CHAPTER 7

Analyzing Movie Reviews Sentiment

In this chapter, we continue with our focus on case-study oriented chapters, where we will focus on specific real-world problems and scenarios and how we can use Machine Learning to solve them. We will cover aspects pertaining to natural language processing (NLP), text analytics, and Machine Learning in this chapter. The problem at hand is sentiment analysis or opinion mining, where we want to analyze some textual documents and predict their sentiment or opinion based on the content of these documents. Sentiment analysis is perhaps one of the most popular applications of natural language processing and text analytics with a vast number of websites, books and tutorials on this subject. Typically sentiment analysis seems to work best on subjective text, where people express opinions, feelings, and their mood. From a real-world industry standpoint, sentiment analysis is widely used to analyze corporate surveys, feedback surveys, social media data, and reviews for movies, places, commodities, and many more. The idea is to analyze and understand the reactions of people toward a specific entity and take insightful actions based on their sentiment.

A text corpus consists of multiple text documents and each document can be as simple as a single sentence to a complete document with multiple paragraphs. Textual data, in spite of being highly unstructured, can be classified into two major types of documents. Factual documents that typically depict some form of statements or facts with no specific feelings or emotion attached to them. These are also known as objective documents. Subjective documents on the other hand have text that expresses feelings, moods, emotions, and opinions.

Sentiment analysis is also popularly known as opinion analysis or opinion mining. The key idea is to use techniques from text analytics, NLP, Machine Learning, and linguistics to extract important information or data points from unstructured text. This in turn can help us derive qualitative outputs like the overall sentiment being on a positive, neutral, or negative scale and quantitative outputs like the sentiment polarity, subjectivity, and objectivity proportions. Sentiment polarity is typically a numeric score that's assigned to both the positive and negative aspects of a text document based on subjective parameters like specific words and phrases expressing feelings and emotion. *Neutral* sentiment typically has `0` polarity since it does not express and specific sentiment, *positive* sentiment will have polarity `> 0`, and *negative* `< 0`. Of course, you can always change these thresholds based on the type of text you are dealing with; there are no hard constraints on this.

In this chapter, we focus on trying to analyze a large corpus of movie reviews and derive the sentiment. We cover a wide variety of techniques for analyzing sentiment, which include the following.

- Unsupervised lexicon-based models
- Traditional supervised Machine Learning models
- Newer supervised Deep Learning models
- Advanced supervised Deep Learning models

D. Sarkar et al., *Practical Machine Learning with Python*, https://doi.org/10.1007/978-1-4842-3207-1_7

Besides looking at various approaches and models, we also focus on important aspects in the Machine Learning pipeline including text pre-processing, normalization, and in-depth analysis of models, including model interpretation and topic models. The key idea here is to understand how we tackle a problem like sentiment analysis on unstructured text, learn various techniques, models and understand how to interpret the results. This will enable you to use these methodologies in the future on your own datasets. Let's get started!

Problem Statement

The main objective in this chapter is to predict the sentiment for a number of movie reviews obtained from the *Internet Movie Database* (IMDb). This dataset contains 50,000 movie reviews that have been pre-labeled with "positive" and "negative" sentiment class labels based on the review content. Besides this, there are additional movie reviews that are unlabeled. The dataset can be obtained from `http://ai.stanford.edu/~amaas/data/sentiment/`, courtesy of Stanford University and Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. This dataset was also used in their famous paper, *Learning Word Vectors for Sentiment Analysis* proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011). They have datasets in the form of raw text as well as already processed bag of words formats. We will only be using the raw labeled movie reviews for our analyses in this chapter. Hence our task will be to predict the sentiment of 15,000 labeled movie reviews and use the remaining 35,000 reviews for training our supervised models. We will still predict sentiments for only 15,000 reviews in case of unsupervised models to maintain consistency and enable ease of comparison.

Setting Up Dependencies

We will be using several Python libraries and frameworks specific to text analytics, NLP, and Machine Learning. While most of them will be mentioned in each section, you need to make sure you have `pandas`, `numpy`, `scipy`, and `scikit-learn` installed, which will be used for data processing and Machine Learning. Deep Learning frameworks used in this chapter include `keras` with the `tensorflow` backend, but you can also use `theano` as the backend if you choose to do so. NLP libraries which will be used include `spacy`, `nltk`, and `gensim`. Do remember to check that your installed `nltk` version is at least `>= 3.2.4`, otherwise, the `ToktokTokenizer` class may not be present. If you want to use a lower `nltk` version for some reason, you can use any other tokenizer like the default `word_tokenize()` based on the `TreebankWordTokenizer`. The version for `gensim` should be at least `2.3.0` and for `spacy`, the version used was `1.9.0`. We recommend using the latest version of spacy which was recently released (version 2.x) as this has fixed several bugs and added several improvements. You also need to download the necessary dependencies and corpora for `spacy` and `nltk` in case you are installing them for the first time. The following snippets should get this done. For `nltk` you need to type the following code from a Python or ipython shell after installing `nltk` using either `pip` or `conda`.

```
import nltk
nltk.download('all', halt_on_error=False)
```

For `spacy`, you need to type the following code in a Unix shell\windows command prompt, to install the library (use `pip install spacy` if you don't want to use `conda`) and also get the English model dependency.

```
$ conda config --add channels conda-forge
$ conda install spacy
$ python -m spacy download en
```

We also use our custom developed text pre-processing and normalization module, which you will find in the files named `contractions.py` and `text_normalizer.py`. Utilities related to supervised model fitting, prediction, and evaluation are present in `model_evaluation_utils.py`, so make sure you have these modules in the same directory and the other Python files and jupyter notebooks for this chapter.

Getting the Data

The dataset will be available along with the code files for this chapter in the GitHub repository for this book at https://github.com/dipanjanS/practical-machine-learning-with-python under the filename `movie_reviews.csv` containing 50,000 labeled IMDb movie reviews. You can also download the same data from http://ai.stanford.edu/~amaas/data/sentiment/ if needed. Once you have the CSV file, you can easily load it in Python using the `read_csv(...)` utility function from `pandas`.

Text Pre-Processing and Normalization

One of the key steps before diving into the process of feature engineering and modeling involves cleaning, pre-processing, and normalizing text to bring text components like phrases and words to some standard format. This enables standardization across a document corpus, which helps build meaningful features and helps reduce noise that can be introduced due to many factors like irrelevant symbols, special characters, XML and HTML tags, and so on. The file named `text_normalizer.py` has all the necessary utilities we will be using for our text normalization needs. You can also refer to the jupyter notebook named `Text Normalization Demo.ipynb` for a more interactive experience. The main components in our text normalization pipeline are described in this section.

- **Cleaning text**: Our text often contains unnecessary content like HTML tags, which do not add much value when analyzing sentiment. Hence we need to make sure we remove them before extracting features. The `BeautifulSoup` library does an excellent job in providing necessary functions for this. Our `strip_html_tags(...)` function enables in cleaning and stripping out HTML code.
- **Removing accented characters**: In our dataset, we are dealing with reviews in the English language so we need to make sure that characters with any other format, especially accented characters are converted and standardized into ASCII characters. A simple example would be converting **é** to **e**. Our `remove_accented_chars(...)` function helps us in this respect.
- **Expanding contractions**: In the English language, contractions are basically shortened versions of words or syllables. These shortened versions of existing words or phrases are created by removing specific letters and sounds. More than often vowels are removed from the words. Examples would be, ***do not*** to ***don't*** and ***I would*** to ***I'd***. Contractions pose a problem in text normalization because we have to deal with special characters like the apostrophe and we also have to convert each contraction to its expanded, original form. Our `expand_contractions(...)` function uses regular expressions and various contractions mapped in our `contractions.py` module to expand all contractions in our text corpus.
- **Removing special characters**: Another important task in text cleaning and normalization is to remove special characters and symbols that often add to the extra noise in unstructured text. Simple regexes can be used to achieve this. Our function `remove_special_characters(...)` helps us remove special characters. In our code, we have retained numbers but you can also remove numbers if you do not want them in your normalized corpus.

- **Stemming and lemmatization**: Word stems are usually the base form of possible words that can be created by attaching affixes like prefixes and suffixes to the stem to create new words. This is known as inflection. The reverse process of obtaining the base form of a word is known as stemming. A simple example are the words ***WATCHES***, ***WATCHING***, and ***WATCHED.*** They have the word root stem ***WATCH*** as the base form. The `nltk` package offers a wide range of stemmers like the `PorterStemmer` and `LancasterStemmer`. Lemmatization is very similar to stemming, where we remove word affixes to get to the base form of a word. However the base form in this case is known as the root word but not the root stem. The difference being that the root word is always a lexicographically correct word (present in the dictionary) but the root stem may not be so. We will be using lemmatization only in our normalization pipeline to retain lexicographically correct words. The function `lemmatize_text(...)` helps us with this aspect.
- **Removing stopwords**: Words which have little or no significance especially when constructing meaningful features from text are also known as stopwords or stop words. These are usually words that end up having the maximum frequency if you do a simple term or word frequency in a document corpus. Words like ***a***, ***an***, ***the,*** and so on are considered to be stopwords. There is no universal stopword list but we use a standard English language stopwords list from `nltk`. You can also add your own domain specific stopwords if needed. The function `remove_stopwords(...)` helps us remove stopwords and retain words having the most significance and context in a corpus.

We use all these components and tie them together in the following function called `normalize_corpus(...)`, which can be used to take a document corpus as input and return the same corpus with cleaned and normalized text documents.

```
def normalize_corpus(corpus, html_stripping=True, contraction_expansion=True,
                     accented_char_removal=True, text_lower_case=True,
                     text_lemmatization=True, special_char_removal=True,
                     stopword_removal=True):
    normalized_corpus = []
    # normalize each document in the corpus
    for doc in corpus:
        # strip HTML
        if html_stripping:
            doc = strip_html_tags(doc)
        # remove accented characters
        if accented_char_removal:
            doc = remove_accented_chars(doc)
        # expand contractions
        if contraction_expansion:
            doc = expand_contractions(doc)
        # lowercase the text
        if text_lower_case:
            doc = doc.lower()
        # remove extra newlines
        doc = re.sub(r'[\r|\n|\r\n]+', ' ',doc)
        # insert spaces between special characters to isolate them
        special_char_pattern = re.compile(r'([{.(-)!}])')
        doc = special_char_pattern.sub(" \\1 ", doc)
```

```
        # lemmatize text
        if text_lemmatization:
            doc = lemmatize_text(doc)
        # remove special characters
        if special_char_removal:
            doc = remove_special_characters(doc)
        # remove extra whitespace
        doc = re.sub(' +', ' ', doc)
        # remove stopwords
        if stopword_removal:
            doc = remove_stopwords(doc, is_lower_case=text_lower_case)

        normalized_corpus.append(doc)

    return normalized_corpus
```

The following snippet depicts a small demo of text normalization on a sample document using our normalization module.

```
In [1]: from text_normalizer import normalize_corpus

In [2]: document = """<p>Héllo! Héllo! can you hear me! I just heard about <b>Python</
b>!<br/>\r\n
   ...: It's an amazing language which can be used for Scripting, Web development,\r\n\r\n
   ...: Information Retrieval, Natural Language Processing, Machine Learning & Artificial
Intelligence!\n
   ...: What are you waiting for? Go and get started.<br/> He's learning, she's learning,
they've already\n\n
   ...: got a headstart!</p>
   ...:            """

In [3]: document
Out[3]: "<p>Héllo! Héllo! can you hear me! I just heard about <b>Python</b>!<br/>\r\n
\n              It's an amazing language which can be used for Scripting, Web development,
\r\n\r\n\n              Information Retrieval, Natural Language Processing, Machine
Learning & Artificial Intelligence!\n\n              What are you waiting for? Go and
get started.<br/> He's learning, she's learning, they've already\n\n\n              got a
headstart!</p>\n           "

In [4]: normalize_corpus([document], text_lemmatization=False, stopword_removal=False,
                         text_lower_case=False)
Out[4]: ['Hello Hello can you hear me I just heard about Python It is an amazing language
which can be used for Scripting Web development Information Retrieval Natural Language
Processing Machine Learning Artificial Intelligence What are you waiting for Go and get
started He is learning she is learning they have already got a headstart ']

In [5]: normalize_corpus([document])
Out[5]: ['hello hello hear hear python amazing language use scripting web development
information retrieval natural language processing machine learning artificial intelligence
wait go get start learn learn already get headstart']
```

Now that we have our normalization module ready, we can start modeling and analyzing our corpus. NLP and text analytics enthusiasts who might be interested in more in-depth details of text normalization can refer to the section "Text Normalization," Chapter 3, page 115, of *Text Analytics with Python* (Apress; Dipanjan Sarkar, 2016).

Unsupervised Lexicon-Based Models

We have talked about unsupervised learning methods in the past, which refer to specific modeling methods that can be applied directly on data features without the presence of labeled data. One of the major challenges in any organization is getting labeled datasets due the lack of time as well as resources to do this tedious task. Unsupervised methods are very useful in this scenario and we will be looking at some of these methods in this section. Even though we have labeled data, this section should give you a good idea of how lexicon based models work and you can apply the same in your own datasets when you do not have labeled data.

Unsupervised sentiment analysis models use well curated knowledgebases, ontologies, lexicons, and databases that have detailed information pertaining to subjective words, phrases including sentiment, mood, polarity, objectivity, subjectivity, and so on. A lexicon model typically uses a lexicon, also known as a dictionary or vocabulary of words specifically aligned toward sentiment analysis. Usually these lexicons contain a list of words associated with positive and negative sentiment, polarity (magnitude of negative or positive score), parts of speech (POS) tags, subjectivity classifiers (strong, weak, neutral), mood, modality, and so on. You can use these lexicons and compute sentiment of a text document by matching the presence of specific words from the lexicon, look at other additional factors like presence of negation parameters, surrounding words, overall context and phrases and aggregate overall sentiment polarity scores to decide the final sentiment score. There are several popular lexicon models used for sentiment analysis. Some of them are mentioned as follows.

- Bing Liu's Lexicon
- MPQA Subjectivity Lexicon
- Pattern Lexicon
- AFINN Lexicon
- SentiWordNet Lexicon
- VADER Lexicon

This is not an exhaustive list of lexicon models, but definitely lists among the most popular ones available today. We will be covering the last three lexicon models in more detail with hands-on code and examples using our movie review dataset. We will be using the last 15,000 reviews and predict their sentiment and see how well our model performs based on model evaluation metrics like accuracy, precision, recall, and F1-score, which we covered in detail in Chapter 5. Since we have labeled data, it will be easy for us to see how well our actual sentiment values for these movie reviews match our lexicon-model based predicted sentiment values. You can refer to the Python file titled `unsupervised_sentiment_analysis.py` for all the code used in this section or use the jupyter notebook titled `Sentiment Analysis - Unsupervised Lexical.ipynb` for a more interactive experience. Before we start our analysis, let's load the necessary dependencies and configuration settings using the following snippet.

```
In [1]: import pandas as pd
   ...: import numpy as np
   ...: import text_normalizer as tn
   ...: import model_evaluation_utils as meu
   ...:
   ...: np.set_printoptions(precision=2, linewidth=80)
```

Now, we can load our IMDb review dataset, subset out the last 15,000 reviews which will be used for our analysis, and normalize them using the following snippet.

```
In [2]: dataset = pd.read_csv(r'movie_reviews.csv')
   ...:
   ...: reviews = np.array(dataset['review'])
   ...: sentiments = np.array(dataset['sentiment'])
   ...:
   ...: # extract data for model evaluation
   ...: test_reviews = reviews[35000:]
   ...: test_sentiments = sentiments[35000:]
   ...: sample_review_ids = [7626, 3533, 13010]
   ...:
   ...: # normalize dataset
   ...: norm_test_reviews = tn.normalize_corpus(test_reviews)
```

We also extract out some sample reviews so that we can run our models on them and interpret their results in detail.

Bing Liu's Lexicon

This lexicon contains over 6,800 words which have been divided into two files named `positive-words.txt`, containing around 2,000+ words/phrases and `negative-words.txt`, which contains 4,800+ words/phrases. The lexicon has been developed and curated by Bing Liu over several years and has also been explained in detail in his original paper by Nitin Jindal and Bing Liu, "Identifying Comparative Sentences in Text Documents" proceedings of the 29th Annual International ACM SIGIR, Seattle 2006. If you want to use this lexicon, you can get it from `https://` www.cs.uic.edu/~liub/FBS/sentiment-analysis.html#lexicon, which also includes a link to download it as an archive (RAR format).

MPQA Subjectivity Lexicon

The term MPQA stands for Multi-Perspective Question Answering and it contains a diverse set of resources pertaining to opinion corpora, subjectivity lexicon, subjectivity sense annotations, argument lexicon, debate corpora, opinion finder, and many more. This is developed and maintained by the University of Pittsburgh and their official web site http://mpqa.cs.pitt.edu/ contains all the necessary information. The subjectivity lexicon is a part of their opinion finder framework and contains subjectivity clues and contextual polarity. Details on this can be found in the paper by Theresa Wilson, Janyce Wiebe, and Paul Hoffmann, "Recognizing Contextual Polarity in Phrase-Level Sentiment Analysis" proceeding of HLT-EMNLP-2005.

You can download the subjectivity lexicon from their official web site at http://mpqa.cs.pitt.edu/lexicons/subj_lexicon/, contains subjectivity clues present in the dataset named `subjclueslen1-HLTEMNLP05.tff`. The following snippet shows some sample lines from the lexicon.

```
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=zenith pos1=noun stemmed1=n priorpolarity=positive
type=strongsubj len=1 word1=zest pos1=noun stemmed1=n priorpolarity=positive
```

Each line consists of a specific word and its associated polarity, POS tag information, length (right now only words of length 1 are present), subjective context, and stem information.

Pattern Lexicon

The pattern package is a complete natural language processing framework available in Python which can be used for text processing, sentiment analysis and more. This has been developed by CLiPS (Computational Linguistics & Psycholinguistics), a research center associated with the Linguistics Department of the Faculty of Arts of the University of Antwerp. Pattern uses its own sentiment module which internally uses a lexicon which you can access from their official GitHub repository at https://github.com/clips/pattern/blob/master/pattern/text/en/en-sentiment.xml and this contains the complete subjectivity based lexicon database. Each line in the lexicon typically looks like the following sample.

```
<word form="absurd" wordnet_id="a-02570643" pos="JJ" sense="incongruous" polarity="-0.5"
subjectivity="1.0" intensity="1.0" confidence="0.9" />
```

Thus you get important metadata information like WordNet corpus identifiers, polarity scores, word sense, POS tags, intensity, subjectivity scores, and so on. These can in turn be used to compute sentiment over a text document based on polarity and subjectivity score. Unfortunately, `pattern` has still not been ported officially for Python `3.x` and it works on Python `2.7.x`. However, you can still load this lexicon and do your own modeling as needed.

AFINN Lexicon

The AFINN lexicon is perhaps one of the simplest and most popular lexicons that can be used extensively for sentiment analysis. Developed and curated by Finn Årup Nielsen, you can find more details on this lexicon in the paper by Finn Årup Nielsen, "A new ANEW: evaluation of a word list for sentiment analysis in microblogs", proceedings of the ESWC2011 Workshop. The current version of the lexicon is `AFINN-en-165.txt` and it contains over 3,300+ words with a polarity score associated with each word. You can find this lexicon at the author's official GitHub repository along with previous versions of this lexicon including `AFINN-111` at https://github.com/fnielsen/afinn/blob/master/afinn/data/. The author has also created a nice wrapper library on top of this in Python called `afinn` which we will be using for our analysis needs. You can import the library and instantiate an object using the following code.

```
In [3]: from afinn import Afinn
   ...:
   ...: afn = Afinn(emoticons=True)
```

We can now use this object and compute the polarity of our chosen four sample reviews using the following snippet.

```
In [4]: for review, sentiment in zip(test_reviews[sample_review_ids], test_
sentiments[sample_review_ids]):
   ...:     print('REVIEW:', review)
   ...:     print('Actual Sentiment:', sentiment)
   ...:     print('Predicted Sentiment polarity:', afn.score(review))
   ...:     print('-'*60)
REVIEW: no comment - stupid movie, acting average or worse... screenplay - no sense at
all... SKIP IT!
Actual Sentiment: negative
```

```
Predicted Sentiment polarity: -7.0
------------------------------------------------------------
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.
Actual Sentiment: positive
Predicted Sentiment polarity: 3.0
------------------------------------------------------------
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 unbelievable but you have to see it really!!!! P.S. Watch the
carrot
Actual Sentiment: positive
Predicted Sentiment polarity: -3.0
------------------------------------------------------------
```

We can compare the actual sentiment label for each review and also check out the predicted sentiment polarity score. A negative polarity typically denotes negative sentiment. To predict sentiment on our complete test dataset of 15,000 reviews (I used the raw text documents because AFINN takes into account other aspects like emoticons and exclamations), we can now use the following snippet. I used a threshold of `>= 1.0` to determine if the overall sentiment is positive else negative. You can choose your own threshold based on analyzing your own corpora in the future.

```
In [5]: sentiment_polarity = [afn.score(review) for review in test_reviews]
   ...: predicted_sentiments = ['positive' if score >= 1.0 else 'negative' for score in
                                                                    sentiment_polarity]
```

Now that we have our predicted sentiment labels, we can evaluate our model performance based on standard performance metrics using our utility function. See Figure 7-1.

```
In [6]: meu.display_model_performance_metrics(true_labels=test_sentiments,
                                              predicted_labels=predicted_sentiments,
                                              classes=['positive', 'negative'])
```

```
Model Performance metrics:
------------------------------
Accuracy: 0.71
Precision: 0.73
Recall: 0.71
F1 Score: 0.71

Model Classification report:
------------------------------
              precision    recall  f1-score   support

    positive       0.67      0.85      0.75      7510
    negative       0.79      0.57      0.67      7490

 avg / total       0.73      0.71      0.71     15000

Prediction Confusion Matrix:
------------------------------
                   Predicted:
                     positive negative
Actual: positive         6376     1134
        negative         3189     4301
```

Figure 7-1. *Model performance metrics for AFINN lexicon based model*

We get an overall **F1-Score** of **71%**, which is quite decent considering it's an unsupervised model. Looking at the confusion matrix we can clearly see that quite a number of negative sentiment based reviews have been misclassified as positive (3,189) and this leads to the lower recall of 57% for the negative sentiment class. Performance for positive class is better with regard to recall or hit-rate, where we correctly predicted 6,376 out of 7,510 positive reviews, but precision is 67% because of the many wrong positive predictions made in case of negative sentiment reviews.

SentiWordNet Lexicon

The WordNet corpus is definitely one of the most popular corpora for the English language used extensively in natural language processing and semantic analysis. WordNet gave us the concept of synsets or synonym sets. The SentiWordNet lexicon is based on WordNet synsets and can be used for sentiment analysis and opinion mining. The SentiWordNet lexicon typically assigns three sentiment scores for each WordNet synset. These include a positive polarity score, a negative polarity score and an objectivity score. Further details are available on the official web site http://sentiwordnet.isti.cnr.it, including research papers and download links for the lexicon. We will be using the nltk library, which provides a Pythonic interface into SentiWordNet. Consider we have the adjective *awesome*. We can get the sentiment scores associated with the synset for this word using the following snippet.

```
In [8]: from nltk.corpus import sentiwordnet as swn
   ...:
   ...: awesome = list(swn.senti_synsets('awesome', 'a'))[0]
   ...: print('Positive Polarity Score:', awesome.pos_score())
   ...: print('Negative Polarity Score:', awesome.neg_score())
   ...: print('Objective Score:', awesome.obj_score())
Positive Polarity Score: 0.875
Negative Polarity Score: 0.125
Objective Score: 0.0
```

Let's now build a generic function to extract and aggregate sentiment scores for a complete textual document based on matched synsets in that document.

```
def analyze_sentiment_sentiwordnet_lexicon(review,
                                           verbose=False):

    # tokenize and POS tag text tokens
    tagged_text = [(token.text, token.tag_) for token in tn.nlp(review)]
    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 list(swn.senti_synsets(word, 'n')):
            ss_set = list(swn.senti_synsets(word, 'n'))[0]
        elif 'VB' in tag and list(swn.senti_synsets(word, 'v')):
            ss_set = list(swn.senti_synsets(word, 'v'))[0]
        elif 'JJ' in tag and list(swn.senti_synsets(word, 'a')):
            ss_set = list(swn.senti_synsets(word, 'a'))[0]
        elif 'RB' in tag and list(swn.senti_synsets(word, 'r')):
            ss_set = list(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
```

Our function basically takes in a movie review, tags each word with its corresponding POS tag, extracts out sentiment scores for any matched synset token based on its POS tag, and finally aggregates the scores. This will be clearer when we run it on our sample documents.

```
In [10]: for review, sentiment in zip(test_reviews[sample_review_ids], test_
sentiments[sample_review_ids]):
    ...:     print('REVIEW:', review)
    ...:     print('Actual Sentiment:', sentiment)
    ...:     pred = analyze_sentiment_sentiwordnet_lexicon(review, verbose=True)
    ...:     print('-'*60)
REVIEW: no comment - stupid movie, acting average or worse... screenplay - no sense at
all... SKIP IT!
Actual Sentiment: negative
     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0            negative        0.76     0.09     0.15   -0.06
------------------------------------------------------------
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 Sentiment: positive
     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0            positive        0.74      0.2     0.06    0.14
------------------------------------------------------------
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 unbelievable but you have to see it really!!!! P.S. watch the
carrot
```

```
Actual Sentiment: positive
     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0            positive         0.8     0.14     0.07    0.07
-------------------------------------------------------------
```

We can clearly see the predicted sentiment along with sentiment polarity scores and an objectivity score for each sample movie review depicted in formatted dataframes. Let's use this model now to predict the sentiment of all our test reviews and evaluate its performance. A threshold of >=0 has been used for the overall sentiment polarity to be classified as positive and < 0 for negative sentiment. See Figure 7-2.

```
In [11]: predicted_sentiments = [analyze_sentiment_sentiwordnet_lexicon(review,
verbose=False)
                                                          for review in norm_test_reviews]
    ...: meu.display_model_performance_metrics(true_labels=test_sentiments,
                                               predicted_labels=predicted_sentiments,
    ...:                                       classes=['positive', 'negative'])
```

```
Model Performance metrics:
------------------------------
Accuracy: 0.69
Precision: 0.69
Recall: 0.69
F1 Score: 0.68

Model Classification report:
------------------------------
              precision    recall  f1-score   support

    positive       0.66      0.76      0.71      7510
    negative       0.72      0.61      0.66      7490

 avg / total       0.69      0.69      0.68     15000

Prediction Confusion Matrix:
------------------------------
                Predicted:
                  positive negative
Actual: positive      5742     1768
        negative      2932     4558
```

Figure 7-2. *Model performance metrics for SentiWordNet lexicon based model*

We get an overall **F1-Score** of **68%**, which is definitely a step down from our AFINN based model. While we have lesser number of negative sentiment based reviews being misclassified as positive, the other aspects of the model performance have been affected.

VADER Lexicon

The VADER lexicon, developed by C.J. Hutto, is a lexicon that is based on a rule-based sentiment analysis framework, specifically tuned to analyze sentiments in social media. VADER stands for Valence Aware Dictionary and Sentiment Reasoner. Details about this framework can be read in the original paper by Hutto, C.J. & Gilbert, E.E. (2014) titled "VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text", proceedings of the Eighth International Conference on Weblogs and Social Media (ICWSM-14). You can use the library based on `nltk's` interface under the `nltk.sentiment.vader` module. Besides this, you can also download the actual lexicon or install the framework from https://github.com/cjhutto/vaderSentiment, which also contains detailed information about VADER. This lexicon, present in the file titled `vader_lexicon.txt` contains necessary sentiment scores associated with words, emoticons and slangs (like *wtf, lol, nah,* and so on). There were a total of over 9000 lexical features from which over 7500 curated lexical features were finally selected in the 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)"`. The process of selecting lexical features was done by keeping all features that 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. We depict a sample from the `VADER` lexicon as follows.

```
:(     -1.9     1.13578 [-2, -3, -2, 0, -1, -1, -2, -3, -1, -4]
:)      2.0     1.18322 [2, 2, 1, 1, 1, 1, 4, 3, 4, 1]
...
terrorizing     -3.0    1.0     [-3, -1, -4, -4, -4, -3, -2, -3, -2, -4]
thankful         2.7    0.78102 [4, 2, 2, 3, 2, 4, 3, 3, 2, 2]
```

Each line in the preceding lexicon sample depicts a unique term, which can either be an emoticon or a word. The first token indicates the word/emoticon, the second token indicates the mean sentiment polarity score, the third token indicates the standard deviation, and the final token indicates a list of scores given by ten independent scorers. Now let's use VADER to analyze our movie reviews! We build our own modeling function as follows.

```
from nltk.sentiment.vader import SentimentIntensityAnalyzer

def analyze_sentiment_vader_lexicon(review,
                                    threshold=0.1,
                                    verbose=False):
    # pre-process text
    review = tn.strip_html_tags(review)
    review = tn.remove_accented_chars(review)
    review = tn.expand_contractions(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
```

In our modeling function, we do some basic pre-processing but keep the punctuations and emoticons intact. Besides this, we use VADER to get the sentiment polarity and also proportion of the review text with regard to positive, neutral and negative sentiment. We also predict the final sentiment based on a user-input threshold for the aggregated sentiment polarity. Typically, VADER recommends using positive sentiment

for aggregated polarity >= 0.5, *neutral* between [-0.5, 0.5], and *negative* for polarity < -0.5. We use a threshold of >= 0.4 for positive and < 0.4 for negative in our corpus. The following is the analysis of our sample reviews.

```
In [13]: for review, sentiment in zip(test_reviews[sample_review_ids], test_
sentiments[sample_review_ids]):
    ...:     print('REVIEW:', review)
    ...:     print('Actual Sentiment:', sentiment)
    ...:     pred = analyze_sentiment_vader_lexicon(review, threshold=0.4, verbose=True)
    ...:     print('-'*60)
REVIEW: no comment - stupid movie, acting average or worse... screenplay - no sense at
all... SKIP IT!
Actual Sentiment: negative
     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0            negative           -0.8     0.0%    40.0%   60.0%
------------------------------------------------------------
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.
Actual Sentiment: positive
     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive  Negative Neutral
0            negative          -0.16    16.0%  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 unbelievable but you have to see it really!!!! P.S. Watch the carrot
Actual Sentiment: positive
     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0            positive           0.49    11.0%    11.0%   77.0%
------------------------------------------------------------
```

We can see the details statistics pertaining to the sentiment and polarity for each sample movie review. Let's try out our model on the complete test movie review corpus now and evaluate the model performance. See Figure 7-3.

```
In [14]: predicted_sentiments = [analyze_sentiment_vader_lexicon(review, threshold=0.4,
                                                    verbose=False) for review in test_
reviews]
    ...: meu.display_model_performance_metrics(true_labels=test_sentiments,
                                           predicted_labels=predicted_sentiments,
    ...:                                   classes=['positive', 'negative'])
```

```
Model Performance metrics:
--------------------------
Accuracy: 0.71
Precision: 0.72
Recall: 0.71
F1 Score: 0.71

Model Classification report:
----------------------------
              precision    recall  f1-score   support

    positive       0.67      0.83      0.74      7510
    negative       0.78      0.59      0.67      7490

 avg / total       0.72      0.71      0.71     15000

Prediction Confusion Matrix:
----------------------------
                Predicted:
                  positive negative
Actual: positive      6235     1275
        negative      3068     4422
```

Figure 7-3. *Model performance metrics for VADER lexicon based model*

We get an overall **F1-Score** and model **accuracy** of **71%**, which is quite similar to the AFINN based model. The AFINN based model only wins out on the average precision by 1%; otherwise, both models have a similar performance.

Classifying Sentiment with Supervised Learning

Another way to build a model to understand the text content and predict the sentiment of the text based reviews is to use supervised Machine Learning. To be more specific, we will be using classification models for solving this problem. We have already covered the concepts relevant to supervised learning and classification in Chapter 1 under the section "Supervised Learning". With regard to details on building and evaluating classification models, you can head over to Chapter 5 and refresh your memory if needed. We will be building an automated sentiment text classification system in subsequent sections. The major steps to achieve this are mentioned as follows.

1. Prepare train and test datasets (optionally a validation dataset)
2. Pre-process and normalize text documents
3. Feature engineering
4. Model training
5. Model prediction and evaluation

These are the major steps for building our system. Optionally the last step would be to deploy the model in your server or on the cloud. Figure 7-4 shows a detailed workflow for building a standard text classification system with supervised learning (classification) models.

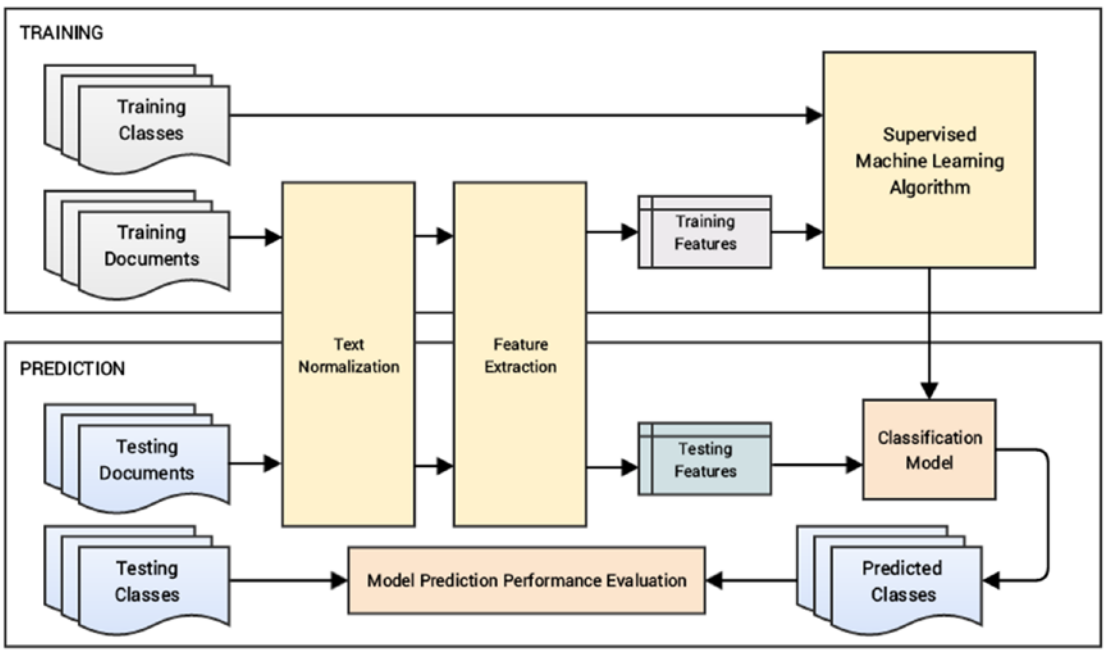

Figure 7-4. *Blueprint for building an automated text classification system (Source: Text Analytics with Python, Apress 2016)*

In our scenario, documents indicate the movie reviews and classes indicate the review sentiments that can either be positive or negative, making it a binary classification problem. We will build models using both traditional Machine Learning methods and newer Deep Learning in the subsequent sections. You can refer to the Python file titled `supervised_sentiment_analysis.py` for all the code used in this section or use the jupyter notebook titled `Sentiment Analysis - Supervised.ipynb` for a more interactive experience. Let's load the necessary dependencies and settings before getting started.

```
In [1]: import pandas as pd
   ...: import numpy as np
   ...: import text_normalizer as tn
   ...: import model_evaluation_utils as meu
   ...:
   ...: np.set_printoptions(precision=2, linewidth=80)
```

We can now load our IMDb movie reviews dataset, use the first 35,000 reviews for training models and the remaining 15,000 reviews as the test dataset to evaluate model performance. Besides this, we will also use our normalization module to normalize our review datasets (Steps 1 and 2 in our workflow).

```
In [2]: dataset = pd.read_csv(r'movie_reviews.csv')
   ...:
   ...: # take a peek at the data
   ...: print(dataset.head())
   ...: reviews = np.array(dataset['review'])
   ...: sentiments = np.array(dataset['sentiment'])
   ...:
   ...: # build train and test datasets
   ...: train_reviews = reviews[:35000]
   ...: train_sentiments = sentiments[:35000]
   ...: test_reviews = reviews[35000:]
   ...: test_sentiments = sentiments[35000:]
   ...:
   ...: # normalize datasets
   ...: norm_train_reviews = tn.normalize_corpus(train_reviews)
   ...: norm_test_reviews = tn.normalize_corpus(test_reviews)
                                              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
```

Our datasets are now prepared and normalized so we can proceed from Step 3 in our text classification workflow described earlier to build our classification system.

Traditional Supervised Machine Learning Models

We will be using traditional classification models in this section to classify the sentiment of our movie reviews. Our ***feature engineering*** techniques (Step 3) will be based on the Bag of Words model and the TF-IDF model, which we discussed extensively in the section titled "Feature Engineering on Text Data" in Chapter 4. The following snippet helps us engineer features using both these models on our `train` and `test` datasets.

```
In [3]: from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
   ...:
   ...: # build BOW features on train reviews
   ...: cv = CountVectorizer(binary=False, min_df=0.0, max_df=1.0, ngram_range=(1,2))
   ...: cv_train_features = cv.fit_transform(norm_train_reviews)
   ...: # build TFIDF features on train reviews
   ...: tv = TfidfVectorizer(use_idf=True, min_df=0.0, max_df=1.0, ngram_range=(1,2),
   ...:                      sublinear_tf=True)
   ...: tv_train_features = tv.fit_transform(norm_train_reviews)
   ...:
   ...: # transform test reviews into features
   ...: cv_test_features = cv.transform(norm_test_reviews)
   ...: tv_test_features = tv.transform(norm_test_reviews)
   ...:
   ...: print('BOW model:> Train features shape:', cv_train_features.shape,
              ' Test features shape:', cv_test_features.shape)
   ...: print('TFIDF model:> Train features shape:', tv_train_features.shape,
              ' Test features shape:', tv_test_features.shape)

BOW model:> Train features shape: (35000, 2114021)  Test features shape: (15000, 2114021)
TFIDF model:> Train features shape: (35000, 2114021)  Test features shape: (15000, 2114021)
```

We take into account word as well as bi-grams for our feature-sets. We can now use some traditional supervised Machine Learning algorithms which work very well on text classification. We recommend using logistic regression, support vector machines, and multinomial Naïve Bayes models when you work on your own datasets in the future. In this chapter, we built models using Logistic Regression as well as SVM. The following snippet helps initialize these classification model estimators.

```
In [4]: from sklearn.linear_model import SGDClassifier, LogisticRegression
   ...:
   ...: lr = LogisticRegression(penalty='l2', max_iter=100, C=1)
   ...: svm = SGDClassifier(loss='hinge', n_iter=100)
```

Without going into too many theoretical complexities, the logistic regression model is a supervised linear Machine Learning model used for classification regardless of its name. In this model, we try to predict the probability that a given movie review will belong to one of the discrete classes (binary classes in our scenario). The function used by the model for learning is represented here.

$$P\left(y = positive|X\right) = \sigma\left(\theta^T X\right)$$

$$P\left(y = negative|X\right) = 1 - \sigma\left(\theta^T X\right)$$

Where the model tries to predict the sentiment class using the feature vector X and $\sigma(z) = \frac{1}{1+e^{-z}}$, which is popularly known as the sigmoid function or logistic function or the logit function. The main objective of this model is to search for an optimal value of θ such that probability of the positive sentiment class is maximum when the feature vector X is for a positive movie review and small when it is for a negative movie review. The logistic function helps model the probability to describe the final prediction class. The optimal value of θ can be obtained by minimizing an appropriate cost\loss function using standard methods like

gradient descent (refer to the section, "The Three Stages of Logistic Regression" in Chapter 5 if you are interested in more details). Logistic regression is also popularly known as logit regression or `MaxEnt` (maximum entropy) classifier.

We will now use our utility function `train_predict_model(...)` from our `model_evaluation_utils` module to build a logistic regression model on our training features and evaluate the model performance on our test features (Steps 4 and 5). See Figure 7-5.

```
In [5]: # Logistic Regression model on BOW features
   ...: lr_bow_predictions = meu.train_predict_model(classifier=lr,
   ...:                         train_features=cv_train_features, train_labels=train_
                                sentiments,
   ...:                         test_features=cv_test_features, test_labels=test_sentiments)
   ...: meu.display_model_performance_metrics(true_labels=test_sentiments,
   ...:                                       predicted_labels=lr_bow_predictions,
   ...:                                       classes=['positive', 'negative'])
```

```
Model Performance metrics:
------------------------------
Accuracy: 0.91
Precision: 0.91
Recall: 0.91
F1 Score: 0.91

Model Classification report:
------------------------------

              precision    recall  f1-score   support

    positive       0.90      0.91      0.91      7510
    negative       0.91      0.90      0.90      7490

 avg / total       0.91      0.91      0.91     15000

Prediction Confusion Matrix:
------------------------------
                   Predicted:
                     positive negative
Actual: positive       6817      693
        negative        731     6759
```

Figure 7-5. *Model performance metrics for logistic regression on Bag of Words features*

We get an overall **F1-Score** and model **accuracy** of **91%**, as depicted in Figure 7-5, which is really excellent! We can now build a logistic regression model similarly on our TF-IDF features using the following snippet. See Figure 7-6.

```
In [6]: # Logistic Regression model on TF-IDF features
   ...: lr_tfidf_predictions = meu.train_predict_model(classifier=lr,
   ...:                          train_features=tv_train_features, train_labels=train_
                                 sentiments,
   ...:                          test_features=tv_test_features, test_labels=test_sentiments)
   ...: meu.display_model_performance_metrics(true_labels=test_sentiments,
   ...:                                       predicted_labels=lr_tfidf_predictions,
   ...:                                       classes=['positive', 'negative'])
```

```
Model Performance metrics:
------------------------------
Accuracy: 0.9
Precision: 0.9
Recall: 0.9
F1 Score: 0.9

Model Classification report:
------------------------------

              precision    recall  f1-score   support

    positive       0.89      0.90      0.90      7510
    negative       0.90      0.89      0.90      7490

 avg / total       0.90      0.90      0.90     15000

Prediction Confusion Matrix:
------------------------------
                   Predicted:
                     positive negative
Actual: positive       6780      730
        negative        828     6662
```

Figure 7-6. *Model performance metrics for logistic regression on TF-IDF features*

We get an overall **F1-Score** and model **accuracy** of **90%**, depicted in Figure 7-6, which is great but our previous model is still slightly better. You can similarly use the Support Vector Machine model estimator object `svm`, which we created earlier, and use the same snippet to train and predict using an SVM model. We obtained a maximum **accuracy** and **F1-score** of **90%** with the SVM model (refer to the jupyter notebook for step-by-step code snippets). Thus you can see how effective and accurate these supervised Machine Learning classification algorithms are in building a text sentiment classifier.

Newer Supervised Deep Learning Models

We have already mentioned multiple times in previous chapters about how Deep Learning has revolutionized the Machine Learning landscape over the last decade. In this section, we will be building some deep neural networks and train them on some advanced text features based on word embeddings to build a text sentiment classification system similar to what we did in the previous section. Let's load the following necessary dependencies before we start our analysis.

```
In [7]: import gensim
   ...: import keras
   ...: from keras.models import Sequential
   ...: from keras.layers import Dropout, Activation, Dense
   ...: from sklearn.preprocessing import LabelEncoder
Using TensorFlow backend.
```

If you remember in Chapter 4, we talked about encoding categorical class labels and also the one-hot encoding scheme. So far, our models in `scikit-learn` directly accepted the sentiment class labels as `positive` and `negative` and internally performed these operations. However for our Deep Learning models, we need to do this explicitly. The following snippet helps us tokenize our movie reviews and also converts the text-based sentiment class labels into one-hot encoded vectors (forms a part of Step 2).

```
In [8]: le = LabelEncoder()
   ...: num_classes=2
   ...: # tokenize train reviews & encode train labels
   ...: tokenized_train = [tn.tokenizer.tokenize(text)
   ...:                    for text in norm_train_reviews]
   ...: y_tr = le.fit_transform(train_sentiments)
   ...: y_train = keras.utils.to_categorical(y_tr, num_classes)
   ...: # tokenize test reviews & encode test labels
   ...: tokenized_test = [tn.tokenizer.tokenize(text)
   ...:                    for text in norm_test_reviews]
   ...: y_ts = le.fit_transform(test_sentiments)
   ...: y_test = keras.utils.to_categorical(y_ts, num_classes)
   ...:
   ...: # print class label encoding map and encoded labels
   ...: print('Sentiment class label map:', dict(zip(le.classes_, le.transform
        (le.classes_))))
   ...: print('Sample test label transformation:\n'+'-'*35,
   ...:       '\nActual Labels:', test_sentiments[:3], '\nEncoded Labels:', y_ts[:3],
   ...:       '\nOne hot encoded Labels:\n', y_test[:3])
Sentiment class label map: {'positive': 1, 'negative': 0}
```

```
Sample test label transformation:
----------------------------------
Actual Labels: ['negative' 'positive' 'negative']
Encoded Labels: [0 1 0]
One hot encoded Labels:
 [[ 1.  0.]
  [ 0.  1.]
  [ 1.  0.]]
```

Thus, we can see from the preceding sample outputs how our sentiment class labels have been encoded into numeric representations, which in turn have been converted into one-hot encoded vectors. The feature engineering techniques we will be using in this section (Step 3) are slightly more advanced word vectorization techniques that are based on the concept of word embeddings. We will be using the `word2vec` and GloVe models to generate embeddings. The `word2vec` model was built by Google and we have covered this in detail in Chapter 4 under the section "Word Embeddings". We will be choosing the size parameter to be 500 in this scenario representing feature vector size to be 500 for each word.

```
In [9]: # build word2vec model
   ...: w2v_num_features = 500
   ...: w2v_model = gensim.models.Word2Vec(tokenized_train, size=w2v_num_features,
        window=150,
   ...:                                     min_count=10, sample=1e-3)
```

We will be using the document word vector averaging scheme on this model from Chapter 4 to represent each movie review as an averaged vector of all the word vector representations for the different words in the review. The following function helps us compute averaged word vector representations for any corpus of text documents.

```
def averaged_word2vec_vectorizer(corpus, model, num_features):
    vocabulary = set(model.wv.index2word)
    def average_word_vectors(words, model, vocabulary, num_features):
        feature_vector = np.zeros((num_features,), dtype="float64")
        nwords = 0.
        for word in words:
            if word in vocabulary:
                nwords = nwords + 1.
                feature_vector = np.add(feature_vector, model[word])
        if nwords:
            feature_vector = np.divide(feature_vector, nwords)
        return feature_vector

    features = [average_word_vectors(tokenized_sentence, model, vocabulary, num_features)
                    for tokenized_sentence in corpus]
    return np.array(features)
```

We can now use the previous function to generate averaged word vector representations on our two movie review datasets.

```
In [10]: # generate averaged word vector features from word2vec model
    ...: avg_wv_train_features = averaged_word2vec_vectorizer(corpus=tokenized_train,
    ...:                                                      model=w2v_model, num_features=500)
    ...: avg_wv_test_features = averaged_word2vec_vectorizer(corpus=tokenized_test,
    ...:                                                     model=w2v_model, num_features=500)
```

The GloVe model, which stands for Global Vectors, is an unsupervised model for obtaining word vector representations. Created at Stanford University, this model is trained on various corpora like Wikipedia, Common Crawl, and Twitter and corresponding pre-trained word vectors are available that can be used for our analysis needs. You can refer to the original paper by Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014, called *GloVe: Global Vectors for Word Representation,* for more details. The `spacy` library provided 300-dimensional word vectors trained on the Common Crawl corpus using the GloVe model. They provide a simple standard interface to get feature vectors of size 300 for each word as well as the averaged feature vector of a complete text document. The following snippet leverages `spacy` to get the GloVe embeddings for our two datasets. Do note that you can also build your own GloVe model by leveraging other pre-trained models or by building a model on your own corpus by using the resources available at https://nlp.stanford.edu/projects/glove which contains pre-trained word embeddings, code and examples.

```
In [11]: # feature engineering with GloVe model
    ...: train_nlp = [tn.nlp(item) for item in norm_train_reviews]
    ...: train_glove_features = np.array([item.vector for item in train_nlp])
    ...:
    ...: test_nlp = [tn.nlp(item) for item in norm_test_reviews]
    ...: test_glove_features = np.array([item.vector for item in test_nlp])
```

You can check the feature vector dimensions for our datasets based on each of the previous models using the following code.

```
In [12]: print('Word2Vec model:> Train features shape:', avg_wv_train_features.shape,
                ' Test features shape:', avg_wv_test_features.shape)
    ...: print('GloVe model:> Train features shape:', train_glove_features.shape,
                ' Test features shape:', test_glove_features.shape)
Word2Vec model:> Train features shape: (35000, 500)  Test features shape: (15000, 500)
GloVe model:> Train features shape: (35000, 300)  Test features shape: (15000, 300)
```

We can see from the preceding output that as expected the `word2vec` model features are of size 500 and the GloVe features are of size 300. We can now proceed to Step 4 of our classification system workflow where we will build and train a deep neural network on these features. We have already briefly covered the various aspects and architectures with regard to deep neural networks in Chapter 1 under the section "Deep Learning". We will be using a fully-connected four layer deep neural network (multi-layer perceptron or deep ANN) for our model. We do not count the input layer usually in any deep architecture, hence our model will consist of three hidden layers of 512 neurons or units and one output layer with two units that will be used to either predict a positive or negative sentiment based on the input layer features. Figure 7-7 depicts our deep neural network model for sentiment classification.

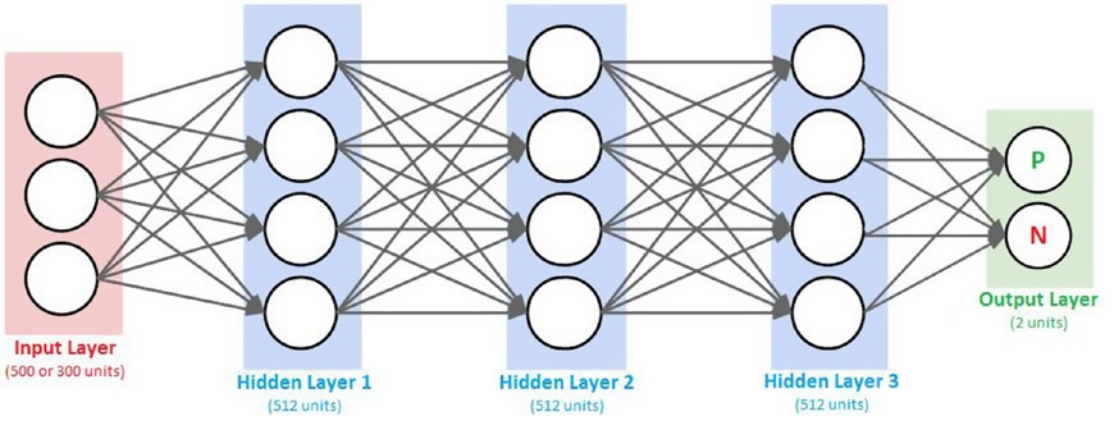

Figure 7-7. *Fully connected deep neural network model for sentiment classification*

We call this a fully connected deep neural network (DNN) because neurons or units in each pair of adjacent layers are fully pairwise connected. These networks are also known as deep artificial neural networks (ANNs) or Multi-Layer Perceptrons (MLPs) since they have more than one hidden layer. The following function leverages `keras` on top of `tensorflow` to build the desired DNN model.

```
def construct_deepnn_architecture(num_input_features):
    dnn_model = Sequential()
    dnn_model.add(Dense(512, activation='relu', input_shape=(num_input_features,)))
    dnn_model.add(Dropout(0.2))
    dnn_model.add(Dense(512, activation='relu'))
    dnn_model.add(Dropout(0.2))
    dnn_model.add(Dense(512, activation='relu'))
    dnn_model.add(Dropout(0.2))
    dnn_model.add(Dense(2))
    dnn_model.add(Activation('softmax'))

    dnn_model.compile(loss='categorical_crossentropy', optimizer='adam',
                      metrics=['accuracy'])
    return dnn_model
```

From the preceding function, you can see that we accept a parameter `num_input_features`, which decides the number of units needed in the input layer (`500` for `word2vec` and `300` for `glove` features). We build a `Sequential` model, which helps us linearly stack our hidden and output layers.

We use `512` units for all our hidden layers and the activation function `relu` indicates a rectified linear unit. This function is typically defined as $relu(x) = \max(0, x)$ where x is typically the input to a neuron. This is popularly known as the ramp function also in electronics and electrical engineering. This function is preferred now as compared to the previously popular sigmoid function because it tries to solve the vanishing gradient problem. This problem occurs when $x > 0$ and as x increases, the gradient from sigmoids becomes really small (almost vanishing) but relu prevents this from happening. Besides this, it also helps with faster convergence of gradient descent. We also use regularization in the network in the form of `Dropout` layers. By adding a dropout rate of `0.2`, it randomly sets `20%` of the input feature units to `0` at each update during training the model. This form of regularization helps prevent overfitting the model.

The final output layer consists of two units with a `softmax` activation function. The softmax function is basically a generalization of the logistic function we saw earlier, which can be used to represent a probability distribution over n possible class outcomes. In our case $n=2$ where the class can either be positive or negative and the softmax probabilities will help us determine the same. The binary softmax classifier is also interchangeably known as the binary logistic regression function.

The `compile(...)` method is used to configure the learning or training process of the DNN model before we actually train it. This involves providing a cost or loss function in the loss parameter. This will be the goal or objective which the model will try to minimize. There are various loss functions based on the type of problem you want to solve, for example the mean squared error for regression and categorical cross-entropy for classification. Check out https://keras.io/losses/ for a list of possible loss functions.

We will be using `categorical_crossentropy`, which helps us minimize the error or loss from the softmax output. We need an optimizer for helping us converge our model and minimize the loss or error function. Gradient descent or stochastic gradient descent is a popular optimizer. We will be using the `adam` optimizer which only required first order gradients and very little memory. Adam also uses momentum where basically each update is based on not only the gradient computation of the current point but also includes a fraction of the previous update. This helps with faster convergence. You can refer to the original paper from https://arxiv.org/pdf/1412.6980v8.pdf for further details on the ADAM optimizer. Finally, the `metrics` parameter is used to specify model performance metrics that are used to evaluate the model when training (but not used to modify the training loss itself). Let's now build a DNN model based on our `word2vec` input feature representations for our training reviews.

```
In [13]: w2v_dnn = construct_deepnn_architecture(num_input_features=500)
```

You can also visualize the DNN model architecture with the help of `keras`, similar to what we had done in Chapter 4, by using the following code. See Figure 7-8.

```
In [14]: from IPython.display import SVG
    ...: from keras.utils.vis_utils import model_to_dot
    ...:
    ...: SVG(model_to_dot(w2v_dnn, show_shapes=True, show_layer_names=False,
    ...:                  rankdir='TB').create(prog='dot', format='svg'))
```

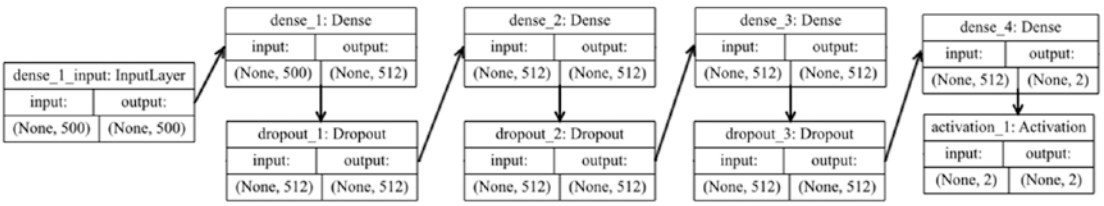

Figure 7-8. *Visualizing the DNN model architecture using* `keras`

We will now be training our model on our training reviews dataset of `word2vec` features represented by `avg_wv_train_features` (Step 4). We will be using the `fit(...)` function from `keras` for the training process and there are some parameters which you should be aware of. The `epoch` parameter indicates one complete forward and backward pass of all the training examples through the network. The `batch_size` parameter indicates the total number of samples which are propagated through the DNN model at a time for one backward and forward pass for training the model and updating the gradient. Thus if you have 1,000 observations and your batch size is 100, each epoch will consist of 10 iterations where 100 observations will be passed through the network at a time and the weights on the hidden layer units will be updated. We also

specify a `validation_split` of `0.1` to extract 10% of the training data and use it as a validation dataset for evaluating the performance at each epoch. The `shuffle` parameter helps shuffle the samples in each epoch when training the model.

```
In [18]: batch_size = 100
    ...: w2v_dnn.fit(avg_wv_train_features, y_train, epochs=5, batch_size=batch_size,
    ...:             shuffle=True, validation_split=0.1, verbose=1)
Train on 31500 samples, validate on 3500 samples
Epoch 1/5 31500/31500 - 11s - loss: 0.3097 - acc: 0.8720 - val_loss: 0.3159 - val_acc: 0.8646
Epoch 2/5 31500/31500 - 11s - loss: 0.2869 - acc: 0.8819 - val_loss: 0.3024 - val_acc: 0.8743
Epoch 3/5 31500/31500 - 11s - loss: 0.2778 - acc: 0.8857 - val_loss: 0.3012 - val_acc: 0.8763
Epoch 4/5 31500/31500 - 11s - loss: 0.2708 - acc: 0.8901 - val_loss: 0.3041 - val_acc: 0.8734
Epoch 5/5 31500/31500 - 11s - loss: 0.2612 - acc: 0.8920 - val_loss: 0.3023 - val_acc: 0.8763
```

The preceding snippet tells us that we have trained our DNN model on the training data for five epochs with 100 as the batch size. We get a validation accuracy of close to 88%, which is quite good. Time now to put our model to the real test! Let's evaluate our model performance on the test review `word2vec` features (Step 5).

```
In [19]: y_pred = w2v_dnn.predict_classes(avg_wv_test_features)
    ...: predictions = le.inverse_transform(y_pred)
    ...: meu.display_model_performance_metrics(true_labels=test_sentiments,
    ...:                predicted_labels=predictions, classes=['positive', 'negative'])
```

```
Model Performance metrics:
-------------------------------
Accuracy: 0.88
Precision: 0.88
Recall: 0.88
F1 Score: 0.88

Model Classification report:
-------------------------------

                   precision    recall  f1-score   support

         positive       0.88      0.89      0.88      7510
         negative       0.89      0.87      0.88      7490

avg / total       0.88      0.88      0.88     15000

Prediction Confusion Matrix:
------------------------------
                   Predicted:
                     positive negative
Actual: positive       6711      799
        negative        952     6538
```

Figure 7-9. *Model performance metrics for deep neural networks on word2vec features*

The results depicted in Figure 7-9 show us that we have obtained a model **accuracy** and **F1-score** of **88%**, which is great! You can use a similar workflow to build and train a DNN model for our GloVe based features and evaluate the model performance. The following snippet depicts the workflow for Steps 4 and 5 of our text classification system blueprint.

```
# build DNN model
glove_dnn = construct_deepnn_architecture(num_input_features=300)
# train DNN model on GloVe training features
batch_size = 100
glove_dnn.fit(train_glove_features, y_train, epochs=5, batch_size=batch_size,
              shuffle=True, validation_split=0.1, verbose=1)
# get predictions on test reviews
y_pred = glove_dnn.predict_classes(test_glove_features)
predictions = le.inverse_transform(y_pred)
# Evaluate model performance
meu.display_model_performance_metrics(true_labels=test_sentiments, predicted_
labels=predictions,
                                      classes=['positive', 'negative'])
```

We obtained an overall model **accuracy** and **F1-score** of **85%** with the GloVe features, which is still good but not better than what we obtained using our `word2vec` features. You can refer to the `Sentiment Analysis - Supervised.ipynb` jupyter notebook to see the step-by-step outputs obtained for the previous code. This concludes our discussion on building text sentiment classification systems leveraging newer Deep Learning models and methodologies. Onwards to learning about advanced Deep Learning models!

Advanced Supervised Deep Learning Models

We have used fully connected deep neural network and word embeddings in the previous section. Another new and interesting approach toward supervised Deep Learning is the use of recurrent neural networks (RNNs) and long short term memory networks (LSTMs) which also considers the sequence of data (words, events, and so on). These are more advanced models than your regular fully connected deep networks and usually take more time to train. We will leverage `keras` on top of `tensorflow` and try to build a LSTM-based classification model here and use word embeddings as our features. You can refer to the Python file titled `sentiment_analysis_adv_deep_learning.py` for all the code used in this section or use the jupyter notebook titled `Sentiment Analysis - Advanced Deep Learning.ipynb` for a more interactive experience.

We will be working on our normalized and pre-processed train and test review datasets, `norm_train_reviews` and `norm_test_reviews`, which we created in our previous analyses. Assuming you have them loaded up, we will first tokenize these datasets such that each text review is decomposed into its corresponding tokens (workflow Step 2).

```
In [1]: tokenized_train = [tn.tokenizer.tokenize(text) for text in norm_train_reviews]
    ...: tokenized_test = [tn.tokenizer.tokenize(text) for text in norm_test_reviews]
```

For feature engineering (Step 3), we will be creating word embeddings. However, we will create them ourselves using `keras` instead of using pre-built ones like `word2vec` or GloVe, which we used earlier. Word embeddings tend to vectorize text documents into fixed sized vectors such that these vectors try to capture contextual and semantic information.

For generating embeddings, we will use the `Embedding` layer from `keras`, which requires documents to be represented as tokenized and numeric vectors. We already have tokenized text vectors in our `tokenized_train` and `tokenized_text` variables. However we would need to convert them into numeric representations. Besides this, we would also need the vectors to be of uniform size even though the tokenized text reviews will be of variable length due to the difference in number of tokens in each review. For this, one strategy could be to take the length of the longest review (with maximum number of tokens\words) and set it as the vector size, let's call this `max_len`. Reviews of shorter length can be padded with a `PAD` term in the beginning to increase their length to `max_len`.

We would need to create a word to index vocabulary mapping for representing each tokenized text review in a numeric form. Do note you would also need to create a numeric mapping for the padding term which we shall call `PAD_INDEX` and assign it the numeric index of `0`. For unknown terms, in case they are encountered later on in the test dataset or newer, previously unseen reviews, we would need to assign it to some index too. This would be because we will vectorize, engineer features, and build models only on the training data. Hence, if some new term should come up in the future (which was originally not a part of the model training), we will consider it as an out of vocabulary ***(OOV)*** term and assign it to a constant index (we will name this term `NOT_FOUND_INDEX` and assign it the index of `vocab_size+1`). The following snippet helps us create this vocabulary from our `tokenized_train` corpus of training text reviews.

```
In [2]: from collections import Counter
   ...:
   ...: # build word to index vocabulary
   ...: token_counter = Counter([token for review in tokenized_train for token in review])
   ...: vocab_map = {item[0]: index+1
                     for index, item in enumerate(dict(token_counter).items())}
   ...: max_index = np.max(list(vocab_map.values()))
   ...: vocab_map['PAD_INDEX'] = 0
   ...: vocab_map['NOT_FOUND_INDEX'] = max_index+1
   ...: vocab_size = len(vocab_map)
   ...: # view vocabulary size and part of the vocabulary map
   ...: print('Vocabulary Size:', vocab_size)
   ...: print('Sample slice of vocabulary map:', dict(list(vocab_map.items())[10:20]))
Vocabulary Size: 82358
Sample slice of vocabulary map: {'martyrdom': 6, 'palmira': 7, 'servility': 8, 'gardening':
9, 'melodramatically': 73505, 'renfro': 41282, 'carlin': 41283, 'overtly': 41284, 'rend':
47891, 'anticlimactic': 51}
```

In this case we have used all the terms in our vocabulary, you can easily filter and use more relevant terms here (based on their frequency) by using the `most_common(count)` function from `Counter` and taking the first `count` terms from the list of unique terms in the training corpus. We will now encode the tokenized text reviews based on the previous `vocab_map`. Besides this, we will also encode the text sentiment class labels into numeric representations.

```
In [3]: from keras.preprocessing import sequence
   ...: from sklearn.preprocessing import LabelEncoder
   ...:
   ...: # get max length of train corpus and initialize label encoder
   ...: le = LabelEncoder()
   ...: num_classes=2 # positive -> 1, negative -> 0
   ...: max_len = np.max([len(review) for review in tokenized_train])
   ...:
   ...: ## Train reviews data corpus
   ...: # Convert tokenized text reviews to numeric vectors
   ...: train_X = [[vocab_map[token] for token in tokenized_review]
                    for tokenized_review in tokenized_train]
   ...: train_X = sequence.pad_sequences(train_X, maxlen=max_len) # pad
   ...: ## Train prediction class labels
   ...: # Convert text sentiment labels (negative\positive) to binary encodings (0/1)
   ...: train_y = le.fit_transform(train_sentiments)
   ...:
   ...: ## Test reviews data corpus
   ...: # Convert tokenized text reviews to numeric vectors
   ...: test_X = [[vocab_map[token] if vocab_map.get(token) else vocab_map['NOT_FOUND_INDEX']
   ...:            for token in tokenized_review]
   ...:               for tokenized_review in tokenized_test]
   ...: test_X = sequence.pad_sequences(test_X, maxlen=max_len)
```

```
   ...: ## Test prediction class labels
   ...: # Convert text sentiment labels (negative\positive) to binary encodings (0/1)
   ...: test_y = le.transform(test_sentiments)
   ...:
   ...: # view vector shapes
   ...: print('Max length of train review vectors:', max_len)
   ...: print('Train review vectors shape:', train_X.shape,
              ' Test review vectors shape:', test_X.shape)

Max length of train review vectors: 1442
Train review vectors shape: (35000, 1442)  Test review vectors shape: (15000, 1442)
```

From the preceding code snippet and the output, it is clear that we encoded each text review into a numeric sequence vector so that the size of each review vector is `1442`, which is basically the maximum length of reviews from the training dataset. We pad shorter reviews and truncate extra tokens from longer reviews such that the shape of each review is constant as depicted in the output. We can now proceed with Step 3 and a part of Step 4 of the classification workflow by introducing the `Embedding` layer and coupling it with the deep network architecture based on LSTMs.

```
from keras.models import Sequential
from keras.layers import Dense, Embedding, Dropout, SpatialDropout1D
from keras.layers import LSTM

EMBEDDING_DIM = 128 # dimension for dense embeddings for each token
LSTM_DIM = 64 # total LSTM units

model = Sequential()
model.add(Embedding(input_dim=vocab_size, output_dim=EMBEDDING_DIM, input_length=max_len))
model.add(SpatialDropout1D(0.2))
model.add(LSTM(LSTM_DIM, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation="sigmoid"))

model.compile(loss="binary_crossentropy", optimizer="adam",
              metrics=["accuracy"])
```

The `Embedding` layer helps us generate the word embeddings from scratch. This layer is also initialized with some weights initially and this gets updated based on our optimizer similar to weights on the neuron units in other layers when the network tries to minimize the loss in each epoch. Thus, the embedding layer tries to optimize its weights such that we get the best word embeddings which will generate minimum error in the model and also capture semantic similarity and relationships among words. How do we get the embeddings, let's consider we have a review with `3` terms `['movie', 'was', 'good']` and a `vocab_map` consisting of word to index mappings for `82358` words. The word embeddings would be generated somewhat similar to Figure 7-10.

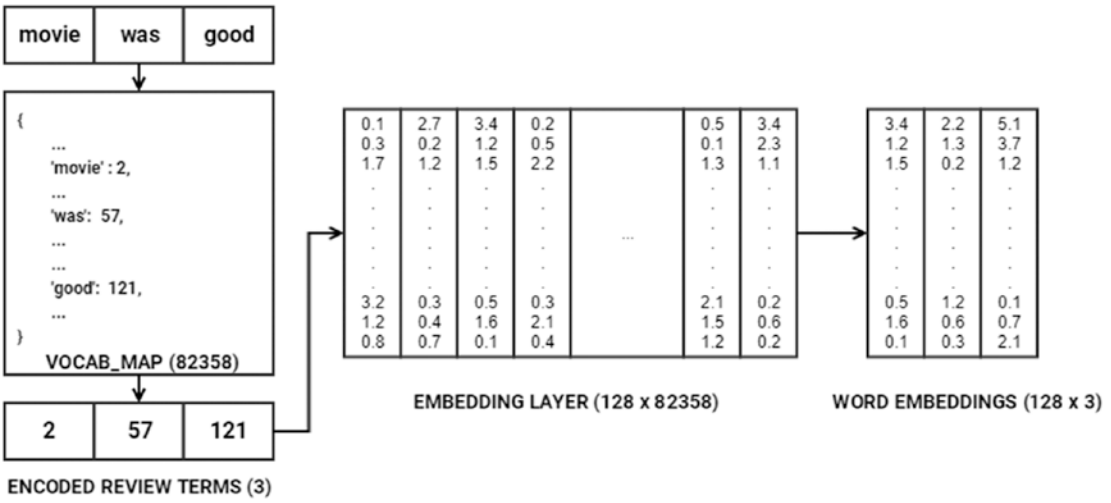

Figure 7-10. *Understanding how word embeddings are generated*

Based on our model architecture, the `Embedding` layer takes in three parameters—`input_dim`, which is equal to the vocabulary size (`vocab_size`) of `82358`, `output_dim`, which is `128`, representing the dimension of dense embedding (depicted by rows in the `EMBEDDING LAYER` in Figure 7-10), and `input_len`, which specifies the length of the input sequences (movie review sequence vectors), which is `1442`. In the example depicted in Figure 7-10, since we have one review, the dimension is `(1, 3)`. This review is converted into a numeric sequence `[2, 57, 121]` based on the `VOCAB_MAP`. Then the specific columns representing the indices in the review sequence are selected from the `EMBEDDING LAYER` (vectors at column indices `2`, `57` and `121` respectively), to generate the final word embeddings. This gives us an embedding vector of dimension `(1, 128, 3)` also represented as `(1, 3, 128)` when each row is represented based on each sequence word embedding vector. Many Deep Learning frameworks like `keras` represent the embedding dimensions as (m, n) where m represents all the unique terms in our vocabulary (`82358`) and n represents the `output_dim` which is `128` in this case. Consider a transposed version of the layer depicted in Figure 7-10 and you are good to go!

Usually if you have the encoded review terms sequence vector represented in one-hot encoded format `(3, 82358)` and do a matrix multiplication with the `EMBEDDING LAYER` represented as `(82358, 128)` where each row represents the embedding for a word in the vocabulary, you will directly obtain the word embeddings for the review sequence vector as `(3, 128)`. The weights in the embedding layer get updated and optimized in each epoch based on the input data when propagated through the whole network like we mentioned earlier such that overall loss and error is minimized to get maximum model performance.

These dense word embeddings are then passed to the `LSTM` layer having `64` units. We already introduced you to the LSTM architecture briefly in Chapter 1 in the subsection titled "Long Short Term Memory Networks" in the "Important Concepts" section under "Deep Learning". LSTMs basically try to overcome the shortcomings of RNN models especially with regard to handling long term dependencies and problems which occur when the weight matrix associated with the units (neurons) become too small (leading to vanishing gradient) or too large (leading to exploding gradient). These architectures are more complex than regular deep networks and going into detailed internals and math concepts would be out of the current scope, but we will try to cover the essentials here without making it math heavy. If you're interested in researching the internals of LSTMs, check out the original paper which inspired it all, by Hochreiter, S., and Schmidhuber, J. (1997). *Long short-term memory. Neural computation. 9(8), 1735-1780*. We depict the basic architecture of RNNs and compare it with LSTMs in Figure 7-11.

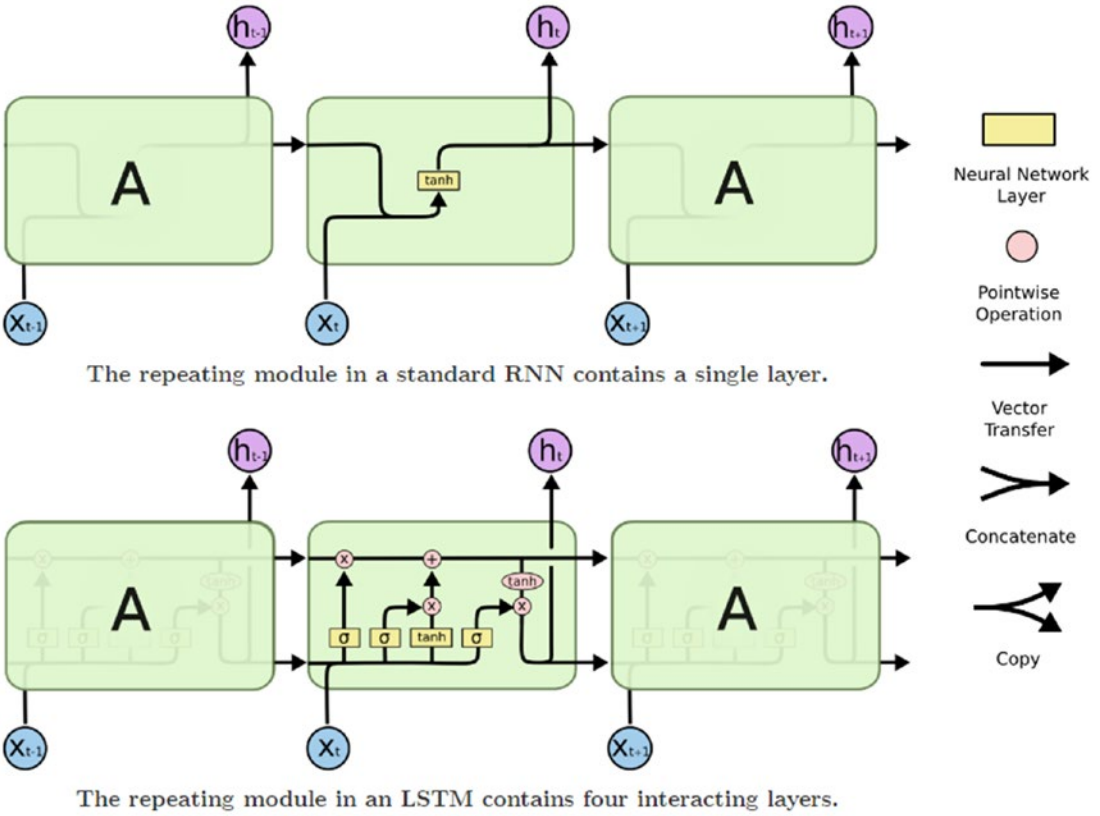

Figure 7-11. *Basic structure of RNN and LSTM units (Source: Christopher Olah's blog: colah.github.io)*

The RNN units usually have a chain of repeating modules (this happens when we unroll the loop; refer to Figure 1-13 in Chapter 1, where we talk about this) such that the module has a simple structure of having maybe one layer with the `tanh` activation. LSTMs are also a special type of RNN, having a similar structure but the LSTM unit has four neural network layers instead of just one. The detailed architecture of the LSTM cell is shown in Figure 7-12.

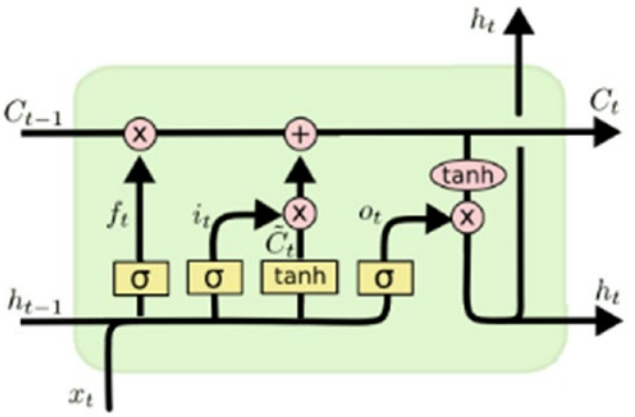

Figure 7-12. *Detailed architecture of an LSTM cell (Source: Christopher Olah's blog: colah.github.io)*

The detailed architecture of an LSTM cell is depicted in Figure 7-12. The notation *t* indicates one time step, C depicts the cell states, and *h* indicates the hidden states. The gates $i, f, o\,\breve{\mathrm{C}}$ help in removing or adding information to the cell state. The gates *i, f* & *o* represent the *input*, *output* and *forget* gates respectively and each of them are modulated by the sigmoid layer which outputs numbers from 0 to 1 controlling how much of the output from these gates should pass. Thus this helps is protecting and controlling the cell state. Detailed work flow of how information flows through the LSTM cell is depicted in Figure 7-13 in four steps.

1. The first step talks about the forget gate layer *f*, which helps us decide what information should we throw away from the cell state. This is done by looking at the previous hidden state h_{t-1} and current inputs x_t as depicted in the equation.

   The sigmoid layer helps control how much of this should be kept or forgotten.

2. The second step depicts the input gate layer *t*, which helps decide what information will be stored in the current cell state. The sigmoid layer in the input gate helps decide which values will be updated based on $h_{t-1}\,x_t$ again. The tanh layer helps create a vector of the new candidate values $\breve{\mathrm{C}}_t$ based on $h_{t-1}\,x_t$, which can be added to the current cell state. Thus the tanh layer creates the values and the input gate with sigmoid layer helps choose which values should be updated.

3. The third step involves updating the old cell state C_{t-1} to the new cell state C_t by leveraging what we obtained in the first two steps. We multiply the old cell state by the forget gate $\left(f_t \times C_{t-1}\right)$ and then add the new candidate values scaled by the input gate with sigmoid layer $\left(i_t \times \breve{\mathrm{C}}_t\right)$.

4. The fourth and final step helps us decide what should be the final output which is basically a filtered version of our cell state. The output gate with the sigmoid layer *o* helps us select which parts of the cell state will pass to the final output. This is multiplied with the cell state values when passed through the tanh layer to give us the final hidden state values $h_t = o_t \times \tanh\left(\breve{\mathrm{C}}_t\right)$.

All these steps in this detailed workflow are depicted in Figure 7-13 with necessary annotations and equations. We would like to thank our good friend Christopher Olah for providing us detailed information as well as the images for depicting the internal workings of LSTM networks. We recommend checking out Christopher's blog at http://colah.github.io/posts/2015-08-Understanding-LSTMs for more details. A shout out also goes to Edwin Chen, for explaining RNNs and LSTMs in an easy-to-understand format. We recommend referring to Edwin's blog at http://blog.echen.me/2017/05/30/exploring-lstms for information on the workings of RNNs and LSTMs.

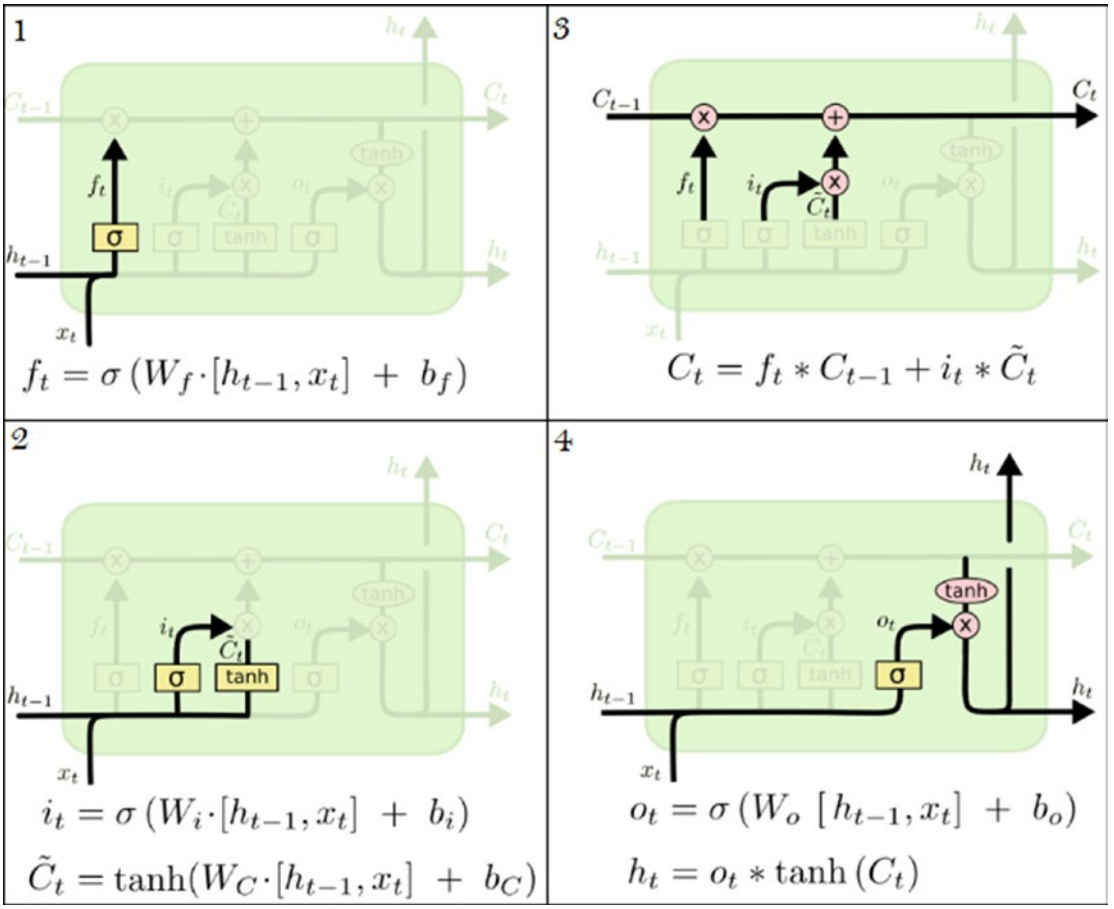

Figure 7-13. *Walkthrough of data flow in an LSTM cell (Source: Christopher Olah's blog: colah.github.io)*

The final layer in our deep network is the `Dense` layer with `1` unit and the sigmoid activation function. We basically use the `binary_crossentropy` function with the `adam` optimizer since this is a binary classification problem and the model will ultimately predict a `0` or a `1`, which we can decode back to a negative or positive sentiment prediction with our label encoder. You can also use the `categorical_crossentropy` loss function here, but you would need to then use a `Dense` layer with `2` units instead with a `softmax` function. Now that our model is compiled and ready, we can head on to Step 4 of our classification workflow of actually training the model. We use a similar strategy from our previous deep network models, where we train our model on the training data with five epochs, batch size of 100 reviews, and a 10% validation split of training data to measure validation accuracy.

```
In [4]: batch_size = 100
   ...: model.fit(train_X, train_y, epochs=5, batch_size=batch_size,
   ...:           shuffle=True, validation_split=0.1, verbose=1)
Train on 31500 samples, validate on 3500 samples
Epoch 1/5 31500/31500 - 2491s - loss: 0.4081 - acc: 0.8184 - val_loss: 0.3006 - val_acc:
0.8751
Epoch 2/5 31500/31500 - 2489s - loss: 0.2253 - acc: 0.9158 - val_loss: 0.3209 - val_acc:
0.8780
Epoch 3/5 31500/31500 - 2656s - loss: 0.1431 - acc: 0.9493 - val_loss: 0.3483 - val_acc:
0.8671
Epoch 4/5 31500/31500 - 2604s - loss: 0.1023 - acc: 0.9658 - val_loss: 0.3803 - val_acc:
0.8729
Epoch 5/5 31500/31500 - 2701s - loss: 0.0694 - acc: 0.9761 - val_loss: 0.4430 - val_acc:
0.8706
```

Training LSTMs on CPU is notoriously slow and as you can see my model took approximately 3.6 hours to train for just five epochs on an i5 3rd Gen Intel CPU with 8 GB of memory. Of course, a cloud-based environment like Google Cloud Platform or AWS on GPU took me approximately less than an hour to train the same model. So I would recommend you choose a GPU based Deep Learning environment, especially when working with RNNs or LSTM based network architectures. Based on the preceding output, we can see that just with five epochs we have decent validation accuracy but the training accuracy starts shooting up indicating some over-fitting might be happening. Ways to overcome this include adding more data or by increasing the drouput rate. Do give it a shot and see if it works! Time to put our model to the test! Let's see how well it predicts the sentiment for our test reviews and use the same model evaluation framework we have used for our previous models (Step 5).

```
In [5]: # predict sentiments on test data
   ...: pred_test = model.predict_classes(test_X)
   ...: predictions = le.inverse_transform(pred_test.flatten())
   ...: # evaluate model performance
   ...: meu.display_model_performance_metrics(true_labels=test_sentiments,
   ...:                    predicted_labels=predictions, classes=['positive', 'negative'])
```

```
Model Performance metrics:
------------------------------
Accuracy: 0.88
Precision: 0.88
Recall: 0.88
F1 Score: 0.88

Model Classification report:
------------------------------
                precision    recall  f1-score   support

positive           0.88      0.89      0.88      7510
negative           0.89      0.87      0.88      7490

avg / total        0.88      0.88      0.88     15000

Prediction Confusion Matrix:
------------------------------
                 Predicted:
                   positive negative
Actual: positive       6711      799
        negative        952     6538
```

Figure 7-14. *Model performance metrics for LSTM based Deep Learning model on word embeddings*

The results depicted in Figure 7-14 show us that we have obtained a model **accuracy** and **F1-score** of **88%**, which is quite good! With more quality data, you can expect to get even better results. Try experimenting with different architectures and see if you get better results!

Analyzing Sentiment Causation

We built both supervised and unsupervised models to predict the sentiment of movie reviews based on the review text content. While feature engineering and modeling is definitely the need of the hour, you also need to know how to analyze and interpret the root cause behind how model predictions work. In this section, we analyze sentiment causation. The idea is to determine the root cause or key factors causing positive or negative sentiment. The first area of focus will be model interpretation, where we will try to understand, interpret, and explain the mechanics behind predictions made by our classification models. The second area of focus is to apply topic modeling and extract key topics from positive and negative sentiment reviews.

Interpreting Predictive Models

One of the challenges with Machine Learning models is the transition from a pilot or proof-of-concept phase to the production phase. Business and key stakeholders often perceive Machine Learning models as complex black boxes and poses the question, why should I trust your model? Explaining to them complex mathematical or theoretical concepts doesn't serve the purpose. Is there some way in which we can explain these models in an easy-to-interpret manner? This topic in fact has gained extensive attention very recently in 2016. Refer to the original research paper by M.T. Ribeiro, S. Singh & C. Guestrin titled "Why Should I Trust You?: Explaining the Predictions of Any Classifier" from https://arxiv.org/pdf/1602.04938.pdf to understand more about model interpretation and the LIME framework. Check out more on model interpretation in Chapter 5 where we cover the skater framework in detail which performs excellent interpretations of various models.

There are various ways to interpret the predictions made by our predictive sentiment classification models. We want to understand more into why a positive review was correctly predicted as having positive sentiment or a negative review having negative sentiment. Besides this, no model is a 100% accurate always, so we would also want to understand the reason for mis-classifications or wrong predictions. The code used in this section is available in the file named `sentiment_causal_model_interpretation.py` or you can also refer to the jupyter notebook named `Sentiment Causal Analysis - Model Interpretation.ipynb` for an interactive experience.

Let's first build a basic text classification pipeline for the model that worked best for us so far. This is the Logistic Regression model based on the Bag of Words feature model. We will leverage the pipeline module from `scikit-learn` to build this Machine Learning pipeline using the following code.

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline

# build BOW features on train reviews
cv = CountVectorizer(binary=False, min_df=0.0, max_df=1.0, ngram_range=(1,2))
cv_train_features = cv.fit_transform(norm_train_reviews)
# build Logistic Regression model
lr = LogisticRegression()
lr.fit(cv_train_features, train_sentiments)

# Build Text Classification Pipeline
lr_pipeline = make_pipeline(cv, lr)

# save the list of prediction classes (positive, negative)
classes = list(lr_pipeline.classes_)
```

We build our model based on `norm_train_reviews`, which contains the normalized training reviews that we have used in all our earlier analyses. Now that we have our classification pipeline ready, you can actually deploy the model by using pickle or joblib to save the classifier and feature objects similar to what we discussed in the "Model Deployment" section in Chapter 5. Assuming our pipeline is in production, how do we use it for new movie reviews? Let's try to predict the sentiment for two new sample reviews (which were not used in training the model).

```
In [3]: lr_pipeline.predict(['the lord of the rings is an excellent movie',
   ...:                      'i hated the recent movie on tv, it was so bad'])
Out[3]: array(['positive', 'negative'], dtype=object)
```

Our classification pipeline predicts the sentiment of both the reviews correctly! This is a good start, but how do we interpret the model predictions? One way is to typically use the model prediction class probabilities as a measure of confidence. You can use the following code to get the prediction probabilities for our sample reviews.

```
In [4]: pd.DataFrame(lr_pipeline.predict_proba(['the lord of the rings is an excellent movie',
   ...:                       'i hated the recent movie on tv, it was so bad']),
columns=classes)
Out[4]:
   negative  positive
0  0.169653  0.830347
1  0.730814  0.269186
```

Thus we can say that the first movie review has a prediction confidence or probability of 83% to have positive sentiment as compared to the second movie review with a 73% probability to have negative sentiment. Let's now kick it up a notch, instead of playing around with toy examples, we will now run the same analysis on actual reviews from the `test_reviews` dataset (we will use `norm_test_reviews`, which has the normalized text reviews). Besides prediction probabilities, we will be using the `skater` framework for easy interpretation of the model decisions, similar to what we have done in Chapter 5 under the section "Model Interpretation". You need to load the following dependencies from the `skater` package first. We also define a helper function which takes in a document index, a corpus, its response predictions, and an explainer object and helps us with the our model interpretation analysis.

```
from skater.core.local_interpretation.lime.lime_text import LimeTextExplainer

explainer = LimeTextExplainer(class_names=classes)
# helper function for model interpretation
def interpret_classification_model_prediction(doc_index, norm_corpus, corpus,
                                              prediction_labels, explainer_obj):
    # display model prediction and actual sentiments
    print("Test document index: {index}\nActual sentiment: {actual}
                                      \nPredicted sentiment: {predicted}"
      .format(index=doc_index, actual=prediction_labels[doc_index],
              predicted=lr_pipeline.predict([norm_corpus[doc_index]])))
    # display actual review content
    print("\nReview:", corpus[doc_index])
    # display prediction probabilities
    print("\nModel Prediction Probabilities:")
    for probs in zip(classes, lr_pipeline.predict_proba([norm_corpus[doc_index]])[0]):
        print(probs)
```

```
    # display model prediction interpretation
    exp = explainer.explain_instance(norm_corpus[doc_index],
                                     lr_pipeline.predict_proba, num_features=10,
                                     labels=[1])
    exp.show_in_notebook()
```

The preceding snippet leverages `skater` to explain our text classifier to analyze its decision-making process in an easy to interpret form. Even though the model might be a complex one in a global perspective, it is easier to explain and approximate the model behavior on local instances. This is done by learning the model around the vicinity of the data point of interest *X* by sampling instances around *X* and assigning weightages based on their proximity to*X*. Thus, these locally learned linear models help in explaining complex models in a more easy to interpret way with class probabilities, contribution of top features to the class probabilities that aid in the decision making process. Let's take a movie review from our test dataset where both the actual and predicted sentiment is negative and analyze it with the helper function we created in the preceding snippet.

```
In [6]: doc_index = 100
   ...: interpret_classification_model_prediction(doc_index=doc_index, corpus=norm_test_
reviews,
                                                  corpus=test_reviews, prediction_labels=test_
                                                  sentiments,
                                                  explainer_obj=explainer)

Test document index: 100
Actual sentiment: negative
Predicted sentiment: ['negative']

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

Model Prediction Probabilities:
('negative', 0.8099323456145181)
('positive', 0.19006765438548187)
```

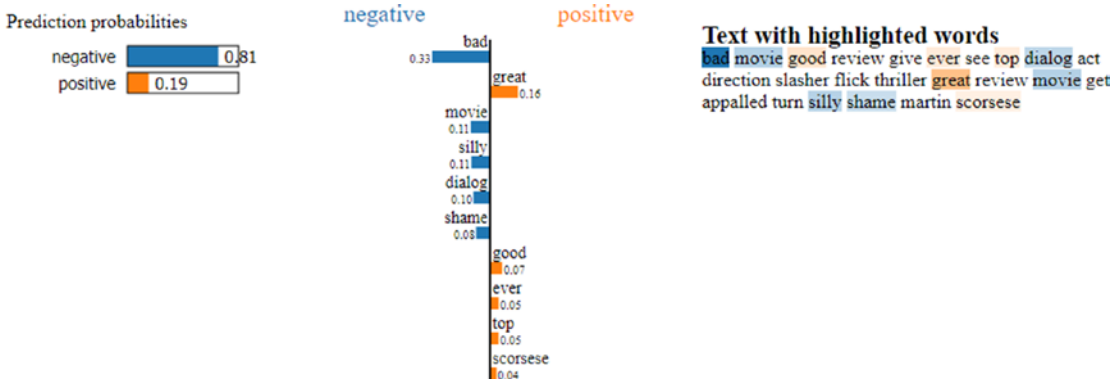

Figure 7-15. *Model interpretation for our classification model's correct prediction for a negative review*

The results depicted in Figure 7-15 show us the class prediction probabilities and also the top 10 features that contributed the maximum to the prediction decision making process. These key features are also highlighted in the normalized movie review text. Our model performs quite well in this scenario and we can see the key features that contributed to the negative sentiment of this review including `bad`, `silly`, `dialog`, and `shame`, which make sense. Besides this, the word `great` contributed the maximum to the positive probability of 0.19 and in fact if we had removed this word from our review text, the positive probability would have dropped significantly.

The following code runs a similar analysis on a test movie review with both actual and predicted sentiment of positive value.

```
In [7]: doc_index = 2000
   ...: interpret_classification_model_prediction(doc_index=doc_index, corpus=norm_test_
reviews,
                                                  corpus=test_reviews, prediction_labels=test_
                                                  sentiments,
                                                  explainer_obj=explainer)

Test document index: 2000
Actual sentiment: positive
Predicted sentiment: ['positive']

Review: I really liked the Movie "JOE." It has really become a cult classic among
certain age groups.<br /><br />The Producer of this movie is a personal friend of mine.
He is my Stepsons Father-In-Law. He lives in Manhattan's West side, and has a Bungalow.
in Southampton, Long Island. His son-in-law live next door to his Bungalow.<br /><br
/>Presently, he does not do any Producing, But dabbles in a business with HBO movies.<br
/><br />As a person, Mr. Gil is a real gentleman and I wish he would have continued in the
production business of move making.

Model Prediction Probabilities:
('negative', 0.020629181561415355)
('positive', 0.97937081843858464)
```

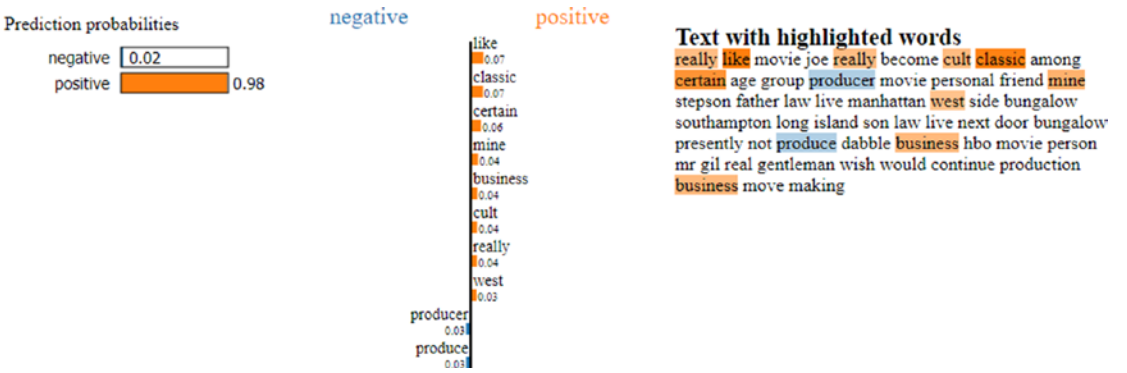

Figure 7-16. *Model interpretation for our classification model's correct prediction for a positive review*

The results depicted in Figure 7-16 show the top features responsible for the model making a decision of predicting this review as positive. Based on the content, the reviewer really liked this model and also it was a real cult classic among certain age groups. In our final analysis, we will look at the model interpretation of an example where the model makes a wrong prediction.

```
In [8]: doc_index = 347
   ...: interpret_classification_model_prediction(doc_index=doc_index, corpus=norm_test_
reviews,
                                                  corpus=test_reviews, prediction_labels=test_
sentiments,
                                                  explainer_obj=explainer)

Test document index: 347
Actual sentiment: negative
Predicted sentiment: ['positive']

Review: When I first saw this film in cinema 11 years ago, I loved it. I still think the
directing and cinematography are excellent, as is the music. But it's really the script that
has over the time started to bother me more and more. I find Emma Thompson's writing self-
absorbed and unfaithful to the original book; she has reduced Marianne to a side-character,
a second fiddle to her much too old, much too severe Elinor - she in the movie is given
many sort of 'focus moments', and often they appear to be there just to show off Thompson
herself.<br /><br />I do understand her cutting off several characters from the book, but
leaving out the one scene where Willoughby in the book is redeemed? For someone who red
and cherished the book long before the movie, those are the things always difficult to
digest.<br /><br />As for the actors, I love Kate Winslet as Marianne. She is not given the
best script in the world to work with but she still pulls it up gracefully, without too much
sentimentality. Alan Rickman is great, a bit old perhaps, but he plays the role beautifully.
And Elizabeth Spriggs, she is absolutely fantastic as always.

Model Prediction Probabilities:
('negative', 0.067198213044844413)
('positive', 0.93280178695515559)
```

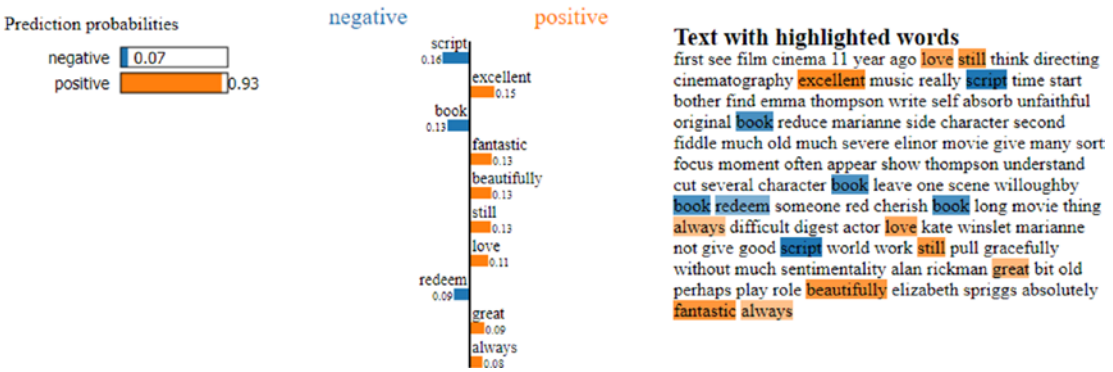

Figure 7-17. *Model interpretation for our classification model's incorrect prediction*

The preceding output tells us that our model predicted the movie review indicating a positive sentiment when in-fact the actual sentiment label is negative for the same review. The results depicted in Figure 7-17 tell us that the reviewer in fact shows signs of positive sentiment in the movie review, especially in parts where he\she tells us that "I loved it. I still think the directing and cinematography are excellent, as is the music... Alan Rickman is great, a bit old perhaps, but he plays the role beautifully. And Elizabeth Spriggs, she is absolutely fantastic as always." and feature words from the same have been depicted in the top features contributing to positive sentiment. The model interpretation also correctly identifies the aspects of the review contributing to negative sentiment like, "But it's really the script that has over the time started to bother me more and more.". Hence, this is one of the more complex reviews which indicate both positive and negative sentiment and the final interpretation would be in the reader's hands. You can now use this same framework to interpret your own classification models in the future and understand where your model might be performing well and where it might need improvements!

Analyzing Topic Models

Another way of analyzing key terms, concepts or topics responsible for sentiment is to use a different approach known as topic modeling. We have already covered some basics into topic modeling in the section titled "Topic Models" under "Feature Engineering on Text Data" in Chapter 4. The main aim of topic models is to extract and depict key topics or concepts which are otherwise latent and not very prominent in huge corpora of text documents. We have already seen the use of Latent Dirichlet Allocation (LDA) for topic modeling in Chapter 4. In this section, we use another topic modeling technique called Non-Negative Matrix factorization. Refer to the Python file named `sentiment_causal_topic_models.py` or the jupyter notebook titled `Sentiment Causal Analysis - Topic Models.ipynb` for a more interactive experience.

The first step in this analysis is to combine all our normalized `train` and `test` reviews and separate out these reviews into positive and negative sentiment reviews. Once we do this, we will extract features from these two datasets using the TF-IDF feature vectorizer. The following snippet helps us achieve this.

```
In [11]: from sklearn.feature_extraction.text import TfidfVectorizer
    ...:
    ...: # consolidate all normalized reviews
    ...: norm_reviews = norm_train_reviews+norm_test_reviews
    ...: # get tf-idf features for only positive reviews
    ...: positive_reviews = [review for review, sentiment in zip(norm_reviews, sentiments)
                             if sentiment == 'positive']
    ...: ptvf = TfidfVectorizer(use_idf=True, min_df=0.05, max_df=0.95,
                            ngram_range=(1,1), sublinear_tf=True)
    ...: ptvf_features = ptvf.fit_transform(positive_reviews)
    ...: # get tf-idf features for only negative reviews
    ...: negative_reviews = [review for review, sentiment in zip(norm_reviews, sentiments)
                             if sentiment == 'negative']
    ...: ntvf = TfidfVectorizer(use_idf=True, min_df=0.05, max_df=0.95,
                            ngram_range=(1,1), sublinear_tf=True)
    ...: ntvf_features = ntvf.fit_transform(negative_reviews)
    ...: # view feature set dimensions
    ...: print(ptvf_features.shape, ntvf_features.shape)

(25000, 331) (25000, 331)
```

From the preceding output dimensions, you can see that we have filtered out a lot of the features we used previously when building our classification models by making `min_df` to be `0.05` and `max_df` to be `0.95`. This is to speed up the topic modeling process and remove features that either occur too much or too rarely. Let's now import the necessary dependencies for the topic modeling process.

```
In [12]: import pyLDAvis
    ...: import pyLDAvis.sklearn
    ...: from sklearn.decomposition import NMF
    ...: import topic_model_utils as tmu
    ...:
    ...: pyLDAvis.enable_notebook()
    ...: total_topics = 10
```

The `NMF` class from `scikit-learn` will help us with topic modeling. We also use `pyLDAvis` for building interactive visualizations of topic models. The core principle behind Non-Negative Matrix Factorization (NNMF) is to apply matrix decomposition (similar to SVD) to a non-negative feature matrix X such that the decomposition can be represented as $X \approx WH$ where $W \& H$ are both non-negative matrices which if multiplied should approximately re-construct the feature matrix X. A cost function like L2 norm can be used for getting this approximation. Let's now apply NNMF to get 10 topics from our positive sentiment reviews. We will also leverage some utility functions from our `topic_model_utils` module to display the results in a clean format.

```
In [13]: # build topic model on positive sentiment review features
    ...: pos_nmf = NMF(n_components=total_topics,
    ...:           random_state=42, alpha=0.1, l1_ratio=0.2)
    ...: pos_nmf.fit(ptvf_features)
    ...: # extract features and component weights
    ...: pos_feature_names = ptvf.get_feature_names()
    ...: pos_weights = pos_nmf.components_
    ...: # extract and display topics and their components
    ...: pos_topics = tmu.get_topics_terms_weights(pos_weights, pos_feature_names)
    ...: tmu.print_topics_udf(topics=pos_topics, total_topics=total_topics,
    ...:                   num_terms=15, display_weights=False)
Topic #1 without weights
['like', 'not', 'think', 'really', 'say', 'would', 'get', 'know', 'thing', 'much', 'bad',
'go', 'lot', 'could', 'even']

Topic #2 without weights
['movie', 'see', 'watch', 'great', 'good', 'one', 'not', 'time', 'ever', 'enjoy',
'recommend', 'make', 'acting', 'like', 'first']

Topic #3 without weights
['show', 'episode', 'series', 'tv', 'watch', 'dvd', 'first', 'see', 'time', 'one', 'good',
'year', 'remember', 'ever', 'would']

Topic #4 without weights
['performance', 'role', 'play', 'actor', 'cast', 'good', 'well', 'great', 'character',
'excellent', 'give', 'also', 'support', 'star', 'job']
...
Topic #10 without weights
['love', 'fall', 'song', 'wonderful', 'beautiful', 'music', 'heart', 'girl', 'would',
'watch', 'great', 'favorite', 'always', 'family', 'woman']
```

We depict some of the topics out of the 10 topics generated in the preceding output. You can leverage `pyLDAvis` now to visualize these topics in an interactive visualization. See Figure 7-18.

```
In [14]: pyLDAvis.sklearn.prepare(pos_nmf, ptvf_features, ptvf, R=15)
```

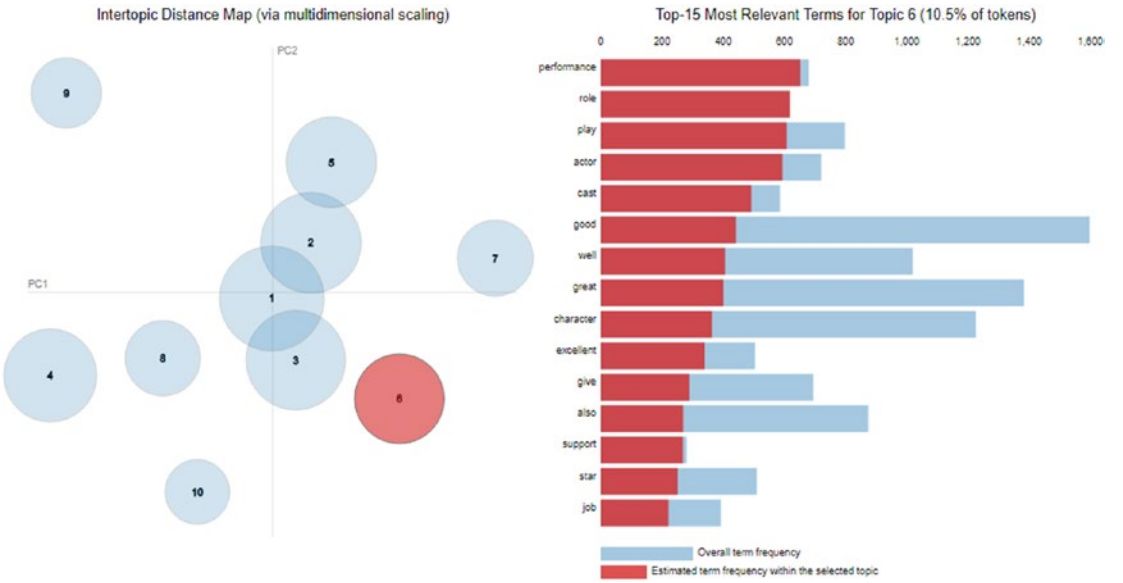

Figure 7-18. *Visualizing topic models on positive sentiment movie reviews*

The visualization depicted in Figure 7-18 shows us the 10 topics from positive movie reviews and we can see the top relevant terms for Topic 6 highlighted in the output. From the topics and the terms, we can see terms like *movie cast, actors, performance, play, characters, music, wonderful, good,* and so on have contributed toward positive sentiment in various topics. This is quite interesting and gives you a good insight into the components of the reviews that contribute toward positive sentiment of the reviews. This visualization is completely interactive if you are using the jupyter notebook and you can click on any of the bubbles representing topics in the Intertopic Distance Map on the left and see the most relevant terms in each of the topics in the right bar chart.

The plot on the left is rendered using Multi-dimensional Scaling (MDS). Similar topics should be close to one another and dissimilar topics should be far apart. The size of each topic bubble is based on the frequency of that topic and its components in the overall corpus.

The visualization on the right shows the top terms. When no topic it selected, it shows the top 15 most salient topics in the corpus. A term's saliency is defined as a measure of how frequently the term appears the corpus and its distinguishing factor when used to distinguish between topics. When some topic is selected, the chart changes to show something similar to Figure 7-13, which shows the top 15 most relevant terms for that topic. The relevancy metric is controlled by λ, which can be changed based on a slider on top of the bar chart (refer to the notebook to interact with this). If you're interested in more mathematical theory behind these visualizations, you are encouraged to check out more details at `https://cran.r-project.org/web/packages/LDAvis/vignettes/details.pdf`, which is a vignette for the R package `LDAvis`, which has been ported to Python as `pyLDAvis`.

Let's now extract topics and run this same analysis on our negative sentiment reviews from the movie reviews dataset.

```
In [15]: # build topic model on negative sentiment review features
    ...: neg_nmf = NMF(n_components=10,
    ...:           random_state=42, alpha=0.1, l1_ratio=0.2)
    ...: neg_nmf.fit(ntvf_features)
    ...: # extract features and component weights
    ...: neg_feature_names = ntvf.get_feature_names()
    ...: neg_weights = neg_nmf.components_
    ...: # extract and display topics and their components
    ...: neg_topics = tmu.get_topics_terms_weights(neg_weights, neg_feature_names)
    ...: tmu.print_topics_udf(topics=neg_topics,
    ...:                  total_topics=total_topics,
    ...:                  num_terms=15,
    ...:                  display_weights=False)
Topic #1 without weights
['get', 'go', 'kill', 'guy', 'scene', 'take', 'end', 'back', 'start', 'around', 'look',
'one', 'thing', 'come', 'first']

Topic #2 without weights
['bad', 'movie', 'ever', 'acting', 'see', 'terrible', 'one', 'plot', 'effect', 'awful',
'not', 'even', 'make', 'horrible', 'special']
...
Topic #10 without weights
['waste', 'time', 'money', 'watch', 'minute', 'hour', 'movie', 'spend', 'not', 'life',
'save', 'even', 'worth', 'back', 'crap']

In [16]: pyLDAvis.sklearn.prepare(neg_nmf, ntvf_features, ntvf, R=15)
```

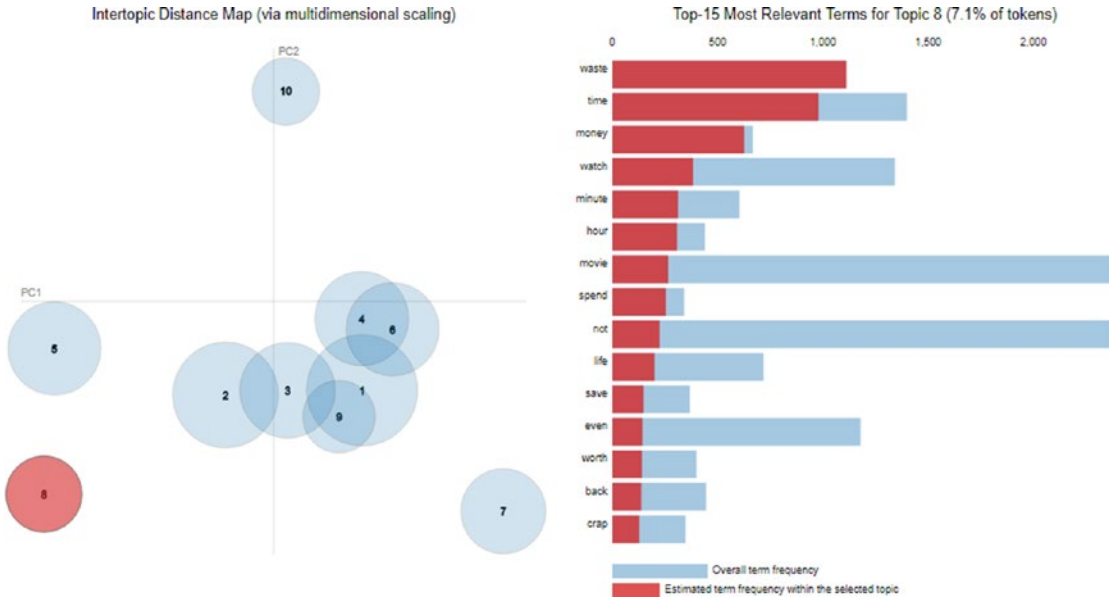

Figure 7-19. *Visualizing topic models on positive sentiment movie reviews*

The visualization depicted in Figure 7-19 shows us the 10 topics from negative movie reviews and we can see the top relevant terms for Topic 8 highlighted in the output. From the topics and the terms, we can see terms like *waste, time, money, crap, plot, terrible, acting,* and so on have contributed toward negative sentiment in various topics. Of course, there are high chances of overlap between topics from positive and negative sentiment reviews, but there will be distinguishable, distinct topics that further help us with interpretation and causal analysis.

Summary

This case-study oriented chapter introduces the IMDb movie review dataset with the objective of predicting the sentiment of the reviews based on the textual content. We covered concepts and techniques from natural language processing (NLP), text analytics, Machine Learning and Deep Learning in this chapter. We covered multiple aspects from NLP including text pre-processing, normalization, feature engineering as well as text classification. Unsupervised learning techniques using sentiment lexicons like Afinn, SentiWordNet, and VADER were covered in extensive detail, to show how we can analyze sentiment in the absence of labeled training data, which is a very valid problem in today's organizations. Detailed workflow diagrams depicting text classification as a supervised Machine Learning problem helped us in relating NLP with Machine Learning so that we can use Machine Learning techniques and methodologies to solve this problem of predicting sentiment when labeled data is available.

The focus on supervised methods was two-fold. This included traditional Machine Learning approaches and models like Logistic Regression and Support Vector Machines and newer Deep Learning models including deep neural networks, RNNs, and LSTMs. Detailed concepts, workflows, hands-on examples and comparative analyses with multiple supervised models and different feature engineering techniques have been covered for the purpose of predicting sentiment from movie reviews with maximum model performance. The final section of this chapter covered a very important aspect of Machine Learning that is often neglected in our analyses. We looked at ways to analyze and interpret the cause of positive or negative sentiment. Analyzing and visualizing model interpretations and topic models have been covered with several examples, to give you a good insight into how you can re-use these frameworks on your own datasets. The frameworks and methodologies used in this chapter should be useful for you in tackling similar problems on your own text data in the future.