('woody_plant.n.01')]
```

You can even navigate up the entire entity/concept hierarchy depicting all the hyponyms or parent classes for `'tree'` using the following code snippet:

```
# get total hierarchy pathways for 'tree'
In [91]: hypernym_paths = tree.hypernym_paths()
```

```
    ...: print 'Total Hypernym paths:', len(hypernym_paths)
Total Hypernym paths: 1

# print the entire hypernym hierarchy
In [92]: print 'Hypernym Hierarchy'
    ...: print ' -> '.join(synset.name() for synset in hypernym_paths[0])
Hypernym Hierarchy
entity.n.01 -> physical_entity.n.01 -> object.n.01 -> whole.n.02 -> living_
thing.n.01 -> organism.n.01 -> plant.n.02 -> vascular_plant.n.01 -> woody_
plant.n.01 -> tree.n.01
```

From the preceding output, you can see that `'entity'` is the most generic concept in which `'tree'` is present, and the complete hypernym hierarchy showing the corresponding hypernym or superclass at each level is shown. As you navigate further down, you get into more specific concepts/entities, and if you go in the reverse direction you will get into more generic concepts/entities.

Holonyms and Meronyms

Holonyms are entities that contain a specific entity of our interest. Basically *holonym* refers to the relationship between a term or entity that denotes the whole and a term denoting a specific part of the whole. The following snippet shows the holonyms for `'tree'`:

```
In [94]: member_holonyms = tree.member_holonyms()
    ...: print 'Total Member Holonyms:', len(member_holonyms)
    ...: print 'Member Holonyms for [tree]:-'
    ...: for holonym in member_holonyms:
    ...:     print holonym.name(), '-', holonym.definition()
Total Member Holonyms: 1
Member Holonyms for [tree]:-
forest.n.01 - the trees and other plants in a large densely wooded area
```

From the output, we can see that `'forest'` is a holonym for `'tree'`, which is semantically correct because, of course, a forest is a collection of trees. *Meronyms* are semantic relationships that relate a term or entity as a part or constituent of another term or entity. The following snippet depicts different types of meronyms for `'tree'`:

```
# part based meronyms for tree
In [95]: part_meronyms = tree.part_meronyms()
    ...: print 'Total Part Meronyms:', len(part_meronyms)
    ...: print 'Part Meronyms for [tree]:-'
    ...: for meronym in part_meronyms:
    ...:     print meronym.name(), '-', meronym.definition()
Total Part Meronyms: 5
Part Meronyms for [tree]:-
burl.n.02 - a large rounded outgrowth on the trunk or branch of a tree
crown.n.07 - the upper branches and leaves of a tree or other plant
```

```
limb.n.02 - any of the main branches arising from the trunk or a bough of a
tree
stump.n.01 - the base part of a tree that remains standing after the tree
has been felled
trunk.n.01 - the main stem of a tree; usually covered with bark; the bole is
usually the part that is commercially useful for lumber

# substance based meronyms for tree
In [96]: substance_meronyms = tree.substance_meronyms()
    ...: print 'Total Substance Meronyms:', len(substance_meronyms)
    ...: print 'Substance Meronyms for [tree]:-'
    ...: for meronym in substance_meronyms:
    ...:     print meronym.name(), '-', meronym.definition()
Total Substance Meronyms: 2
Substance Meronyms for [tree]:-
heartwood.n.01 - the older inactive central wood of a tree or woody plant;
usually darker and denser than the surrounding sapwood
sapwood.n.01 - newly formed outer wood lying between the cambium and the
heartwood of a tree or woody plant; usually light colored; active in water
conduction
```

The preceding output shows various meronyms that include various constituents of trees like *stump* and *trunk* and also various derived substances from trees like *heartwood* and *sapwood*.

Semantic Relationships and Similarity

In the previous sections, we have looked at various concepts related to lexical semantic relationships. We will now look at ways to connect similar entities based on their semantic relationships and also measure semantic similarity between them. Semantic similarity is different from the conventional similarity metrics discussed in Chapter 6. We will use some sample synsets related to living entities as shown in the following snippet for our analysis:

```
tree = wn.synset('tree.n.01')
lion = wn.synset('lion.n.01')
tiger = wn.synset('tiger.n.02')
cat = wn.synset('cat.n.01')
dog = wn.synset('dog.n.01')
# create entities and extract names and definitions
entities = [tree, lion, tiger, cat, dog]
entity_names = [entity.name().split('.')[0] for entity in entities]
entity_definitions = [entity.definition() for entity in entities]

# print entities and their definitions
In [99]: for entity, definition in zip(entity_names, entity_definitions):
    ...:     print entity, '-', definition
```

```
tree - a tall perennial woody plant having a main trunk and branches forming
a distinct elevated crown; includes both gymnosperms and angiosperms
lion - large gregarious predatory feline of Africa and India having a tawny
coat with a shaggy mane in the male
tiger - large feline of forests in most of Asia having a tawny coat with
black stripes; endangered
cat - feline mammal usually having thick soft fur and no ability to roar:
domestic cats; wildcats
dog - a member of the genus Canis (probably descended from the common wolf)
that has been domesticated by man since prehistoric times; occurs in many
breeds
```

Now that we know our entities a bit better from these definitions explaining them, we will try to correlate the entities based on common hypernyms. For each pair of entities, we will try to find the lowest common hypernym in the relationship hierarchy tree. Correlated entities are expected to have very specific hypernyms, and unrelated entities should have very abstract or generic hypernyms. The following code snippet illustrates:

```
common_hypernyms = []
for entity in entities:
    # get pairwise lowest common hypernyms
    common_hypernyms.append([entity.lowest_common_hypernyms(compared_entity)[0]
                                        .name().split('.')[0]
                             for compared_entity in entities])
# build pairwise lower common hypernym matrix
common_hypernym_frame = pd.DataFrame(common_hypernyms,
                                     index=entity_names,
                                     columns=entity_names)
# print the matrix
In [101]: print common_hypernym_frame
     ...:
           tree       lion      tiger        cat        dog
tree       tree   organism   organism   organism   organism
lion   organism       lion    big_cat     feline  carnivore
tiger  organism    big_cat      tiger     feline  carnivore
cat    organism     feline     feline        cat  carnivore
dog    organism  carnivore  carnivore  carnivore        dog
```

Ignoring the main diagonal of the matrix, for each pair of entities, we can see their lowest common hypernym which depicts the nature of relationship between them. *Trees* are unrelated to the other animals except that they are all living organisms. Hence we get the `'organism'` relationship amongst them. *Cats* are related to *lions* and *tigers* with respect to being feline creatures, and we can see the same in the preceding output. *Tigers* and *lions* are connected to each other with the `'big cat'` relationship. Finally, we can see *dogs* having the relationship of `'carnivore'` with the other animals since they all typically eat meat.

We can also measure the semantic similarity between these entities using various semantic concepts. We will use `'path similarity'`, which returns a value between `[0, 1]` based on the shortest path connecting two terms based on their hypernym/hyponym based taxonomy. The following snippet shows us how to generate this similarity matrix:

```
similarities = []
for entity in entities:
    # get pairwise similarities
    similarities.append([round(entity.path_similarity(compared_entity), 2)
                         for compared_entity in entities])
# build pairwise similarity matrix
similarity_frame = pd.DataFrame(similarities,
                                index=entity_names,
                                columns=entity_names)
# print the matrix
print similarity_frame

       tree  lion  tiger   cat   dog
tree   1.00  0.07   0.07  0.08  0.13
lion   0.07  1.00   0.33  0.25  0.17
tiger  0.07  0.33   1.00  0.25  0.17
cat    0.08  0.25   0.25  1.00  0.20
dog    0.13  0.17   0.17  0.20  1.00
```

From the preceding output, as expected, *lion* and *tiger* are the most similar with a value of 0.33, followed by their semantic similarity with *cat* having a value of 0.25. And *tree* has the lowest semantic similarity values when compared with other animals.

This concludes our discussion on analyzing lexical semantic relations. I encourage you to try exploring more concepts with different examples by leveraging WordNet.

Word Sense Disambiguation

In the previous section, we looked at homographs and homonyms, which are basically words that look or sound similar but have very different meanings. This meaning is contextual based on how it has been used and also depends on the word semantics, also called *word sense*. Identifying the correct sense or semantics of a word based on its usage is called *word sense disambiguation* with the assumption that the word has multiple meanings based on its context. This is a very popular problem in NLP and is used in various applications, such as improving the relevance of search engine results, coherence, and so on.

There are various ways to solve this problem, including lexical and dictionary-based methods and supervised and unsupervised ML methods. Covering everything would be out of the current scope, so I will be showing word sense disambiguation using the Lesk algorithm, a classic algorithm invented by M. E. Lesk in 1986. The basic principle behind this algorithm is to leverage dictionary or vocabulary definitions for a word we want to disambiguate in a body of text and compare the words in these definitions with a section of text surrounding our word of interest. We will be using the WordNet definitions for words instead of a dictionary. The main objective for us would be to return the synset with the maximum number of overlapping words or terms between the context sentence and the different definitions from each synset for the word we target for disambiguation. The following snippet leverages `nltk` to depict how to use word sense disambiguation for various examples:

```
from nltk.wsd import lesk
from nltk import word_tokenize

# sample text and word to disambiguate
samples = [('The fruits on that plant have ripened', 'n'),
           ('He finally reaped the fruit of his hard work as he won the
           race', 'n')]
word = 'fruit'
# perform word sense disambiguation
In [106]: for sentence, pos_tag in samples:
     ...:     word_syn = lesk(word_tokenize(sentence.lower()), word, pos_tag)
     ...:     print 'Sentence:', sentence
     ...:     print 'Word synset:', word_syn
     ...:     print 'Corresponding definition:', word_syn.definition()
     ...:     print
Sentence: The fruits on that plant have ripened
Word synset: Synset('fruit.n.01')
Corresponding definition: the ripened reproductive body of a seed plant

Sentence: He finally reaped the fruit of his hard work as he won the race
Word synset: Synset('fruit.n.03')
Corresponding definition: the consequence of some effort or action

# sample text and word to disambiguate
samples = [('Lead is a very soft, malleable metal', 'n'),
           ('John is the actor who plays the lead in that movie', 'n'),
           ('This road leads to nowhere', 'v')]
word = 'lead'
# perform word sense disambiguation
In [108]: for sentence, pos_tag in samples:
     ...:     word_syn = lesk(word_tokenize(sentence.lower()), word,
              pos_tag)
     ...:     print 'Sentence:', sentence
     ...:     print 'Word synset:', word_syn
     ...:     print 'Corresponding definition:', word_syn.definition()
     ...:     print
Sentence: Lead is a very soft, malleable metal
Word synset: Synset('lead.n.02')
Corresponding definition: a soft heavy toxic malleable metallic element;
bluish white when freshly cut but tarnishes readily to dull grey

Sentence: John is the actor who plays the lead in that movie
Word synset: Synset('star.n.04')
Corresponding definition: an actor who plays a principal role

Sentence: This road leads to nowhere
Word synset: Synset('run.v.23')
Corresponding definition: cause something to pass or lead somewhere
```

We try to disambiguate two words, `'fruit'` and `'lead'` in various text documents in the preceding examples. You can see how we use the Lesk algorithm to get the correct word sense for the word we are disambiguating based on its usage and context in each document. This tells you how *fruit* can mean both an entity that is consumed as well as some consequence one faces on applying efforts. We also see how *lead* can mean the soft metal, causing something/someone to go somewhere, or even an actor who plays the main role in a play or movie.

Named Entity Recognition

In any text document, there are particular terms that represent entities that are more informative and have a unique context compared to the rest of the text. These entities are known as *named entities*, which more specifically refers to terms that represent real-world objects like people, places, organizations, and so on, which are usually denoted by proper names. We can find these typically by looking at the noun phrases in text documents. *Named entity recognition*, also known as *entity chunking/extraction*, is a popular technique used in information extraction to identify and segment named entities and classify or categorize them under various predefined classes. Some of these classes that are used most frequently are shown in Figure 7-1 (courtesy of `nltk` and The Stanford NLP group).

Named Entity Type	Examples
PERSON	President Obama, Franz Beckenbauer
ORGANIZATION	WHO, ISRO, FC Bayern
LOCATION	Germany, India, USA, Mt. Everest
DATE	December, 2016-12-25
TIME	12:30:00 AM, one thirty pm
MONEY	Twenty dollars, Rs. 50, 100 GBP
PERCENT	20%, forty five percent
FACILITY	Stonehenge, Taj Mahal, Washington Monument
GPE	Asia, Europe, Germany, North America

Figure 7-1. *Common named entities with examples*

There is some overlap between `GPE` and `LOCATION`. The `GPE` entities are usually more generic and represent geo-political entities like cities, states, countries, and continents. `LOCATION` can also refer to these entities (it varies across different NER systems) along with very specific locations like a mountain, river, or hill-station. `FACILITY` on the other hand refers to popular monuments or artifacts that are usually man-made. The remaining categories are pretty self-explanatory from their names and the examples depicted in Figure 7-1.

The Bundesliga is perhaps the most popular top-level professional association football league in Germany, and FC Bayern Munchen is one of the most popular clubs in this league with a global presence. We will now take a sample description of this club

from Wikipedia and try to extract named entities from it. We will reuse our normalization module (accessible as `normalization.py` in the code files) from the last chapter in this section to parse the document to remove unnecessary new lines. We will start by leveraging `nltk`'s Named Entity Chunker:

```
# sample document
text = """
Bayern Munich, or FC Bayern, is a German sports club based in Munich,
Bavaria, Germany. It is best known for its professional football team,
which plays in the Bundesliga, the top tier of the German football
league system, and is the most successful club in German football
history, having won a record 26 national titles and 18 national cups.
FC Bayern was founded in 1900 by eleven football players led by Franz John.
Although Bayern won its first national championship in 1932, the club
was not selected for the Bundesliga at its inception in 1963. The club
had its period of greatest success in the middle of the 1970s when,
under the captaincy of Franz Beckenbauer, it won the European Cup three
times in a row (1974-76). Overall, Bayern has reached ten UEFA Champions
League finals, most recently winning their fifth title in 2013 as part
of a continental treble.
"""

import nltk
from normalization import parse_document
import pandas as pd

# tokenize sentences
sentences = parse_document(text)
tokenized_sentences = [nltk.word_tokenize(sentence) for sentence in
sentences]

# tag sentences and use nltk's Named Entity Chunker
tagged_sentences = [nltk.pos_tag(sentence) for sentence in tokenized_
sentences]
ne_chunked_sents = [nltk.ne_chunk(tagged) for tagged in tagged_sentences]

# extract all named entities
named_entities = []
for ne_tagged_sentence in ne_chunked_sents:
    for tagged_tree in ne_tagged_sentence:
        # extract only chunks having NE labels
        if hasattr(tagged_tree, 'label'):
                entity_name = ' '.join(c[0] for c in tagged_tree.leaves()) #
                get NE name
                entity_type = tagged_tree.label() # get NE category
                named_entities.append((entity_name, entity_type))
# get unique named entities
named_entities = list(set(named_entities))
```

```
# store named entities in a data frame
entity_frame = pd.DataFrame(named_entities,
                            columns=['Entity Name', 'Entity Type'])
# display results
In [116]: print entity_frame
          Entity Name   Entity Type
0               Bayern        PERSON
1           Franz John        PERSON
2   Franz Beckenbauer         PERSON
3               Munich  ORGANIZATION
4             European  ORGANIZATION
5           Bundesliga  ORGANIZATION
6               German           GPE
7              Bavaria           GPE
8              Germany           GPE
9            FC Bayern  ORGANIZATION
10                UEFA  ORGANIZATION
11              Munich           GPE
12              Bayern           GPE
13             Overall           GPE
```

The Named Entity Chunker identifies named entities from the preceding text document, and we extract these named entities from the tagged annotated sentences and display them in the data frame as shown. You can clearly see how it has correctly identified `PERSON`, `ORGANIZATION`, and `GPE` related named entities, although a few of them are incorrectly identified.

We will now use the Stanford NER tagger on the same text and compare the results. For this, you need to have Java installed and then download the Stanford NER resources from `http://nlp.stanford.edu/software/stanford-ner-2014-08-27.zip`. Unzip them to a location of your choice (I used `E:/stanford` in my system). Once done, you can use `nltk`'s interface to access this, similar to what we did in Chapter 3 for constituency and dependency parsing. For more details on Stanford NER, visit `http://nlp.stanford.edu/software/CRF-NER.shtml`, the official web site, which also contains the latest version of their Named Entity Recognizer (I used an older version):

```
from nltk.tag import StanfordNERTagger
import os

# set java path in environment variables
java_path = r'C:\Program Files\Java\jdk1.8.0_102\bin\java.exe'
os.environ['JAVAHOME'] = java_path

# load stanford NER
sn = StanfordNERTagger('E:/stanford/stanford-ner-2014-08-27/classifiers/
english.all.3class.distsim.crf.ser.gz',
                       path_to_jar='E:/stanford/stanford-ner-2014-08-27/
                       stanford-ner.jar')
```

```
# tag sentences
ne_annotated_sentences = [sn.tag(sent) for sent in tokenized_sentences]

# extract named entities
named_entities = []
for sentence in ne_annotated_sentences:
    temp_entity_name = ''
    temp_named_entity = None
    for term, tag in sentence:
        # get terms with NE tags
        if tag != 'O':
            temp_entity_name = ' '.join([temp_entity_name, term]).strip() #
            get NE name
            temp_named_entity = (temp_entity_name, tag) # get NE and its
            category
        else:
            if temp_named_entity:
                named_entities.append(temp_named_entity)
                temp_entity_name = ''
                temp_named_entity = None

# get unique named entities
named_entities = list(set(named_entities))
# store named entities in a data frame
entity_frame = pd.DataFrame(named_entities,
                            columns=['Entity Name', 'Entity Type'])

# display results
In [118]: print entity_frame
         Entity Name   Entity Type
0         Franz John        PERSON
1  Franz Beckenbauer        PERSON
2            Germany      LOCATION
3             Bayern  ORGANIZATION
4            Bavaria      LOCATION
5             Munich      LOCATION
6          FC Bayern  ORGANIZATION
7               UEFA  ORGANIZATION
8      Bayern Munich  ORGANIZATION
```

The preceding output depicts various named entities obtained from our document. You can compare this with the results obtained from `nltk`'s NER chunker. The results here are definitely better—there are no misclassifications and each category is also assigned correctly. Some really interesting points: It has correctly identified *Munich* as a `LOCATION` and *Bayern Munich* as an `ORGANIZATION`. Does this mean the second NER tagger is better? Not really. It depends on the type of corpus you are analyzing, and you can even build your own NER tagger using supervised learning by training on pre-tagged corpora similar to what we did in Chapter 3. In fact, both the taggers just discussed have been trained on pre-tagged corpora like CoNLL, MUC, and Penn Treebank.

Analyzing Semantic Representations

We usually communicate in the form of messages in spoken form or in written form with other people or interfaces. Each of these messages is typically a collection of words, phrases, or sentences, and they have their own semantics and context. So far, we've talked about semantics and relations between various lexical units. But how do we represent the meaning of semantics conveyed by a message or messages? How do humans understand what someone is telling them? How do we believe in statements and propositions and evaluate outcomes and what action to take? It feels easy because the brain helps us with logic and reasoning—but computationally can we do the same?

The answer is yes we can. Frameworks like propositional logic and first-order logic help us in representation of semantics. We discussed this in detail in Chapter 1 in the subsection "Representation of Semantics" under the "Language Semantics" section. I encourage you to go through that once more to refresh your memory. In the following sections, we will look at ways to represent propositional and first order logic and prove or disprove propositions, statements, and predicates using practical examples and code.

Propositional Logic

We have already discussed propositional logic (PL) as the study of propositions, statements, and sentences. A *proposition* is usually declarative, having a binary value of being either true or false. There also exist various logical operators like conjunction, disjunction, implication, and equivalence, and we also study the effects of applying these operators on multiple propositions to understand their behavior and outcome.

Let us consider our example from Chapter 1 with regard to two propositions `P` and `Q` such that they can be represented as follows:

`P`: He is hungry

`Q`: He will eat a sandwich

We will now try to build the truth tables for various operations on these propositions using `nltk` based on the various logical operators discussed in Chapter 1 (refer to the "Propositional Logic" section for more details) and derive outcomes computationally:

```
import nltk
import pandas as pd
import os

# assign symbols and propositions
symbol_P = 'P'
symbol_Q = 'Q'
proposition_P = 'He is hungry'
propositon_Q = 'He will eat a sandwich'
# assign various truth values to the propositions
p_statuses = [False, False, True, True]
q_statuses = [False, True, False, True]
# assign the various expressions combining the logical operators
conjunction = '(P & Q)'
disjunction = '(P | Q)'
implication = '(P -> Q)'
```

```
equivalence = '(P <-> Q)'
expressions = [conjunction, disjunction, implication, equivalence]

# evaluate each expression using propositional logic
results = []
for status_p, status_q in zip(p_statuses, q_statuses):
    dom = set([])
    val = nltk.Valuation([(symbol_P, status_p),
                          (symbol_Q, status_q)])
    assignments = nltk.Assignment(dom)
    model = nltk.Model(dom, val)
    row = [status_p, status_q]
    for expression in expressions:
        # evaluate each expression based on proposition truth values
        result = model.evaluate(expression, assignments)
        row.append(result)
    results.append(row)
# build the result table
columns = [symbol_P, symbol_Q, conjunction,
           disjunction, implication, equivalence]
result_frame = pd.DataFrame(results, columns=columns)

# display results
In [125]: print 'P:', proposition_P
     ...: print 'Q:', propositon_Q
     ...: print
     ...: print 'Expression Outcomes:-'
     ...: print result_frame
P: He is hungry
Q: He will eat a sandwich

Expression Outcomes:-
       P      Q (P & Q) (P | Q) (P -> Q) (P <-> Q)
0  False  False   False   False     True      True
1  False   True   False    True     True     False
2   True  False   False    True    False     False
3   True   True    True    True     True      True
```

The preceding output depicts the various truth values of the two propositions, and when we combine them with various logical operators, you will find the results matching with what we manually evaluated in Chapter 1. For example, `P & Q` indicates *He is hungry and he will eat a sandwich* is `True` only when both of the individual propositions is `True`. We use `nltk`'s `Valuation` class to create a dictionary of the propositions and their various outcome states. We use the `Model` class to evaluate each expression, where the `evaluate()` function internally calls the recursive function `satisfy()`, which helps in evaluating the outcome of each expression with the propositions based on the assigned truth values.

First Order Logic

PL has several limitations, like the inability to represent facts or complex relationships and inferences. PL also has limited expressive power because for each new proposition we would need a unique symbolic representation, and it becomes very difficult to generalize facts. This is where first order logic (FOL) works really well with features like functions, quantifiers, relations, connectives, and symbols. It definitely provides a richer and more powerful representation for semantic information. The "First Order Logic" subsection under "Representation of Semantics" in Chapter 1 provides detailed information about how FOL works.

In this section, we will build several FOL representations similar to what we did manually in Chapter 1 using mathematical representations. Here we will build them in our code using similar syntax and leverage `nltk` and some theorem provers to prove the outcome of various expressions based on predefined conditions and relationships, similar to what we did for PL. The key takeaway for you from this section should be getting to know how to represent FOL representations in Python and how to perform FOL inference using proofs based on some goal and predefined rules and events. There are several theorem provers you can use for evaluating expressions and proving theorems. The `nltk` package has three main different types of provers: `Prover9`, `TableauProver`, and `ResolutionProver`. The first one is a free-to-use prover available for download at `www.cs.unm.edu/~mccune/prover9/download/`. You can extract the contents in a location of your choice (I used `E:/prover9`). We will be using both `ResolutionProver` and `Prover9` in our examples. The following snippet helps in setting up the necessary dependencies for FOL expressions and evaluations:

```
import nltk
import os
# for reading FOL expressions
read_expr = nltk.sem.Expression.fromstring
# initialize theorem provers (you can choose any)
os.environ['PROVER9'] = r'E:/prover9/bin'
prover = nltk.Prover9()
# I use the following one for our examples
prover = nltk.ResolutionProver()
```

Now that we have our dependencies ready, let us evaluate a few FOL expressions. Consider a simple expression that *If an entity jumps over another entity, the reverse cannot happen*. Assuming the entities to be `x` and `y`, we can represent this is FOL as ∀`x` ∀`y` (`jumps_over(x, y)` → ¬`jumps_over(y, x)`) which signifies that for all `x` and `y`, if `x` jumps over `y`, it implies that `y` cannot jump over `x`. Consider now that we have two entities `fox` and `dog` such that the `fox` jumps over the `dog` is an event which has taken place and can be represented by `jumps_over(fox, dog)`. Our end goal or objective is to evaluate the outcome of `jumps_over(dog, fox)` considering the preceding expression and the event that has occurred. The following snippet shows us how we can do this:

```
# set the rule expression
rule = read_expr('all x. all y. (jumps_over(x, y) -> -jumps_over(y, x))')
# set the event occured
```

```
event = read_expr('jumps_over(fox, dog)')
# set the outcome we want to evaluate -- the goal
test_outcome = read_expr('jumps_over(dog, fox)')

# get the result
In [132]: prover.prove(goal=test_outcome,
     ...:              assumptions=[event, rule],
     ...:              verbose=True)
[1] {-jumps_over(dog,fox)}                    A
[2] {jumps_over(fox,dog)}                     A
[3] {-jumps_over(z4,z3), -jumps_over(z3,z4)}  A
[4] {-jumps_over(dog,fox)}                    (2, 3)

Out[132]: False
```

The preceding output depicts the final result for our goal `test_outcome` is `False`, that is, the `dog` cannot jump over the `fox` if the `fox` has already jumped over the `dog` based on our rule expression and the events assigned to the assumptions parameter in the prover already given. The sequence of steps that lead to the result is also shown in the output. Let us now consider another FOL expression rule $\forall$`x studies(x, exam)` → `pass(x, exam)`, which tells us that for all instances of `x`, if `x` studies for the exam, he/she will pass the exam. Let us represent this rule and consider two students, `John` and `Pierre`, such that `John` does not study for the exam and `Pierre` does. Can we then find out the outcome whether they will pass the exam based on the given expression rule? The following snippet shows us how:

```
# set the rule expression
rule = read_expr('all x. (studies(x, exam) -> pass(x, exam))')
# set the events and outcomes we want to determine
event1 = read_expr('-studies(John, exam)')
test_outcome1 = read_expr('pass(John, exam)')
event2 = read_expr('studies(Pierre, exam)')
test_outcome2 = read_expr('pass(Pierre, exam)')

# get results
In [134]: prover.prove(goal=test_outcome1,
     ...:              assumptions=[event1, rule],
     ...:              verbose=True)
[1] {-pass(John,exam)}                  A
[2] {-studies(John,exam)}               A
[3] {-studies(z6,exam), pass(z6,exam)}  A
[4] {-studies(John,exam)}               (1, 3)

Out[134]: False

In [135]: prover.prove(goal=test_outcome2,
     ...:              assumptions=[event2, rule],
     ...:              verbose=True)
```

```
[1] {-pass(Pierre,exam)}                A
[2] {studies(Pierre,exam)}              A
[3] {-studies(z8,exam), pass(z8,exam)}  A
[4] {-studies(Pierre,exam)}             (1, 3)
[5] {pass(Pierre,exam)}                 (2, 3)
[6] {}                                  (1, 5)

Out[135]: True
```

Thus you can see from the above evaluations that `Pierre` does pass the exam because he studied for the exam, unlike `John` who doesn't pass the exam since he did not study for it.

Let us consider a more complex example with several entities. They perform several actions as follows:

- There are two dogs `rover` (`r`) and `alex` (`a`)
- There is one cat `garfield` (`g`)
- There is one fox `felix` (`f`)
- Two animals, alex (`a`) and felix (`f`) run, denoted by function `runs()`
- Two animals `rover` (`r`) and `garfield` (`g`) sleep, denoted by function `sleeps()`
- Two animals, `felix` (`f`) and `alex` (`a`) can jump over the other two, denoted by function `jumps_over()`

Taking all these assumptions, the following snippet builds an FOL-based model with the previously mentioned domain and assignment values based on the entities and functions. Once we build this model, we evaluate various FOL-based expressions to determine their outcome and prove some theorems like we did earlier:

```
# define symbols (entities\functions) and their values
rules = """
    rover => r
    felix => f
    garfield => g
    alex => a
    dog => {r, a}
    cat => {g}
    fox => {f}
    runs => {a, f}
    sleeps => {r, g}
    jumps_over => {(f, g), (a, g), (f, r), (a, r)}
    """
val = nltk.Valuation.fromstring(rules)
# view the valuation object of symbols and their assigned values
(dictionary)
```

```
In [143]: print val
{'rover': 'r', 'runs': set([('f',), ('a',)]), 'alex': 'a', 'sleeps':
set([('r',), ('g',)]), 'felix': 'f', 'fox': set([('f',)]), 'dog':
set([('a',), ('r',)]), 'jumps_over': set([('a', 'g'), ('f', 'g'), ('a',
'r'), ('f', 'r')]), 'cat': set([('g',)]), 'garfield': 'g'}

# define domain and build FOL based model
dom = {'r', 'f', 'g', 'a'}
m = nltk.Model(dom, val)

# evaluate various expressions
In [148]: print m.evaluate('jumps_over(felix, rover) & dog(rover) &
runs(rover)', None)
False

In [149]: print m.evaluate('jumps_over(felix, rover) & dog(rover) &
-runs(rover)', None)
True

In [150]: print m.evaluate('jumps_over(alex, garfield) & dog(alex) &
cat(garfield) & sleeps(garfield)', None)
True

# assign rover to x and felix to y in the domain
g = nltk.Assignment(dom, [('x', 'r'), ('y', 'f')])

# evaluate more expressions based on above assigned symbols
In [152]: print m.evaluate('runs(y) & jumps_over(y, x) & sleeps(x)', g)
True

In [153]: print m.evaluate('exists y. (fox(y) & runs(y))', g)
True
```

The preceding snippet depicts the evaluation of various expressions based on the valuation of different symbols based on the rules and domain. We create various FOL-based expressions and see their outcome based on the predefined assumptions. For example, the first expression gives us `False` because `rover` never `runs()` and the second and third expressions are `True` because they satisfy all the conditions like `felix` and `alex` can `jump over rover` or `garfield` and `rover` is a `dog` that does not `run` and `garfield` is a `cat`. The second set of expressions is evaluated based on assigning `felix` and `rover` to specific symbols in our domain (`dom`), and we pass that variable (`g`) when evaluating the expressions. We can even satisfy open formulae or expressions using the `satisfiers()` function as shown here:

```
# who are the animals who run?
In [154]: formula = read_expr('runs(x)')
     ...: print m.satisfiers(formula, 'x', g)
set(['a', 'f'])
```

```
# animals who run and are also a fox?
In [155]: formula = read_expr('runs(x) & fox(x)')
     ...: print m.satisfiers(formula, 'x', g)
set(['f'])
```

The preceding outputs are self-explanatory wherein we evaluate open-ended questions like *which animals run*? And also *which animals can run and are also foxes*? We get the relevant symbols in our outputs, which you can map back to the actual animal names (Hint: `a: alex, f: felix`). I encourage you to experiment with more propositions and FOL expressions by building your own assumptions, domain, and rules.

Sentiment Analysis

We will now discuss several concepts, techniques, and examples with regard to our second major topic in this chapter, sentiment analysis. Textual data, even though unstructured, mainly has two broad types of data points: factual based (objective) and opinion based (subjective). We briefly talked about these two categories at the beginning of this chapter when I introduced the concept of sentiment analysis and how it works best on text that has a subjective context. In general, social media, surveys, and feedback data all are heavily opinionated and express the beliefs, judgement, emotion, and feelings of human beings. Sentiment analysis, also popularly known as *opinion analysis/mining*, is defined as the process of using techniques like NLP, lexical resources, linguistics, and machine learning (ML) to extract subjective and opinion related information like emotions, attitude, mood, modality, and so on and try to use these to compute the polarity expressed by a text document. By *polarity*, I mean to find out whether the document expresses a positive, negative, or a neutral sentiment. More advanced analysis involves trying to find out more complex emotions like sadness, happiness, anger, and sarcasm.

Typically, sentiment analysis for text data can be computed on several levels, including on an individual sentence level, paragraph level, or the entire document as a whole. Often sentiment is computed on the document as a whole or some aggregations are done after computing the sentiment for individual sentences. *Polarity analysis* usually involves trying to assign some scores contributing to the positive and negative emotions expressed in the document and then finally assigning a label to the document based on the aggregate score. We will depict two major techniques for sentiment analysis here:

- Supervised machine learning
- Unsupervised lexicon-based

The key idea is to learn the various techniques typically used to tackle sentiment analysis problems so that you can apply them to solve your own problems. We will see how to re-use the concepts of supervised machine learning based classification algorithms from Chapter 4 here to classify documents to their associated sentiment. We will also use *lexicons*, which are dictionaries or vocabularies specially constructed to be used for sentiment analysis, and compute sentiment without using any supervised techniques. We will be carrying out our experiments on a large real-world dataset pertaining to movie reviews, which will make this task more interesting. We will compare the performance of the various algorithms and also try to perform some detailed analytics besides just analyzing polarity, which includes analyzing the subjectivity, mood, and modality of the movie reviews. Without further delay, let's get started!

Sentiment Analysis of IMDb Movie Reviews

We will be using a dataset of movie reviews obtained from the Internet Movie Database (IMDb) for sentiment analysis. This dataset, containing over 50,000 movie reviews, can be obtained from http://ai.stanford.edu/~amaas/data/sentiment/, courtesy of Stanford University and A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, Andrew Ng, and C. Potts, and this dataset was used in their famous paper, "Learning Word Vectors for Sentiment Analysis." We will be using 50,000 movie reviews from this dataset, which contain the review and a corresponding sentiment polarity label which is either positive or negative. A positive review is basically a movie review which was rated with more than six stars in IMDb, and a negative review was rated with less than five stars in IMDb. An important thing to remember here before we begin our exercise is the fact that many of these reviews, even though labeled positive or negative, might have some elements of negative or positive context respectively. Hence, there is a possibility for some overlap in many reviews, which make this task harder. Sentiment is not a quantitative number that you can compute and prove mathematically. It expresses complex emotions, feelings, and judgement, and hence you should never focus on trying to get a cent-percent perfect model but a model that generalizes well on data and works decently. We will start with setting up some necessary dependencies and utilities before moving on to the various techniques.

Setting Up Dependencies

There are several utility functions, data, and package dependencies that we need to set up before we jump into sentiment analysis. We will need our movie review dataset, some specific packages that we will be using in our implementations, and we will be defining some utility functions for text normalization, feature extracting, and model evaluation, similar to what we have used in previous chapters.

Getting and Formatting the Data

We will use the IMDb movie review dataset officially available in raw text files for each set (training and testing) from http://ai.stanford.edu/~amaas/data/sentiment/ as mentioned. You can download and unzip the files to a location of your choice and use the `review_data_extractor.py` file included along with the code files of this chapter to extract each review from the unzipped directory, parse them, and neatly format them into a data frame, which is then stored as a csv file named `movie_reviews.csv`. Otherwise, you can directly download the parsed and formatted file from https://github.com/dipanjanS/text-analytics-with-python/tree/master/Chapter-7, which contains all datasets and code used and is the official repository for this book. The data frame consists of two columns, `review` and `sentiment`, for each data point, which indicates the review for a movie and its corresponding sentiment (positive or negative).

Text Normalization

We will be normalizing and standardizing our text data similar to what we did in Chapter 6 as a part of text pre-processing and normalization. For this we will be re-using our `normalization.py` module from Chapter 6 with a few additions. This mainly includes

adding an HTML stripper to remove unnecessary HTML characters from text documents, as shown here:

```
from HTMLParser import HTMLParser

class MLStripper(HTMLParser):
    def __init__(self):
        self.reset()
        self.fed = []
    def handle_data(self, d):
        self.fed.append(d)
    def get_data(self):
        return ' '.join(self.fed)

def strip_html(text):
    html_stripper = MLStripper()
    html_stripper.feed(text)
    return html_stripper.get_data()
```

We also add a new function to normalize special accented characters and convert them into regular ASCII characters so as to standardize the text across all documents. The following snippet helps us achieve this:

```
def normalize_accented_characters(text):
    text = unicodedata.normalize('NFKD',
                                 text.decode('utf-8')
                                 ).encode('ascii', 'ignore')
    return text
```

The overall text normalization function is depicted in the following snippet and it re-uses the expand contractions, lemmatization, HTML unescaping, special characters removal, and stopwords removal functions from the previous chapter's normalization module:

```
def normalize_corpus(corpus, lemmatize=True,
                     only_text_chars=False,
                     tokenize=False):

    normalized_corpus = []
    for index, text in enumerate(corpus):
        text = normalize_accented_characters(text)
        text = html_parser.unescape(text)
        text = strip_html(text)
        text = expand_contractions(text, CONTRACTION_MAP)
        if lemmatize:
            text = lemmatize_text(text)
        else:
```

```
            text = text.lower()
        text = remove_special_characters(text)
        text = remove_stopwords(text)
        if only_text_chars:
            text = keep_text_characters(text)

        if tokenize:
            text = tokenize_text(text)
            normalized_corpus.append(text)
        else:
            normalized_corpus.append(text)

    return normalized_corpus
```

To re-use this code, you can make use of the `normalization.py` and `contractions.py` files provided with the code files of this chapter.

Feature Extraction

We will be reusing the same feature-extraction function we used in Chapter 6, and it is available as a part of the `utils.py` module. The function is shown here for the sake of completeness:

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer

def build_feature_matrix(documents, feature_type='frequency',
                         ngram_range=(1, 1), min_df=0.0, max_df=1.0):

    feature_type = feature_type.lower().strip()

    if feature_type == 'binary':
        vectorizer = CountVectorizer(binary=True, min_df=min_df,
                                     max_df=max_df, ngram_range=ngram_range)
    elif feature_type == 'frequency':
        vectorizer = CountVectorizer(binary=False, min_df=min_df,
                                     max_df=max_df, ngram_range=ngram_range)
    elif feature_type == 'tfidf':
        vectorizer = TfidfVectorizer(min_df=min_df, max_df=max_df,
                                     ngram_range=ngram_range)
    else:
        raise Exception("Wrong feature type entered. Possible values:
        'binary', 'frequency', 'tfidf'")

    feature_matrix = vectorizer.fit_transform(documents).astype(float)
    return vectorizer, feature_matrix
```

You can experiment with various features provided by this function, which include Bag of Words-based frequencies, occurrences, and TF-IDF based features.

Model Performance Evaluation

We will be evaluating our models based on precision, recall, accuracy, and F1-score, similar to our evaluation methods in Chapter 4 for text classification. Additionally we will be looking at the confusion matrix and detailed classification reports for each class, that is, the positive and negative classes to evaluate model performance. You can refer to the "Evaluating Classification Models" section in Chapter 4 to refresh your memory on the various model-evaluation metrics. The following function will help us in getting the model accuracy, precision, recall, and F1-score:

```
from sklearn import metrics
import numpy as np
import pandas as pd

def display_evaluation_metrics(true_labels, predicted_labels, positive_
class=1):
    print 'Accuracy:', np.round(
                        metrics.accuracy_score(true_labels,
                                               predicted_labels),
                        2)
    print 'Precision:', np.round(
                        metrics.precision_score(true_labels,
                                               predicted_labels,
                                               pos_label=positive_class,
                                               average='binary'),
                        2)
    print 'Recall:', np.round(
                        metrics.recall_score(true_labels,
                                               predicted_labels,
                                               pos_label=positive_class,
                                               average='binary'),
                        2)
    print 'F1 Score:', np.round(
                        metrics.f1_score(true_labels,
                                               predicted_labels,
                                               pos_label=positive_class,
                                               average='binary'),
                        2)
```

We will also define a function to help us build the confusion matrix for evaluating the model predictions against the actual sentiment labels for the reviews. The following function will help us achieve that:

```
def display_confusion_matrix(true_labels, predicted_labels, classes=[1,0]):
    cm = metrics.confusion_matrix(y_true=true_labels,
                                  y_pred=predicted_labels,
                                  labels=classes)
    cm_frame = pd.DataFrame(data=cm,
                            columns=pd.MultiIndex(levels=[['Predicted:'],
                            classes],
                                                  labels=[[0,0],[0,1]]),
                            index=pd.MultiIndex(levels=[['Actual:'],
                            classes],
                                                labels=[[0,0],[0,1]]))
    print cm_frame
```

Finally, we will define a function for getting a detailed classification report per sentiment category (positive and negative) by displaying the precision, recall, F1-score, and support (number of reviews) for each of the classes:

```
def display_classification_report(true_labels, predicted_labels,
classes=[1,0]):
    report = metrics.classification_report(y_true=true_labels,
                                           y_pred=predicted_labels,
                                           labels=classes)
    print report
```

You will find all the preceding functions in the `utils.py` module along with the other code files for this chapter and you can re-use them as needed. Besides this, you need to make sure you have `nltk` and `pattern` installed—which you should already have by this point of time because we have used them numerous times in our previous chapters.

Preparing Datasets

We will be loading our movie reviews data and preparing two datasets, namely training and testing, similar to what we did in Chapter 4. We will train our supervised model on the training data and evaluate model performance on the testing data. For unsupervised models, we will directly evaluate them on the testing data so as to compare their performance with the supervised model. Besides that, we will also pick some sample positive and negative reviews to see how the different models perform on them:

```
import pandas as pd
import numpy as np
# load movie reviews data
dataset = pd.read_csv(r'E:/aclImdb/movie_reviews.csv')
# print sample data
In [235]: print dataset.head()
                                              review sentiment
0  One of the other reviewers has mentioned that ...  positive
1  A wonderful little production. <br /><br />The...  positive
2  I thought this was a wonderful way to spend ti...  positive
```

```
3  Basically there's a family where a little boy ...  negative
4  Petter Mattei's "Love in the Time of Money" is...  positive

# prepare training and testing datasets
train_data = dataset[:35000]
test_data = dataset[35000:]

train_reviews = np.array(train_data['review'])
train_sentiments = np.array(train_data['sentiment'])
test_reviews = np.array(test_data['review'])
test_sentiments = np.array(test_data['sentiment'])

# prepare sample dataset for experiments
sample_docs = [100, 5817, 7626, 7356, 1008, 7155, 3533, 13010]
sample_data = [(test_reviews[index],
                test_sentiments[index])
                  for index in sample_docs]
```

We have taken a total of 35,000 reviews out of the 50,000 to be our training dataset and we will evaluate our models and test them on the remaining 15,000 reviews. This is in line with a typical 70:30 separation used for training and testing dataset building. We have also extracted a total of eight reviews from the test dataset and we will be looking closely at the results for these documents as well as evaluating the model performance on the complete test dataset in the following sections.

Supervised Machine Learning Technique

As mentioned before, in this section we will be building a model to analyze sentiment using supervised ML. This model will learn from past reviews and their corresponding sentiment from the training dataset so that it can predict the sentiment for new reviews from the test dataset. The basic principle here is to use the same concepts we used for text classification such that the classes to predict here are positive and negative sentiment corresponding to the movie reviews.

We will be following the same workflow which we followed in Chapter 4 for text classification (refer to Figure 4-2 in Chapter 4) in the "Text Classification Blueprint" section. The following points summarize these steps:

1. Model training
    a. Normalize training data
    b. Extract features and build feature set and feature vectorizer
    c. Use supervised learning algorithm (SVM) to build a predictive model

2. Model testing
    a. Normalize testing data
    b. Extract features using training feature vectorizer
    c. Predict the sentiment for testing reviews using training model
    d. Evaluate model performance

To start, we will be building our training model using the steps in point 1. We will be using our normalization and feature-extraction modules discussed in previous sections:

```
from normalization import normalize_corpus
from utils import build_feature_matrix

# normalization
norm_train_reviews = normalize_corpus(train_reviews, lemmatize=True, only_
text_chars=True)
# feature extraction
vectorizer, train_features = build_feature_matrix(documents=norm_train_
reviews,
                                                  feature_type='tfidf',
                                                  ngram_range=(1, 1),
                                                  min_df=0.0, max_df=1.0)
```

We will now build our model using the *support vector machine* (SVM) algorithm which we used for text classification in Chapter 4. Refer to the "Support Vector Machines" subsection under the "Classification Algorithms" section in Chapter 4 to refresh your memory:

```
from sklearn.linear_model import SGDClassifier
# build the model
svm = SGDClassifier(loss='hinge', n_iter=200)
svm.fit(train_features, train_sentiments)
```

The preceding snippet trainings the classifier and builds the model that is in the `svm` variable, which we can now use for predicting sentiment for new movie reviews (not used for training) from the test dataset. Let us normalize and extract features from the test dataset first as mentioned in step 2 in our workflow:

```
# normalize reviews
norm_test_reviews = normalize_corpus(test_reviews, lemmatize=True, only_
text_chars=True)
# extract features
test_features = vectorizer.transform(norm_test_reviews)
```

Now that we have our features for the entire test dataset, before we predict the sentiment and measure model prediction performance for the entire test dataset, let us look at some of the predictions for the sample documents we extracted earlier:

```
# predict sentiment for sample docs from test data
In [253]: for doc_index in sample_docs:
     ...:     print 'Review:-'
     ...:     print test_reviews[doc_index]
     ...:     print 'Actual Labeled Sentiment:', test_sentiments[doc_index]
     ...:     doc_features = test_features[doc_index]
     ...:     predicted_sentiment = svm.predict(doc_features)[0]
     ...:     print 'Predicted Sentiment:', predicted_sentiment
     ...:     print
     ...:
     ...:
Review:-
Worst movie, (with the best reviews given it) I've ever seen. Over the top
dialog, acting, and direction. more slasher flick than thriller.With all the
great reviews this movie got I'm appalled that it turned out so silly. shame
on you martin scorsese
Actual Labeled Sentiment: negative
Predicted Sentiment: negative

Review:-
I hope this group of film-makers never re-unites.
Actual Labeled Sentiment: negative
Predicted Sentiment: negative

Review:-
no comment - stupid movie, acting average or worse... screenplay - no sense
at all... SKIP IT!
Actual Labeled Sentiment: negative
Predicted Sentiment: negative

Review:-
Add this little gem to your list of holiday regulars. It is<br /><br
/>sweet, funny, and endearing
Actual Labeled Sentiment: positive
Predicted Sentiment: positive

Review:-
a mesmerizing film that certainly keeps your attention... Ben Daniels is
fascinating (and courageous) to watch.
Actual Labeled Sentiment: positive
Predicted Sentiment: positive

Review:-
This movie is perfect for all the romantics in the world. John Ritter has
never been better and has the best line in the movie! "Sam" hits close to
home, is lovely to look at and so much fun to play along with. Ben Gazzara
was an excellent cast and easy to fall in love with. I'm sure I've met
Arthur in my travels somewhere. All around, an excellent choice to pick up
any evening.!:-)
```

```
Actual Labeled Sentiment: positive
Predicted Sentiment: positive

Review:-
I don't care if some people voted this movie to be bad. If you want the
Truth this is a Very Good Movie! It has every thing a movie should have. You
really should Get this one.
Actual Labeled Sentiment: positive
Predicted Sentiment: negative

Review:-
Worst horror film ever but funniest film ever rolled in one you have got
to see this film it is so cheap it is unbeliaveble but you have to see it
really!!!! P.s watch the carrot
Actual Labeled Sentiment: positive
Predicted Sentiment: negative
```

You can look at each review, its actual labeled sentiment, and our predicted sentiment in the preceding output and see that we have some negative and positive reviews, and our model is able to correctly identify the sentiment for most of the sampled reviews except the last two reviews. If you look closely at the last two reviews, some part of the review has a negative sentiment (`"worst horror film"`, `"voted this movie to be bad"`) but the general sentiment or opinion of the person who wrote the review was intended positive. These are the examples I mentioned earlier about the overlap of positive and negative emotions, which makes it difficult for the model to predict the actual sentiment!

Let us now predict the sentiment for all our test dataset reviews and evaluate our model performance:

```
# predict the sentiment for test dataset movie reviews
predicted_sentiments = svm.predict(test_features)

# evaluate model prediction performance
from utils import display_evaluation_metrics, display_confusion_matrix,
display_classification_report

# show performance metrics
In [270]: display_evaluation_metrics(true_labels=test_sentiments,
     ...:                            predicted_labels=predicted_sentiments,
     ...:                            positive_class='positive')
Accuracy: 0.89
Precision: 0.88
Recall: 0.9
F1 Score: 0.89

# show confusion matrix
In [271]: display_confusion_matrix(true_labels=test_sentiments,
     ...:                           predicted_labels=predicted_sentiments,
     ...:                           classes=['positive', 'negative'])
```

```
                Predicted:
                  positive negative
Actual: positive       6770      740
        negative        912     6578

# show detailed per-class classification report
In [272]: display_classification_report(true_labels=test_sentiments,
     ...:                               predicted_labels=predicted_
                                        sentiments,
     ...:                               classes=['positive', 'negative'])
              precision    recall  f1-score   support

   positive       0.88      0.90      0.89      7510
   negative       0.90      0.88      0.89      7490

avg / total       0.89      0.89      0.89     15000
```

The preceding outputs show the various performance metrics that depict the performance of our SVM model with regard to predicting sentiment for movie reviews. We have an average sentiment prediction accuracy of 89 percent, which is really good if you compare it with standard baselines for text classification using supervised techniques. The classification report also shows a per-class detailed report, and we see that our F1-score (harmonic mean of precision and recall) is 89 percent for both positive and negative sentiment. The support metric shows the number of reviews having positive (7510) sentiment and negative (7490) sentiment. The confusion matrix shows how many reviews for which we predicted the correct sentiment (*positive*: 6770/7510, *negative*: 6578/7490) and the number of reviews for which we predicted the wrong sentiment (*positive*: 740/7510, *negative*: 912/7490). Do try out building more models with different features (Chapter 4 talks about different feature-extraction techniques) and different supervised learning algorithms. Can you get a better model which predicts sentiment more accurately?

Unsupervised Lexicon-based Techniques

So far, we used labeled training data to learn patterns using features from the movie reviews and their corresponding sentiment. Then we applied this knowledge learned on new movie reviews (the testing dataset) to predict their sentiment. Often, you may not have the convenience of a well-labeled training dataset. In those situations, you need to use unsupervised techniques for predicting the sentiment by using knowledgebases, ontologies, databases, and lexicons that have detailed information specially curated and prepared just for sentiment analysis.

As mentioned, a lexicon is a dictionary, vocabulary, or a book of words. In our case, lexicons are special dictionaries or vocabularies that have been created for analyzing sentiment. Most of these lexicons have a list of positive and negative polar words with some score associated with them, and using various techniques like the position of words, surrounding words, context, parts of speech, phrases, and so on, scores are assigned to the text documents for which we want to compute the sentiment. After aggregating these scores, we get the final sentiment. More advanced analyses can also be done, including detecting the subjectivity, mood, and modality. Various popular lexicons are used for sentiment analysis, including the following:

- AFINN lexicon
- Bing Liu's lexicon
- MPQA subjectivity lexicon
- SentiWordNet
- VADER lexicon
- Pattern lexicon

This is not an exhaustive list of lexicons that can be leveraged for sentiment analysis, and there are several other lexicons which can be easily obtained from the Internet. We will briefly discuss each lexicon and will be using the last three lexicons to analyze the sentiment for our testing dataset in more detail. Although these techniques are unsupervised, you can also use them to analyze and evaluate the sentiment for the training dataset too, but for the sake of consistency and to compare model performances with the supervised model, we will be performing all our analyses on the testing dataset.

AFINN Lexicon

The AFINN lexicon was curated and created by Finn Årup Nielsen, and more details are mentioned in his paper "A New ANEW: Evaluation of a Word List for Sentiment Analysis in Microblogs." The latest version, known as AFINN-111, consists of a total of 2477 words and phrases with their own scores based on sentiment polarity. The polarity basically indicates how positive, negative, or neutral the term might be with some numerical score. You can download it from `www2.imm.dtu.dk/pubdb/views/publication_details.php?id=6010`. It also talks about the lexicon in further details. The author of this lexicon has also built a Python wrapper over the AFINN lexicon, which you can directly use to predict the sentiment of text data. The repository is available from GitHub at `https://github.com/fnielsen/afinn`. You can install the `afinn` library directly and start analyzing sentiment. This library even has support for emoticons and smileys. Following is a sample of the AFINN-111 lexicon:

```
abandon        -2
abandoned      -2
abandons       -2
abducted       -2
abduction      -2
...
...
youthful        2
yucky          -2
yummy           3
zealot         -2
zealots        -2
zealous         2
```

The basic idea is to load the entire list of polar words and phrases in the lexicon along with their corresponding score (sample shown above) in memory and then find the same words/phrases and score them accordingly in a text document. Finally, these scores are aggregated, and the final sentiment and score can be obtained for a text document. Following is an example snippet based on the official documentation:

```
from afinn import Afinn
afn = Afinn(emoticons=True)

In [281]: print afn.score('I really hated the plot of this movie')
-3.0
In [282]: print afn.score('I really hated the plot of this movie :(')
-5.0
```

Thus you can use the `score()` function directly to evaluate the sentiment of your text documents, and from the preceding output you can see that they even give proper weightage to emoticons, which are used extensively in social media like Twitter and Facebook.

Bing Liu's Lexicon

This lexicon has been developed by Bing Liu over several years and is discussed in further details in his paper, by Nitin Jindal and Bing Liu, "Identifying Comparative Sentences in Text Documents." You can get more details about the lexicon at https://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html#lexicon, which also includes a link to download it as an archive (RAR format). This lexicon consists of over 6800 words divided into two files named `positive-words.txt`, containing around 2000+ words/phrases, and `negative-words.txt`, which contains around 4800+ words/phrases. The key idea is to leverage these words to contribute to the positive or negative polarity of any text document when they are identified in that document. This lexicon also includes many misspelled words, taking into account that words or terms are often misspelled on popular social media web sites.

MPQA Subjectivity Lexicon

MPQA stands for Multi-Perspective Question Answering, and it hosts a plethora of resources maintained by the University of Pittsburgh. It contains resources including opinion corpora, subjectivity lexicon, sense annotations, argument-based lexicon, and debate datasets. A lot of these can be leveraged for complex analysis of human emotions and sentiment. The subjectivity lexicon is maintained by Theresa Wilson, Janyce Wiebe, and Paul Hoffmann, and is discussed in detail in their paper, "Recognizing Contextual Polarity in Phrase-Level Sentiment Analysis," which focuses on contextual polarity. You can download the subjectivity lexicon from http://mpqa.cs.pitt.edu/lexicons/subj_lexicon/, which is their official website. It has subjectivity clues present in the dataset named `subjclueslen1-HLTEMNLP05.tff`, which is available once you extract the archive. Some sample lines from the dataset are depicted as follows:

```
type=weaksubj len=1 word1=abandoned pos1=adj stemmed1=n
priorpolarity=negative
type=weaksubj len=1 word1=abandonment pos1=noun stemmed1=n
priorpolarity=negative
type=weaksubj len=1 word1=abandon pos1=verb stemmed1=y
priorpolarity=negative
type=strongsubj len=1 word1=abase pos1=verb stemmed1=y
priorpolarity=negative
...
...
type=strongsubj len=1 word1=zealously pos1=anypos stemmed1=n
priorpolarity=negative
type=strongsubj len=1 word1=zenith pos1=noun stemmed1=n
priorpolarity=positive
type=strongsubj len=1 word1=zest pos1=noun stemmed1=n priorpolarity=positive
```

To understand this data, you can refer to the `readme` file provided along with the dataset. Basically, the clues in this dataset were curated and collected manually with efforts by the above-mentioned maintainers of this project. The various parameters mentioned above are explained briefly as follows:

- `type`: This has values that are either `strongsubj` indicating the presence of a strongly subjective context or `weaksubj` which indicates the presence of a weak/part subjective context.
- `len`: This points to the number of words in the term of the clue (all are single words of length 1 for now).
- `word1`: The actual term present as a token or a stem of the actual token.
- `pos1`: The part of speech for the term (clue) and it can be `noun`, `verb`, `adj`, `adverb`, or `anypos`.
- `stemmed1`: This indicates if the clue (term) is stemmed (`y`) or not stemmed (`n`). If it is stemmed, it can match all its other variants having the same `pos1` tag.
- `priorpolarity`: This has values of negative, positive, both, or neutral, and indicates the polarity of the sentiment associated with this clue (term).

The idea is to load this lexicon into a database or memory (hint: Python dictionary works well) and then use it similarly to the previous lexicons to analyze the sentiment associated with any text document.

SentiWordNet

We know that WordNet is perhaps one of the most popular corpora for the English language, used extensively in semantic analysis, and it introduces the concept of synsets. The SentiWordNet lexicon is a lexical resource used for sentiment analysis and opinion mining. For each synset present in WordNet, the SentiWordNet lexicon assigns three sentiment scores to it, including a positive polarity score, a negative polarity score, and an objectivity score. You can find more details on the official web site http://sentiwordnet.isti.cnr.it, which includes research papers explaining the lexicon in detail and also a link to download the lexicon. The `nltk` package in Python provides an interface directly for accessing the SentiWordNet lexicon, and we will be using this to analyze the sentiment of our movie reviews. The following snippet shows an example synset and its sentiment scores using SentiWordNet:

```
import nltk
from nltk.corpus import sentiwordnet as swn
# get synset for 'good'
good = swn.senti_synsets('good', 'n')[0]
# print synset sentiment scores
In [287]: print 'Positive Polarity Score:', good.pos_score()
     ...: print 'Negative Polarity Score:', good.neg_score()
     ...: print 'Objective Score:', good.obj_score()
Positive Polarity Score: 0.5
Negative Polarity Score: 0.0
Objective Score: 0.5
```

Now that we know how to use the `sentiwordnet` interface, we define a function that can take in a body of text (movie review in our case) and analyze its sentiment by leveraging `sentiwordnet`:

```
from normalization import normalize_accented_characters, html_parser, strip_
html

def analyze_sentiment_sentiwordnet_lexicon(review,
                                           verbose=False):
    # pre-process text
    review = normalize_accented_characters(review)
    review = html_parser.unescape(review)
    review = strip_html(review)
    # tokenize and POS tag text tokens
    text_tokens = nltk.word_tokenize(review)
    tagged_text = nltk.pos_tag(text_tokens)
    pos_score = neg_score = token_count = obj_score = 0
    # get wordnet synsets based on POS tags
    # get sentiment scores if synsets are found
    for word, tag in tagged_text:
        ss_set = None
```

```
        if 'NN' in tag and swn.senti_synsets(word, 'n'):
            ss_set = swn.senti_synsets(word, 'n')[0]
        elif 'VB' in tag and swn.senti_synsets(word, 'v'):
            ss_set = swn.senti_synsets(word, 'v')[0]
        elif 'JJ' in tag and swn.senti_synsets(word, 'a'):
            ss_set = swn.senti_synsets(word, 'a')[0]
        elif 'RB' in tag and swn.senti_synsets(word, 'r'):
            ss_set = swn.senti_synsets(word, 'r')[0]
        # if senti-synset is found
        if ss_set:
            # add scores for all found synsets
            pos_score += ss_set.pos_score()
            neg_score += ss_set.neg_score()
            obj_score += ss_set.obj_score()
            token_count += 1

    # aggregate final scores
    final_score = pos_score - neg_score
    norm_final_score = round(float(final_score) / token_count, 2)
    final_sentiment = 'positive' if norm_final_score >= 0 else 'negative'
    if verbose:
        norm_obj_score = round(float(obj_score) / token_count, 2)
        norm_pos_score = round(float(pos_score) / token_count, 2)
        norm_neg_score = round(float(neg_score) / token_count, 2)
        # to display results in a nice table
        sentiment_frame = pd.DataFrame([[final_sentiment, norm_obj_score,
                                         norm_pos_score, norm_neg_score,
                                         norm_final_score]],
                                         columns=pd.MultiIndex(levels
                                         =[['SENTIMENT STATS:'],
                                                    ['Predicted Sentiment',
                                                     'Objectivity',
                                                     'Positive', 'Negative',
                                                      'Overall']],
                                                    labels=[[0,0,0,0,0],
                                                    [0,1,2,3,4]]))
                                                    print sentiment_frame

    return final_sentiment
```

The comments in the preceding function are pretty self-explanatory. We take in a body of text (a movie review), do some initial pre-processing, and then tokenize and POS tag the tokens. For each pair of (word, tag) we check if any senti-synsets exist for the same word and its corresponding tag. If there is a match, we take the first senti-synset and store its sentiment scores in corresponding variables, and finally we aggregate its scores. We can now see the preceding function in action for our sample reviews (in the `sample_data` variable we created earlier from the test data) in the following snippet:

```
# detailed sentiment analysis for sample reviews
In [292]: for review, review_sentiment in sample_data:
     ...:     print 'Review:'
     ...:     print review
     ...:     print
     ...:     print 'Labeled Sentiment:', review_sentiment
     ...:     print
     ...:     final_sentiment = analyze_sentiment_sentiwordnet_
              lexicon(review,
     ...:
              verbose=True)
     ...:     print '-'*60
     ...:
     ...:
Review:
Worst movie, (with the best reviews given it) I've ever seen. Over the top
dialog, acting, and direction. more slasher flick than thriller.With all the
great reviews this movie got I'm appalled that it turned out so silly. shame
on you martin scorsese

Labeled Sentiment: negative

     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0            negative        0.83     0.08     0.09   -0.01
------------------------------------------------------------
Review:
I hope this group of film-makers never re-unites.

Labeled Sentiment: negative

     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0            negative        0.71     0.04     0.25   -0.21
------------------------------------------------------------
Review:
no comment - stupid movie, acting average or worse... screenplay - no sense
at all... SKIP IT!

Labeled Sentiment: negative

     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0            negative        0.81     0.04     0.15   -0.11
------------------------------------------------------------
Review:
Add this little gem to your list of holiday regulars. It is<br /><br
/>sweet, funny, and endearing
```

```
Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0            positive        0.76     0.18     0.06    0.13
-----------------------------------------------------------
Review:
a mesmerizing film that certainly keeps your attention... Ben Daniels is
fascinating (and courageous) to watch.

Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0            positive        0.84     0.14     0.03    0.11
-----------------------------------------------------------
Review:
This movie is perfect for all the romantics in the world. John Ritter has
never been better and has the best line in the movie! "Sam" hits close to
home, is lovely to look at and so much fun to play along with. Ben Gazzara
was an excellent cast and easy to fall in love with. I'm sure I've met
Arthur in my travels somewhere. All around, an excellent choice to pick up
any evening.!:-)

Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0            positive        0.75      0.2     0.05    0.15
-----------------------------------------------------------
Review:
I don't care if some people voted this movie to be bad. If you want the
Truth this is a Very Good Movie! It has every thing a movie should have. You
really should Get this one.

Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0            positive        0.73     0.21     0.06    0.15
-----------------------------------------------------------
Review:
Worst horror film ever but funniest film ever rolled in one you have got
to see this film it is so cheap it is unbeliaveble but you have to see it
really!!!! P.s watch the carrot

Labeled Sentiment: positive
```

```
     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0            positive        0.79     0.13     0.08    0.05
-----------------------------------------------------------
```

You can see detailed statistics related to each sentiment score and also the overall sentiment and compare it with the actual labeled sentiment for each review in the preceding output. Interestingly, we were able to predict the sentiment correctly for all our sampled reviews as compared to the supervised learning technique. But how well does this technique perform for our complete test movie reviews dataset? The following snippet will give us the answer!

```
# predict sentiment for test movie reviews dataset
sentiwordnet_predictions = [analyze_sentiment_sentiwordnet_lexicon(review)
                            for review in test_reviews]

from utils import display_evaluation_metrics, display_confusion_matrix,
display_classification_report

# get model performance statistics
In [295]: print 'Performance metrics:'
     ...: display_evaluation_metrics(true_labels=test_sentiments,
     ...:                            predicted_labels=sentiwordnet_
                                     predictions,
     ...:                            positive_class='positive')
     ...: print '\nConfusion Matrix:'
     ...: display_confusion_matrix(true_labels=test_sentiments,
     ...:                          predicted_labels=sentiwordnet_
                                   predictions,
     ...:                          classes=['positive', 'negative'])
     ...: print '\nClassification report:'
     ...: display_classification_report(true_labels=test_sentiments,
     ...:                               predicted_labels=sentiwordnet_
                                        predictions,
     ...:                               classes=['positive', 'negative'])
Performance metrics:
Accuracy: 0.59
Precision: 0.56
Recall: 0.92
F1 Score: 0.7

Confusion Matrix:
                Predicted:
                  positive negative
Actual: positive       6941      569
        negative       5510     1980

Classification report:
```

```
              precision    recall  f1-score   support

    positive       0.56      0.92      0.70      7510
    negative       0.78      0.26      0.39      7490

avg / total       0.67      0.59      0.55     15000
```

Our model has a sentiment prediction accuracy of around 60% and an F1-score of 70% approximately. If you look at the detailed classification report and the confusion matrix, you will observe that we correctly classify 6941/7510 positive movie reviews as positive, but we incorrectly classify 5510/7490 negative movie reviews as positive—which is quite high! A way to redress this would be to change our logic slightly in our function and relax the threshold for overall sentiment score to decide whether a document will have an overall positive or negative sentiment from 0 to maybe 0.1 or higher. Experiment with this threshold and see what kind of results you get.

VADER Lexicon

VADER stands for Valence Aware Dictionary and sEntiment Reasoner. It is a lexicon with a rule-based sentiment analysis framework that was specially built for analyzing sentiment from social media resources. This lexicon was developed by C. J. Hutto and Eric Gilbert, and you will find further details in the paper, "VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text." You can read more about it and even download the dataset or install the library from https://github.com/cjhutto/vaderSentiment, which contains all the resources pertaining to the VADER lexicon. The file `vader_sentiment_lexicon.txt` contains all the necessary sentiment scores associated with various terms, including words, emoticons, and even slang language-based tokens (like `lol`, `wtf`, `nah`, and so on). There are over 9000 lexical features from which it was further curated to 7500 lexical features in this lexicon with proper validated valence scores. Each feature was rated on a scale from "`[-4] Extremely Negative`" to "`[4] Extremely Positive`", with allowance for "`[0] Neutral (or Neither, N/A)`". This curation was done by keeping all lexical features which had a non-zero mean rating and whose standard deviation was less than 2.5, which was determined by the aggregate of ten independent raters. A sample of the VADER lexicon is depicted as follows:

```
)-:<   -2.2   0.4     [-2, -2, -2, -2, -2, -2, -3, -3, -2, -2]
)-:{   -2.1   0.9434  [-1, -3, -2, -1, -2, -2, -3, -4, -1, -2]
):     -1.8   0.87178 [-1, -3, -1, -2, -1, -3, -1, -3, -1, -2]
...
...
resolved      0.7   0.78102  [1, 2, 0, 1, 1, 0, 2, 0, 0, 0]
resolvent     0.7   0.78102  [1, 0, 1, 2, 0, -1, 1, 1, 1, 1]
resolvents    0.4   0.66332  [2, 0, 0, 1, 0, 0, 1, 0, 0, 0]
...
...
}:-(   -2.1   0.7       [-2, -1, -2, -2, -2, -4, -2, -2, -2, -2]
}:-)    0.3   1.61555   [1, 1, -2, 1, -1, -3, 2, 2, 1, 1]
```

Each line in the preceding lexicon depicts a unique term, which can be a word or even an emoticon. The first term indicates the word/emoticon, the second column indicates the mean or average score, the third column indicates the standard deviation, and the final column indicates a list of scores given by ten independent scorers. The `nltk` package has a nice interface for leveraging the VADER lexicon, and the following function makes use of the same for analyzing sentiment for any text document:

```
from nltk.sentiment.vader import SentimentIntensityAnalyzer

def analyze_sentiment_vader_lexicon(review,
                                    threshold=0.1,
                                    verbose=False):
    # pre-process text
    review = normalize_accented_characters(review)
    review = html_parser.unescape(review)
    review = strip_html(review)
    # analyze the sentiment for review
    analyzer = SentimentIntensityAnalyzer()
    scores = analyzer.polarity_scores(review)
    # get aggregate scores and final sentiment
    agg_score = scores['compound']
    final_sentiment = 'positive' if agg_score >= threshold\
                                 else 'negative'
    if verbose:
        # display detailed sentiment statistics
        positive = str(round(scores['pos'], 2)*100)+'%'
        final = round(agg_score, 2)
        negative = str(round(scores['neg'], 2)*100)+'%'
        neutral = str(round(scores['neu'], 2)*100)+'%'
        sentiment_frame = pd.DataFrame([[final_sentiment, final, positive,
                                         negative, neutral]],
                columns=pd.MultiIndex(levels=[['SENTIMENT STATS:'],
                                              ['Predicted Sentiment',
                                               'Polarity Score',
                                               'Positive', 'Negative',
                                               'Neutral']],
                                      labels=[[0,0,0,0,0],[0,1,2,3,4]]))
        print sentiment_frame

    return final_sentiment
```

That function helps in computing the sentiment and various statistics associated with it for any text document (movie reviews in our case). The comments explain the main sections of the function, which include text-preprocessing, getting the necessary sentiment scores using the VADER lexicon, aggregating them, and computing the final sentiment (positive/negative) using a specific threshold we talked about earlier. A threshold of 0.1

seemed to work best on an average, but you can experiment further with it. The following snippet shows us how to use this function on our sampled test movie reviews:

```
# get detailed sentiment statistics
In [301]: for review, review_sentiment in sample_data:
     ...:     print 'Review:'
     ...:     print review
     ...:     print
     ...:     print 'Labeled Sentiment:', review_sentiment
     ...:     print
     ...:     final_sentiment = analyze_sentiment_vader_lexicon(review,
     ...:                                                       threshold=0.1,
     ...:                                                       verbose=True)
     ...:     print '-'*60

Review:
Worst movie, (with the best reviews given it) I've ever seen. Over the top
dialog, acting, and direction. more slasher flick than thriller.With all the
great reviews this movie got I'm appalled that it turned out so silly. shame
on you martin scorsese

Labeled Sentiment: negative

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0            negative           0.03    20.0%    18.0%   62.0%
------------------------------------------------------------
Review:
I hope this group of film-makers never re-unites.

Labeled Sentiment: negative

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0            positive           0.44    33.0%     0.0%   67.0%
------------------------------------------------------------
Review:
no comment - stupid movie, acting average or worse... screenplay - no sense
at all... SKIP IT!

Labeled Sentiment: negative

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0            negative           -0.8     0.0%    40.0%   60.0%
------------------------------------------------------------
Review:
Add this little gem to your list of holiday regulars. It is<br /><br />sweet,
funny, and endearing
```

```
Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0            positive           0.82    40.0%     0.0%   60.0%
-----------------------------------------------------------
Review:
a mesmerizing film that certainly keeps your attention... Ben Daniels is
fascinating (and courageous) to watch.

Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0            positive           0.71    31.0%     0.0%   69.0%
-----------------------------------------------------------
Review:
This movie is perfect for all the romantics in the world. John Ritter has
never been better and has the best line in the movie! "Sam" hits close to
home, is lovely to look at and so much fun to play along with. Ben Gazzara
was an excellent cast and easy to fall in love with. I'm sure I've met
Arthur in my travels somewhere. All around, an excellent choice to pick up
any evening.!:-)

Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0            positive           0.99    37.0%     2.0%   61.0%
-----------------------------------------------------------
Review:
I don't care if some people voted this movie to be bad. If you want the
Truth this is a Very Good Movie! It has every thing a movie should have. You
really should Get this one.

Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0            negative          -0.16    17.0%    14.0%   69.0%
-----------------------------------------------------------
Review:
Worst horror film ever but funniest film ever rolled in one you have got
to see this film it is so cheap it is unbeliaveble but you have to see it
really!!!! P.s watch the carrot
```

```
Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0            positive           0.49    11.0%    11.0%   77.0%
-------------------------------------------------------------
```

The preceding statistics are similar to our previous function except the `Positive`, `Negative`, and `Neutral` columns indicate the percentage or proportion of the document that is positive, negative, or neutral, and the final score is determined based on the polarity score and the threshold. The following snippet shows the model sentiment prediction performance on the entire test movie reviews dataset:

```
# predict sentiment for test movie reviews dataset
vader_predictions = [analyze_sentiment_vader_lexicon(review, threshold=0.1)
                     for review in test_reviews]

# get model performance statistics
In [302]: print 'Performance metrics:'
     ...: display_evaluation_metrics(true_labels=test_sentiments,
     ...:                            predicted_labels=vader_predictions,
     ...:                            positive_class='positive')
     ...: print '\nConfusion Matrix:'
     ...: display_confusion_matrix(true_labels=test_sentiments,
     ...:                          predicted_labels=vader_predictions,
     ...:                          classes=['positive', 'negative'])
     ...: print '\nClassification report:'
     ...: display_classification_report(true_labels=test_sentiments,
     ...:                               predicted_labels=vader_predictions,
     ...:                               classes=['positive', 'negative'])
Performance metrics:
Accuracy: 0.7
Precision: 0.65
Recall: 0.86
F1 Score: 0.74

Confusion Matrix:
                Predicted:
                  positive negative
Actual: positive       6434     1076
        negative       3410     4080

Classification report:
             precision    recall  f1-score   support

   positive       0.65      0.86      0.74      7510
   negative       0.79      0.54      0.65      7490

avg / total       0.72      0.70      0.69     15000
```

The preceding metrics depict that our model has a sentiment prediction accuracy of around 70 percent and an F1-score close to 75 percent, which is definitely better than our previous model. Also notice that we are able to correctly predict positive sentiment for 6434 out of 7510 positive movie reviews, and negative sentiment correctly for 4080 out of 7490 negative movie reviews.

Pattern Lexicon

The `pattern` package is a complete package for NLP, text analytics, and information retrieval. We discussed it in detail in previous chapters and have also used it several times to solve several problems. This package is developed by CLiPS (Computational Linguistics & Psycholinguistics), a research center associated with the Linguistics Department of the Faculty of Arts of the University of Antwerp. It has a sentiment module associated with it, along with modules for analyzing mood and modality of a body of text.

For sentiment analysis, it analyzes any body of text by decomposing it into sentences and then tokenizing it and tagging the various tokens with necessary parts of speech. It then uses its own subjectivity-based sentiment lexicon, which you can access from its official repository at `https://github.com/clips/pattern/blob/master/pattern/text/en/en-sentiment.xml`. It contains scores like polarity, subjectivity, intensity, and confidence, along with other tags like the part of speech, WordNet identifier, and so on. It then leverages this lexicon to compute the overall polarity and subjectivity score associated with a text document. A threshold of 0.1 is recommended by `pattern` itself to compute the final sentiment of a document as positive, and anything below it as negative.

You can also analyze the mood and modality of text documents by leveraging the mood and modality functions provided by the `pattern` package. The mood function helps in determining the mood expressed by a particular text document. This function returns `INDICATIVE`, `IMPERATIVE`, `CONDITIONAL`, or `SUBJUNCTIVE` for any text based on its content. The table in Figure 7-2 talks about each type of mood in further detail, courtesy of the official documentation provided by CLiPS `pattern`. The column `Use` talks about the typical usage patterns for each type of mood, and the examples provide some actual examples from the English language.

Mood	Form	Use	Example
INDICATIVE	none of the below	fact, belief	*It rains.*
IMPERATIVE	infinitive without *to*	command, warning	*Don't rain!*
CONDITIONAL	*would, could, should, may,* or *will, can* + *if*	conjecture	*It might rain.*
SUBJUNCTIVE	*wish, were,* or *it is* + infinitive	wish, opinion	*I hope it rains.*

Figure 7-2. *Different types of mood and their examples (figure courtesy of CLiPS pattern)*

Modality for any text represents the degree of certainty expressed by the text as a whole. This value is a number that ranges between 0 and 1. Values > 0.5 indicate factual texts having a high certainty, and < 0.5 indicate wishes and hopes and have a low

certainty associated with them. We will define a function now to analyze the sentiment for text documents using the pattern lexicon:

```
from pattern.en import sentiment, mood, modality

def analyze_sentiment_pattern_lexicon(review, threshold=0.1,
                                      verbose=False):
    # pre-process text
    review = normalize_accented_characters(review)
    review = html_parser.unescape(review)
    review = strip_html(review)
    # analyze sentiment for the text document
    analysis = sentiment(review)
    sentiment_score = round(analysis[0], 2)
    sentiment_subjectivity = round(analysis[1], 2)
    # get final sentiment
    final_sentiment = 'positive' if sentiment_score >= threshold\
                                  else 'negative'
    if verbose:
        # display detailed sentiment statistics
        sentiment_frame = pd.DataFrame([[final_sentiment, sentiment_score,
                                         sentiment_subjectivity]],
                                         columns=pd.MultiIndex(levels
                                         =[['SENTIMENT STATS:'],
                                                    ['Predicted Sentiment',
                                                    'Polarity Score',
                                                     'Subjectivity Score']],
                                                     labels=[[0,0,0],
                                                     [0,1,2]]))
        print sentiment_frame
        assessment = analysis.assessments
        assessment_frame = pd.DataFrame(assessment,
                                  columns=pd.MultiIndex(levels=[['DETAILED
                                  ASSESSMENT STATS:'],
                                                     ['Key Terms', 'Polarity
                                                     Score',
                                                     'Subjectivity Score',
                                                     'Type']],
                                                     labels=[[0,0,0,0],
                                                     [0,1,2,3]]))
                                                     print assessment_frame
                                                     print

    return final_sentiment
```

We will now test the function we defined to analyze the sentiment of our sample test movie reviews and observe the results. We take a threshold of 0.1 as the cut-off to decide between positive and negative sentiment for a document based on the aggregated sentiment polarity score, based on several experiments and recommendations from the official documentation:

```
# get detailed sentiment statistics
In [303]: for review, review_sentiment in sample_data:
     ...:     print 'Review:'
     ...:     print review
     ...:     print
     ...:     print 'Labeled Sentiment:', review_sentiment
     ...:     print
     ...:     final_sentiment = analyze_sentiment_pattern_lexicon(review,
     ...:
threshold=0.1,
     ...:
verbose=True)
     ...:     print '-'*60

Review:
Worst movie, (with the best reviews given it) I've ever seen. Over the top
dialog, acting, and direction. more slasher flick than thriller.With all the
great reviews this movie got I'm appalled that it turned out so silly. shame
on you martin scorsese

Labeled Sentiment: negative

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Subjectivity Score
0            negative           0.06               0.62
  DETAILED ASSESSMENT STATS:
                   Key Terms Polarity Score Subjectivity Score  Type
0                    [worst]           -1.0              1.000  None
1                     [best]            1.0              0.300  None
2                      [top]            0.5              0.500  None
3                   [acting]            0.0              0.000  None
4                     [more]            0.5              0.500  None
5                    [great]            0.8              0.750  None
6                  [appalled]           -0.8              1.000  None
7                    [silly]           -0.5              0.875  None

------------------------------------------------------------
Review:
I hope this group of film-makers never re-unites.
```

```
Labeled Sentiment: negative

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Subjectivity Score
0            negative            0.0                0.0
Empty DataFrame
Columns: [(DETAILED ASSESSMENT STATS:, Key Terms), (DETAILED ASSESSMENT
STATS:, Polarity Score), (DETAILED ASSESSMENT STATS:, Subjectivity Score),
(DETAILED ASSESSMENT STATS:, Type)]
Index: []

------------------------------------------------------------
Review:
no comment - stupid movie, acting average or worse... screenplay - no sense
at all... SKIP IT!

Labeled Sentiment: negative

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Subjectivity Score
0            negative          -0.36                0.5
  DETAILED ASSESSMENT STATS:
                  Key Terms Polarity Score Subjectivity Score  Type
0                  [stupid]          -0.80                1.0  None
1                  [acting]           0.00                0.0  None
2                 [average]          -0.15                0.4  None
3                [worse, !]          -0.50                0.6  None

------------------------------------------------------------
Review:
Add this little gem to your list of holiday regulars. It is<br /><br
/>sweet, funny, and endearing

Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Subjectivity Score
0            positive           0.19               0.67
  DETAILED ASSESSMENT STATS:
                  Key Terms Polarity Score Subjectivity Score  Type
0                  [little]        -0.1875                0.5  None
1                   [funny]         0.2500                1.0  None
2                [endearing]         0.5000                0.5  None

------------------------------------------------------------
Review:
a mesmerizing film that certainly keeps your attention... Ben Daniels is
fascinating (and courageous) to watch.
```

```
Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Subjectivity Score
0            positive            0.4               0.71
  DETAILED ASSESSMENT STATS:
                   Key Terms Polarity Score Subjectivity Score  Type
0              [mesmerizing]       0.300000           0.700000  None
1                [certainly]       0.214286           0.571429  None
2              [fascinating]       0.700000           0.850000  None

------------------------------------------------------------
Review:
This movie is perfect for all the romantics in the world. John Ritter has
never been better and has the best line in the movie! "Sam" hits close to
home, is lovely to look at and so much fun to play along with. Ben Gazzara
was an excellent cast and easy to fall in love with. I'm sure I've met
Arthur in my travels somewhere. All around, an excellent choice to pick up
any evening.!:-)

Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Subjectivity Score
0            positive           0.66               0.73
   DETAILED ASSESSMENT STATS:
                    Key Terms Polarity Score Subjectivity Score  Type
0                   [perfect]       1.000000           1.000000  None
1                    [better]       0.500000           0.500000  None
2                   [best, !]       1.000000           0.300000  None
3                    [lovely]       0.500000           0.750000  None
4                  [much, fun]       0.300000           0.200000  None
5                  [excellent]       1.000000           1.000000  None
6                       [easy]       0.433333           0.833333  None
7                       [love]       0.500000           0.600000  None
8                       [sure]       0.500000           0.888889  None
9              [excellent, !]       1.000000           1.000000  None
10                      [:-)]       0.500000           1.000000  mood

------------------------------------------------------------
Review:
I don't care if some people voted this movie to be bad. If you want the
Truth this is a Very Good Movie! It has every thing a movie should have.
You really should Get this one.
```

```
Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Subjectivity Score
0            positive           0.17               0.55
  DETAILED ASSESSMENT STATS:
                  Key Terms Polarity Score Subjectivity Score  Type
0                     [bad]           -0.7           0.666667  None
1           [very, good, !]            1.0           0.780000  None
2                  [really]            0.2           0.200000  None

------------------------------------------------------------
Review:
Worst horror film ever but funniest film ever rolled in one you have got
to see this film it is so cheap it is unbeliaveble but you have to see it
really!!!! P.s watch the carrot

Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Subjectivity Score
0            negative          -0.04               0.63
  DETAILED ASSESSMENT STATS:
                  Key Terms Polarity Score Subjectivity Score  Type
0                   [worst]      -1.000000                1.0  None
1                   [cheap]       0.400000                0.7  None
2       [really, !, !, !, !]       0.488281                0.2  None

------------------------------------------------------------
```

The preceding analysis shows the sentiment, polarity, and subjectivity scores for each sampled review. Besides this, we also see key terms and emotions and their polarity scores, which mainly contributed to the overall sentiment of each review. You can see that even exclamations and emoticons are also given importance and weightage when computing sentiment and polarity. The following snippet depicts the mood and modality for the sampled test movie reviews:

```
In [304]: for review, review_sentiment in sample_data:
     ...:     print 'Review:'
     ...:     print review
     ...:     print 'Labeled Sentiment:', review_sentiment
     ...:     print 'Mood:', mood(review)
     ...:     mod_score = modality(review)
     ...:     print 'Modality Score:', round(mod_score, 2)
     ...:     print 'Certainty:', 'Strong' if mod_score > 0.5 \
     ...:                                 else 'Medium' if mod_score > 0.35 \
     ...:                                                       else 'Low'
     ...:     print '-'*60
```

```
Review:
Worst movie, (with the best reviews given it) I've ever seen. Over the top
dialog, acting, and direction. more slasher flick than thriller.With all the
great reviews this movie got I'm appalled that it turned out so silly. shame
on you martin scorsese
Labeled Sentiment: negative
Mood: indicative
Modality Score: 0.75
Certainty: Strong
------------------------------------------------------------
Review:
I hope this group of film-makers never re-unites.
Labeled Sentiment: negative
Mood: subjunctive
Modality Score: -0.25
Certainty: Low
------------------------------------------------------------
Review:
no comment - stupid movie, acting average or worse... screenplay - no sense
at all... SKIP IT!
Labeled Sentiment: negative
Mood: indicative
Modality Score: 0.75
Certainty: Strong
------------------------------------------------------------
Review:
Add this little gem to your list of holiday regulars. It is<br /><br
/>sweet, funny, and endearing
Labeled Sentiment: positive
Mood: imperative
Modality Score: 1.0
Certainty: Strong
------------------------------------------------------------
Review:
a mesmerizing film that certainly keeps your attention... Ben Daniels is
fascinating (and courageous) to watch.
Labeled Sentiment: positive
Mood: indicative
Modality Score: 0.75
Certainty: Strong
------------------------------------------------------------
Review:
This movie is perfect for all the romantics in the world. John Ritter has
never been better and has the best line in the movie! "Sam" hits close to
home, is lovely to look at and so much fun to play along with. Ben Gazzara
was an excellent cast and easy to fall in love with. I'm sure I've met
Arthur in my travels somewhere. All around, an excellent choice to pick up
any evening.!:-)
```

```
Labeled Sentiment: positive
Mood: indicative
Modality Score: 0.58
Certainty: Strong
------------------------------------------------------------
Review:
I don't care if some people voted this movie to be bad. If you want the
Truth this is a Very Good Movie! It has every thing a movie should have. You
really should Get this one.
Labeled Sentiment: positive
Mood: conditional
Modality Score: 0.28
Certainty: Low
------------------------------------------------------------
Review:
Worst horror film ever but funniest film ever rolled in one you have got
to see this film it is so cheap it is unbeliaveble but you have to see it
really!!!! P.s watch the carrot
Labeled Sentiment: positive
Mood: indicative
Modality Score: 0.75
Certainty: Strong
------------------------------------------------------------
```

The preceding output depicts the mood, modality score, and the certainty factor expressed by each review. It is interesting to see phrases like "Add this little gem…" are correctly associated with the right mood, which is an *imperative*, and "I hope this…" is correctly associated with *subjunctive* mood. The other reviews have more of an *indicative* disposition, which is quite obvious since it expresses the beliefs of the review who wrote the movie review. Certainty is lower in cases of reviews that use words like "hope", "if", and higher in case of strongly opinionated reviews.

Finally, we will evaluate the sentiment prediction performance of this model on our entire test review dataset as we have done before for our other models. The following snippet achieves the same:

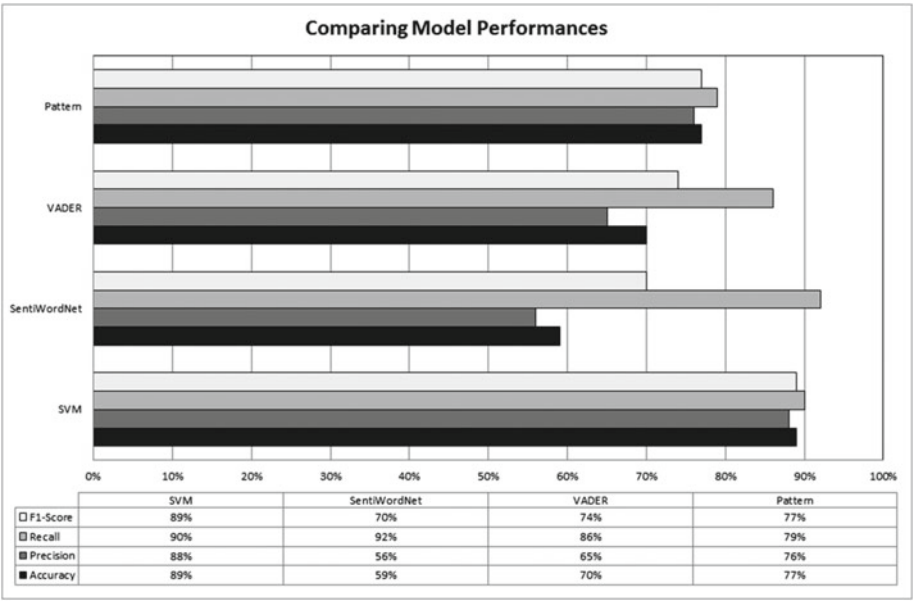
```
# predict sentiment for test movie reviews dataset
pattern_predictions = [analyze_sentiment_pattern_lexicon(review,
                      threshold=0.1)
                       for review in test_reviews]

# get model performance statistics
In [307]: print 'Performance metrics:'
     ...: display_evaluation_metrics(true_labels=test_sentiments,
     ...:                            predicted_labels=pattern_predictions,
     ...:                            positive_class='positive')
     ...: print '\nConfusion Matrix:'
     ...: display_confusion_matrix(true_labels=test_sentiments,
     ...:                          predicted_labels=pattern_predictions,
     ...:                          classes=['positive', 'negative'])
```

```
     ...: print '\nClassification report:'
     ...: display_classification_report(true_labels=test_sentiments,
     ...:                               predicted_labels=pattern_
                                        predictions,
     ...:                               classes=['positive', 'negative'])
Performance metrics:
Accuracy: 0.77
Precision: 0.76
Recall: 0.79
F1 Score: 0.77

Confusion Matrix:
                Predicted:
                  positive negative
Actual: positive       5958     1552
        negative       1924     5566

Classification report:
             precision    recall  f1-score   support

   positive       0.76      0.79      0.77      7510
   negative       0.78      0.74      0.76      7490

avg / total       0.77      0.77      0.77     15000
```

This model gives a better and more balanced performance toward predicting the sentiment of both positive and negative classes. We have an average sentiment prediction accuracy of 77 percent and an average F1-score of 77 percent for this model. Although the number of correct positive predictions has dropped from our previous model to 5958/7510 reviews, the number of correct predictions for negative reviews has increased significantly to 5566/7490 reviews.

Comparing Model Performances

We have built a supervised classification model and three unsupervised lexicon-based models to predict sentiment for movie reviews. For each model, we looked at its detailed analysis and statistics for calculating sentiment. We also evaluated each model on standard metrics like precision, recall, accuracy, and F1-score. In this section, we will briefly look at how each model's performance compares against the other models. Figure 7-3 shows the model performance metrics and a visualization comparing the metrics across all the models.



	SVM	SentiWordNet	VADER	Pattern
F1-Score	89%	70%	74%	77%
Recall	90%	92%	86%	79%
Precision	88%	56%	65%	76%
Accuracy	89%	59%	70%	77%

Figure 7-3. *Comparison of sentiment analysis model performances*

From the visualization and the table in Figure 7-3, it is clear that the supervised model using SVM gives us the best results, which are expected because it was trained on 35,000 training movie reviews. Pattern lexicon performs the best among the unsupervised techniques for our test movie reviews. Does this mean these models will always perform the best? Absolutely not. It depends on the data you are analyzing. Remember to consider various models and also to evaluate all the metrics when evaluating any model, and not just one or two. Some of the models in the chart have really high recall but low precision, which indicates these models have a tendency to make more wrong predictions or false positives. You can re-use these benchmarks and evaluate more sentiment analysis models as you experiment with different features, lexicons, and techniques.

Summary

In this final chapter, we have covered a variety of topics focused on semantic and sentiment analysis of textual data. We revisited several of our concepts from Chapter 1 with regard to language semantics. We looked at the WordNet corpus in detail and explored the concept of synsets with practical examples. We also analyzed various lexical semantic relations from Chapter 1 here, using synsets and real-world examples. We looked at relationships including entailments, homonyms and homographs, synonyms and antonyms, hyponyms and hypernyms, and holonyms and meronyms. Semantic relations and similarity computation techniques were also discussed in detail, with examples that leveraged common hypernyms among various synsets. Some popular techniques widely used in semantic and information extraction were discussed, including word sense disambiguation and named entity recognition, with examples. Besides semantic relations, we also revisited concepts related to semantic representations, namely propositional logic and first order logic. We leveraged the use of theorem provers and evaluated actual propositions and logical expressions computationally.

Next, we introduced the concept of sentiment analysis and opinion mining and saw how it is used in various domains like social media, surveys, and feedback data. We took a practical example of analyzing sentiment on actual movie reviews from IMDb and built several models that included supervised machine learning and unsupervised lexicon-based models. We looked at each technique and its results in detail and compared the performance across all our models.

This brings us to the end of this book. I hope the various concepts and techniques discussed here will be helpful to and that you can use the knowledge and techniques from this book when you tackle challenging problems in the world of text analytics and natural language processing. You may have seen by now that there is a lot of unexplored territory out there in the world of analyzing unstructured text data. I wish you the very best and would like to leave you with the parting thought from Occam's razor: *Sometimes the simplest solution is the best solution*.