CHAPTER 4

Text Classification

Learning to process and understand text is one of the first steps on the journey to getting meaningful insights from textual data. Though it is important to understand how language is structured and specific text syntax patterns, that alone is not sufficient to be of much use to businesses and organizations who want to derive useful patterns and insights and get maximum use out of their vast volumes of text data. Knowledge of language processing coupled with concepts from analytics and machine learning (ML) help in building systems that can leverage text data and help solve real-world practical problems which benefit businesses.

Various aspects of ML include supervised learning, unsupervised learning, reinforcement learning, and more recently deep learning. Each of these concepts involves several techniques and algorithms that can be leveraged on text data and to build self-learning systems that do not need too much manual supervision. An ML model is a combination of data and algorithms—you got a taste of that in Chapter 3 was we built our own parsers and taggers. The benefit of ML is that once a model is trained, we can directly use it on new and previously unseen data to start seeing useful insights and desired results.

One of the most relevant and challenging problems is *text classification* or *categorization*, which involves trying to organize text documents into various categories based on inherent properties or attributes of each text document. This is used in various domains, including email spam identification and news categorization. The concept may seem simple, and if you have a small number of documents, you can look at each document and gain some idea about what it is trying to indicate. Based on this knowledge, you can group similar documents into categories or classes. It's more challenging when the number of text documents to be classified increases to several hundred thousands or millions. This is where techniques like feature extraction and supervised or unsupervised ML come in handy. Document classification is a generic problem not limited to text alone but also can be extended for other items like music, images, video, and other media.

To formalize our problem more clearly, we will have a given set of classes or categories and several text documents. Remember that documents are basically sentences or paragraphs of text. This forms a corpus. Our task would be to determine which class or classes each document belongs to. This entire process involves several steps which we will be discussing in detail later in this chapter. Briefly, for a supervised classification problem, we need to have some labelled data that we could use for training a text classification model. This data would essentially be curated documents that are already assigned to some specific class or category beforehand. Using this, we would essentially

© Dipanjan Sarkar 2016
D. Sarkar, *Text Analytics with Python*, DOI 10.1007/978-1-4842-2388-8_4

extract features and attributes from each document and make our model learn these attributes corresponding to each particular document and its class/category by feeding it to a supervised ML algorithm. Of course, the data would need to be pre-processed and normalized before building the model. Once done, we would follow the same process of normalization and feature extraction and then feed it to the model to predict the class or category for new documents. However, for an unsupervised classification problem, we would essentially not have any pre-labelled training documents. We would use techniques like clustering and document similarity measures to cluster documents together based on their inherent properties and assign labels to them.

In this chapter, we will discuss the concept of text classification and how it can be formulated as a supervised ML problem. We will also talk about the various forms of classification and what they indicate. A clear depiction for the essential steps necessary to complete a text classification workflow will also be presented, and we will be covering some of the essential steps from the same workflow, which have not been discussed before, including feature extraction, classifiers, model evaluation, and finally we will put them all together in building a text classification system on real-world data.

What Is Text Classification?

Before we define text classification, we need to understand the scope of textual data and what we really mean by *classification*. The textual data involved here can be anything ranging from a phrase, sentence, or a complete document with paragraphs of text, which can be obtained from corpora, blogs, or anywhere from the Web. Text classification is also often called *document classification* just to cover all forms of textual content under the word *document*. The word *document* could be defined as some form of concrete representation of thoughts or events that could be in the form of writing, recorded speech, drawings, or presentations. I use the term *document* here to represent textual data such as sentences or paragraphs belonging to the English language.

Text classification is also often called *text categorization*, although I explicitly use the word *classification* here for two reasons. First, it depicts the same essence as text categorization, where we want to classify documents. The second reason is to also show that we would be using classification or a supervised ML approach here to classify or categorize the text. Text categorization can be done in many ways, as mentioned. We will be focusing explicitly on a supervised approach using classification. The process of classification is not restricted to text alone. It is used quite frequently in other domains including science, healthcare, weather forecasting, and technology.

Text or document classification is the process of assigning text documents into one or more classes or categories, assuming that we have a predefined set of classes. Documents here are textual documents, and each document can contain a sentence or even a paragraph of words. A text classification system would successfully be able to classify each document to its correct class(es) based on inherent properties of the document. Mathematically, we can define it like this: given some description and attributes *d* for a document *D*, where $d \in D$, and we have a set of predefined classes or categories, $C = \{c_1, c_2, c_3, \ldots, c_n\}$. The actual document *D* can have many inherent properties and attributes that lead it to being an entity in a high-dimensional space. Using a subset of that space with a limit set of descriptions and features depicted by *d*, we

should be able to successfully assign the original document D to its correct class C_x using a text classification system T. This can be represented by $T : D \rightarrow C_x$.

We will talk more about the text classification system in detail later in the chapter. Figure 4-1 shows a high-level conceptual representation of the text classification process.

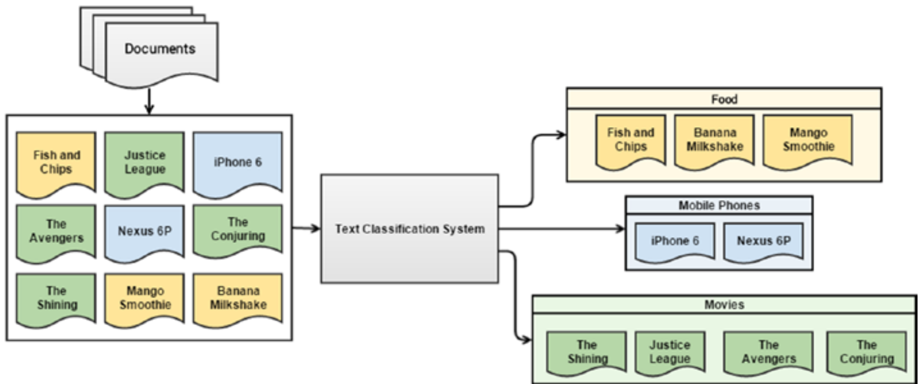

Figure 4-1. *Conceptual overview of text classification*

In Figure 4-1, we can see there are several documents representing products which can be assigned to various categories of food, mobile phones, and movies. Initially, these documents are all present together, just as a text corpus has various documents in it. Once it goes through a text classification system, represented as a black box here, we can see that each document is assigned to one specific class or category we had defined previously. Here the documents are just represented by their names, but in real data, they can contain much more, including descriptions of each product, specific attributes such as movie genre, product specifications, constituents, and many more properties that can be used as features in the text classification system to make document identification and classification easier.

There are various types of text classification. This chapter focuses on two major types, which are based on the type of content that makes up the documents:

- Content-based classification
- Request-based classification

Both types are more like different philosophies or ideals behind approaches to classifying text documents rather than specific technical algorithms or processes. *Content-based classification* is the type of text classification where priorities or weights are given to specific subjects or topics in the text content that would help determine the class of the document. A conceptual example would be that a book with more than 30 percent of its content about food preparations can be classified under cooking/recipes. *Request-based classification* is influenced by user requests and is targeted towards specific user groups and audiences. This type of classification is governed by specific policies and ideals.

Automated Text Classification

We now have an idea of the definition and scope of text classification. We have also formally defined text classification both conceptually and mathematically, where we talked about a "text classification system" being able to classify text documents to their respective categories or classes. Consider several humans doing the task of going through each document and classifying it. They would then be a part of the text classification system we are talking about. However, that would not scale very well once there were millions of text documents to be classified quickly. To make the process more efficient and faster, we can consider automating the task of text classification, which brings us to *automated text classification*.

To automate text classification, we can make use of several ML techniques and concepts. There are mainly two types of ML techniques that are relevant to solving this problem:

- Supervised machine learning
- Unsupervised machine learning

Besides these two techniques, there are also other families of learning algorithms, such as *reinforcement learning* and *semi-supervised learning*. Let us look at both supervised and unsupervised learning algorithms in more detail, from both an ML perspective how it can be leveraged in classifying text documents.

Unsupervised learning refers to specific ML techniques or algorithms that do not require any pre-labelled training data samples to build a model. We usually have a collection of data points, which could be text or numeric, depending on the problem we are trying to solve. We extract features from each of the data points using a process known as *feature extraction* and then feed the feature set for each data point into our algorithm. We are trying to extract meaningful patterns from the data, such as trying to group together similar data points using techniques like clustering or summarizing documents based on topic models. This is extremely useful in text document categorization and is also called document clustering, where we cluster documents into groups purely based on their features, similarity, and attributes, without training any model on previously labelled data. Later chapters further discuss unsupervised learning, covering topic models, document summarization, similarity analysis, and clustering.

Supervised learning refers to specific ML techniques or algorithms that are trained on pre-labelled data samples known as *training data*. Features or attributes are extracted from this data using feature extraction, and for each data point we will have its own feature set and corresponding class/label. The algorithm learns various patterns for each type of class from the training data. Once this process is complete, we have a *trained model*. This model can then be used to predict the class for future test data samples once we feed their features to the model. Thus the machine has actually learned, based on previous training data samples, how to predict the class for new unseen data samples.

There are two major types of supervised learning algorithms:

- *Classification*: The process of supervised learning is referred to as *classification* when the outcomes to be predicted are distinct categories, thus the outcome variable is a categorical variable in this case. Examples would be news categories or movie genres.

- *Regression*: Supervised learning algorithms are known as *regression* algorithms when the outcome we want to predict is a continuous numeric variable. Examples would be house prices or people's weights.

We will be specifically focusing on classification for our problem (hence the name of the chapter—we are trying to classify or categorize text documents into distinct classes or categories. We will be following a supervised learning approach in our implementations later on.

Now we are ready to define the process of automated or ML-based text classification mathematically. Say we have a training set of documents labelled with their corresponding class or category. This can be represented by *TS*, which is a set of paired documents and labels, $TS = \{(d_{1,}\ c_1), (d_{2,}\ c_2), \ldots, (d_{n,}\ c_n)\}$ where $d_1, d_2, \ldots, d_n$ is the list of text documents, and their corresponding labels are $c_1, c_2, \ldots, c_n$ such that $c_x \in C = \{c1, c2, \ldots, cn\}$ where c_x denotes the class label for document *x* and *C* denotes the set of all possible distinct classes, any of which can be the class or classes for each document. Assuming we have our training set, we can define a supervised learning algorithm *F* such that when it is trained on our training dataset *TS*, we build a classification model or classifier γ such that we can say that $F(TS) = \gamma$. Thus the supervised learning algorithm *F* takes the input set of (*document, class*) pairs *TS* and gives us the trained classifier γ, which is our model. This process is known as the *training process*.

This model can then take a new, previously unseen document *ND* and predict its class c_{ND} such that $c_{ND} \in C$. This process is known as the *prediction process* and can be represented by $\gamma : TD \rightarrow c_{ND}$. Thus we can see that there are two main processes in the supervised text classification process:

- Training
- Prediction

An important point to remember is that some manually labelled training data is necessary for supervised text classification, so even though we are talking about automated text classification, to kick start the process we need some manual efforts. Of course, the benefits of this are manifold because once we have a trained classifier, we can keep using it to predict and classify new documents with minimal efforts and manual supervision.

There are various learning methods or algorithms that we will be discussing in a future section. These learning algorithms are not specific to text data but are generic ML algorithms that can be applied toward various types of data after due pre-processing and feature extraction. I will touch upon a couple of supervised ML algorithms and use them in solving a real-world text classification problem. These algorithms are usually trained on the training data set and often an optional validation set such that the model that is trained does not overfit to the training data, which basically means it would then not be able to generalize well and predict properly for new instances of text documents. Often the model is tuned on several of its internal parameters based on the learning algorithm and by evaluating various performance metrics like accuracy on the validation set or by using cross-validation where we split the training dataset itself into training and

validation sets by random sampling. This comprises the training process, the outcome of which yields a fully trained model that is ready to predict. In the prediction stage, we usually have new data points from the test dataset. We can use them to feed into the model after normalization and feature extraction and see how well the model is performing by evaluating its prediction performance.

There are a few types of text classification based on the number of classes to predict and the nature of predictions. These types of classification are based on the dataset, the number of classes/categories pertaining to that dataset, and the number of classes that can be predicted on any data point:

- *Binary classification* is when the total number of distinct classes or categories is two in number and any prediction can contain either one of those classes.
- *Multi-class classification*, also known as *multinomial classification*, refers to a problem where the total number of classes is more than two, and each prediction gives one class or category that can belong to any of those classes. This is an extension of the binary classification problem where the total number of classes is more than two.
- *Multi-label classification* refers to problems where each prediction can yield more than one outcome/predicted class for any data point.

Text Classification Blueprint

Now that we know the basic scope of automated text classification, this section will look at a blueprint for a complete workflow of building an automated text classifier system. This will consist of a series of steps that must be followed in both the training and testing phases mentioned in the earlier section. For building a text classification system, we need to make sure we have our source of data and retrieve that data so that we can start feeding it to our system. The following main steps outline a typical workflow for a text classification system, assuming we have our dataset already downloaded and ready to be used:

1. Prepare train and test datasets
2. Text normalization
3. Feature extraction
4. Model training
5. Model prediction and evaluation
6. Model deployment

These steps are carried out in that order for building a text classifier. Figure 4-2 shows a detailed workflow for a text classification system with the main processes highlighted in training and prediction.

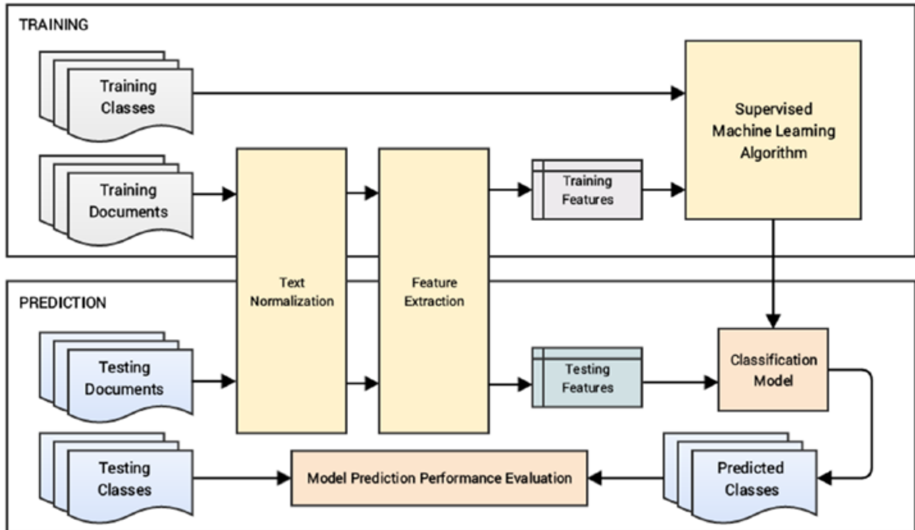

Figure 4-2. *Blueprint for building an automated text classification system*

Notice that there are two main boxes for Training and Prediction, which are the two main processes involved in building a text classifier. In general, the dataset we have is usually divided into two or three splits called the *training*, *validation* (optional), and *testing datasets*, respectively. You can see an overlap of the Text Normalization and Feature Extraction modules in Figure 4-2 for both processes, indicating that no matter which document we want to classify and predict its class, it must go through the same series of transformations in both the training and prediction process. Each document is first pre-processed and normalized, and then specific features pertaining to the document are extracted. These processes are always uniform in both the training and prediction processes to make sure that our classification model performs consistently in its predictions.

In the Training process, each document has its own corresponding class/category that was manually labeled or curated beforehand. These training text documents are processed and normalized in the Text Normalization module, giving us clean and standardized training text documents. They are then passed to the Feature Extraction module where different types of feature-extraction techniques are used to extract meaningful features from the processed text documents. We will cover some popular feature extraction techniques in a future section. These features are usually numeric arrays or vectors because standard ML algorithms work on numeric vectors. Once we have our features, we select a supervised ML algorithm and train our model.

Training the model involves feeding the feature vectors for the documents and the corresponding labels such that the algorithm is able to learn various patterns corresponding to each class/category and can reuse this learned knowledge to predict classes for future new documents. Often an optional validation dataset is used to evaluate the performance of the classification algorithm to make sure it generalizes well with the data during training. A combination of these features and the ML algorithm yields a Classification Model, which is the end stage of the Training process. Often this model is tuned using various model parameters with a process called *hyperparameter tuning* to build a better performing optimal model.

The Prediction process shown in the figure involves trying to either predict classes for new documents or evaluating how predictions are working on testing data. The test dataset documents go through the same process of normalization and feature extraction, and the test document features are passed to the trained Classification Model, which predicts the possible class for each of the documents based on previously learned patterns. If you have the true class labels for the documents that were manually labelled, you can evaluate the prediction performance of the model by comparing the true labels and the predicted labels using various metrics like accuracy. This would give an idea of how well the model performs its predictions for new documents.

Once we have a stable and working model, the last step is usually deploying the model, which normally involves saving the model and its necessary dependencies and deploying it as a service or as a running program that predicts categories for new documents as a batch job, or based on serving user requests if accessed as a web service. There are various ways to deploy ML models, and this usually depends on how you want to access it later on.

We will now discuss some of the main modules from the preceding blueprint and implement these modules so that we can integrate them all together to build a real-world text classifier.

Text Normalization

Chapter 3 covered text processing and normalization in detail—refer it to see the various methods and techniques available. In this section, we will define a normalizer module to normalize text documents and will be using it later when we build our classifier. Although various techniques are available, we will keep it fairly simple and straightforward here so that is it not too hard to follow our implementations step by step. We will implement and use the following normalization techniques in our module:

- Expanding contractions
- Text standardization through lemmatization
- Removing special characters and symbols
- Removing stopwords

We are not focusing too much on correcting spellings and other advanced techniques, but you can integrate the functions from the previous chapter implementation if you are interested. Our normalization module is implemented and available in `normalization.py`, available in the code files for this chapter. I will also be explaining each function here for your convenience. We will first start with loading the necessary dependencies. Remember that you will need our custom-defined contractions mapping file from Chapter 3, named `contractions.py`, for expanding contractions.

The following snippet shows the necessary imports and dependencies:

```
from contractions import CONTRACTION_MAP
import re
import nltk
import string
from nltk.stem import WordNetLemmatizer
```

```
stopword_list = nltk.corpus.stopwords.words('english')
wnl = WordNetLemmatizer()
```

We load all the English stopwords, the contraction mappings in `CONTRACTION_MAP`, and an instance of `WordNetLemmatizer` for carrying our lemmatization. We now define a function to tokenize text into tokens that will be used by our other normalization functions. The following function tokenizes and removes any extraneous whitespace from the tokens:

```
def tokenize_text(text):
    tokens = nltk.word_tokenize(text)
    tokens = [token.strip() for token in tokens]
    return tokens
```

Now we define a function for expanding contractions. This function is similar to our implementation from Chapter 3—it takes in a body of text and returns the same with its contractions expanded if there is a match. The following snippet helps us achieve this:

```
def expand_contractions(text, contraction_mapping):

    contractions_pattern = re.compile('({})'.format('|'.join(contraction_
    mapping.keys())),
                                      flags=re.IGNORECASE|re.DOTALL)
    def expand_match(contraction):
        match = contraction.group(0)
        first_char = match[0]
        expanded_contraction = contraction_mapping.get(match)\
                                if contraction_mapping.get(match)\
                                else contraction_mapping.get(match.lower())
        expanded_contraction = first_char+expanded_contraction[1:]
        return expanded_contraction

    expanded_text = contractions_pattern.sub(expand_match, text)
    expanded_text = re.sub("'", "", expanded_text)
    return expanded_text
```

Now that we have a function for expanding contractions, we implement a function for standardizing our text data by bringing word tokens to their base or root form using lemmatization. The following functions will help us in achieving that:

```
from pattern.en import tag
from nltk.corpus import wordnet as wn

# Annotate text tokens with POS tags
def pos_tag_text(text):
    # convert Penn treebank tag to wordnet tag
```

```
    def penn_to_wn_tags(pos_tag):
        if pos_tag.startswith('J'):
            return wn.ADJ
        elif pos_tag.startswith('V'):
            return wn.VERB
        elif pos_tag.startswith('N'):
            return wn.NOUN
        elif pos_tag.startswith('R'):
            return wn.ADV
        else:
            return None

    tagged_text = tag(text)
    tagged_lower_text = [(word.lower(), penn_to_wn_tags(pos_tag))
                         for word, pos_tag in
                         tagged_text]
    return tagged_lower_text

# lemmatize text based on POS tags
def lemmatize_text(text):

    pos_tagged_text = pos_tag_text(text)
    lemmatized_tokens = [wnl.lemmatize(word, pos_tag) if pos_tag
                         else word
                         for word, pos_tag in pos_tagged_text]
    lemmatized_text = ' '.join(lemmatized_tokens)
    return lemmatized_text
```

The preceding snippet depicts two functions implemented for lemmatization. The main function is `lemmatize_text`, which takes in a body of text data and lemmatizes each word of the text based on its POS tag if it is present and then returns the lemmatized text back to the user. For this, we need to annotate the text tokens with their POS tags. We use the `tag` function from `pattern` to annotate POS tags for each token and then further convert the POS tags from the Penn treebank syntax to WordNet syntax, since the `WordNetLemmatizer` checks for POS tag annotations based on WordNet formats. We convert each word token to lowercase, annotate it with its correct, converted WordNet POS tag, and return these annotated tokens, which are finally fed into the `lemmatize_text` function.

The following function helps us remove special symbols and characters:

```
def remove_special_characters(text):
    tokens = tokenize_text(text)
    pattern = re.compile('[{}]'.format(re.escape(string.punctuation)))
    filtered_tokens = filter(None, [pattern.sub('', token) for token in
    tokens])
    filtered_text = ' '.join(filtered_tokens)
    return filtered_text
```

We remove special characters by tokenizing the text just so we can remove some of the tokens that are actually contractions, but we may have failed to remove them in our first step, like `"s"` or `"re"`. We will do this when we remove stopwords. However, you can also remove special characters without tokenizing the text. We remove all special symbols defined in `string.punctuation` from our text using regular expression matches. The following function helps us remove stopwords from our text data:

```
def remove_stopwords(text):
    tokens = tokenize_text(text)
    filtered_tokens = [token for token in tokens if token not in
    stopword_list]
    filtered_text = ' '.join(filtered_tokens)
    return filtered_text
```

Now that we have all our functions defined, we can build our text normalization pipeline by chaining all these functions one after another. The following function implements this, where it takes in a corpus of text documents and normalizes them and returns a normalized corpus of text documents:

```
def normalize_corpus(corpus, tokenize=False):

    normalized_corpus = []
    for text in corpus:
        text = expand_contractions(text, CONTRACTION_MAP)
        text = lemmatize_text(text)
        text = remove_special_characters(text)
        text = remove_stopwords(text)
        normalized_corpus.append(text)
        if tokenize:
            text = tokenize_text(text)
            normalized_corpus.append(text)

    return normalized_corpus
```

That brings us to the end of our discussion and implementation of necessary functions for our text normalization module. We will now look at concepts and practical implementation for feature extraction.

Feature Extraction

There are various feature-extraction techniques that can be applied on text data, but before we jump into then, let us consider what we mean by features. Why do we need them, and how they are useful? In a dataset, there are typically many data points. Usually the rows of the dataset and the columns are various features or properties of the dataset, with specific values for each row or observation. In ML terminology, *features* are unique, measurable attributes or properties for each observation or data point in a dataset. Features are usually numeric in nature and can be absolute numeric values or categorical

features that can be encoded as binary features for each category in the list using a process called *one-hot encoding*. The process of extracting and selecting features is both art and science, and this process is called *feature extraction* or *feature engineering*.

Usually extracted features are fed into ML algorithms for learning patterns that can be applied on future new data points for getting insights. These algorithms usually expect features in the form of numeric vectors because each algorithm is at heart a mathematical operation of optimization and minimizing loss and error when it tries to learn patterns from data points and observations. So, with textual data there is the added challenge of figuring out how to transform textual data and extract numeric features from it.

Now we will look at some feature-extraction concepts and techniques specially aligned towards text data.

The *Vector Space Model* is a concept and model that is very useful in case we are dealing with textual data and is very popular in information retrieval and document ranking. The Vector Space Model, also known as the *Term Vector Model*, is defined as a mathematical and algebraic model for transforming and representing text documents as numeric vectors of specific terms that form the vector dimensions. Mathematically this can be defines as follows. Say we have a document D in a document vector space VS. The number of dimensions or columns for each document will be the total number of distinct terms or words for all documents in the vector space. So, the vector space can be denoted

$$VS = \{W_1, W_2, \ldots, W_n\}$$

where there are n distinct words across all documents. Now we can represent document D in this vector space as

$$D = \{w_{D1}, w_{D2}, \ldots, w_{Dn}\}$$

where w_{Dn} denotes the weight for word n in document D. This weight is a numeric value and can represent anything, ranging from the frequency of that word in the document, to the average frequency of occurrence, or even to the TF-IDF weight (discussed shortly).

We will be talking about and implementing the following feature-extraction techniques:

- Bag of Words model
- TF-IDF model
- Advanced word vectorization models

An important thing to remember for feature extraction is that once we build a feature extractor using some transformations and mathematical operations, we need to make sure we reuse the same process when extracting features from new documents to be predicted, and not rebuild the whole algorithm again based on the new documents. We will be depicting this also with an example for each technique. Do note that for implementations based on practical examples in this section, we will be making use of the `nltk`, `gensim`, and `scikit-learn` libraries, which you can install using `pip` as discussed earlier (in case you do not have them installed already).

The implementations are divided into two major modules. The file `feature_extractors.py` contains the generic functions we will be using later on when building the classifier, and we have used the same functions in the `feature_extraction_demo.py` file to show how each technique works with some practical examples. You can access them from the code files, and as always I will be presenting the same code in this chapter for ease of understanding. We will be using the following documents depicted in the `CORPUS` variable to extract features from and building some of the vectorization models. To illustrate how feature extraction will work for a new document (as a part of test dataset), we will also use a separate document as shown in the variable `new_doc` in the following snippet:

```
CORPUS = [
'the sky is blue',
'sky is blue and sky is beautiful',
'the beautiful sky is so blue',
'i love blue cheese'
]

new_doc = ['loving this blue sky today']
```

Bag of Words Model

The Bag of Words model is perhaps one of the simplest yet most powerful techniques to extract features from text documents. The essence of this model is to convert text documents into vectors such that each document is converted into a vector that represents the frequency of all the distinct words that are present in the document vector space for that specific document. Thus, considering our sample vector from the previous mathematical notation for *D*, the weight for each word is equal to its frequency of occurrence in that document.

An interesting thing is that we can even create the same model for individual word occurrences as well as occurrences for n-grams, which would make it an n-gram Bag of Words model such that frequency of distinct n-grams in each document would also be considered.

The following code snippet gives us a function that implements a Bag of Words–based feature-extraction model that also accepts an `ngram_range` parameter to take into account n-grams as features:

```
from sklearn.feature_extraction.text import CountVectorizer

def bow_extractor(corpus, ngram_range=(1,1)):

    vectorizer = CountVectorizer(min_df=1, ngram_range=ngram_range)
    features = vectorizer.fit_transform(corpus)
    return vectorizer, features
```

The preceding function uses the `CountVectorizer` class. You can access its detailed API (Application Programming Interface) documentation at `http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.`

html#sklearn.feature_extraction.text.CountVectorizer, which has a whole bunch of various parameters for more fine-tuning based on the type of features you want to extract. We use its default configuration, which is enough for most scenarios, with min_df set to 1 indicating taking terms having a minimum frequency of 1 in the overall document vector space. You can set ngram_range to various parameters like (1, 3) would build feature vectors consisting of all unigrams, bigrams, and trigrams. The following snippet shows the function in action on our sample corpora of four training documents and one test document:

```
# build bow vectorizer and get features
In [371]: bow_vectorizer, bow_features = bow_extractor(CORPUS)
     ...: features = bow_features.todense()
     ...: print features
[[0 0 1 0 1 0 1 0 1]
 [1 1 1 0 2 0 2 0 0]
 [0 1 1 0 1 0 1 1 1]
 [0 0 1 1 0 1 0 0 0]]

# extract features from new document using built vectorizer
In [373]: new_doc_features = bow_vectorizer.transform(new_doc)
     ...: new_doc_features = new_doc_features.todense()
     ...: print new_doc_features
[[0 0 1 0 0 0 1 0 0]]

# print the feature names
In [374]: feature_names = bow_vectorizer.get_feature_names()
     ...: print feature_names
[u'and', u'beautiful', u'blue', u'cheese', u'is', u'love', u'sky', u'so',
u'the']
```

That output shows how each text document has been converted to vectors. Each row represents one document from our corpus, and we do the same for both our corpora. The vectorizer is built using documents from CORPUS. We extract features from it and also use this built vectorizer to extract features from a completely new document. Each column in a vector represents the words depicted in feature_names, and the value is the frequency of that word in the document represented by the vector. It may be hard to comprehend this at first glance, so I have prepared the following function, which I hope you can use to understand the feature vectors better:

```
import pandas as pd

def display_features(features, feature_names):
    df = pd.DataFrame(data=features,
                      columns=feature_names)
    print df
```

Now you can feed the feature names and vectors to this function and see the feature matrix in a much easier-to-understand structure, shown here:

```
In [379]: display_features(features, feature_names)
   and  beautiful  blue  cheese  is  love  sky  so  the
0    0          0     1       0   1     0    1   0    1
1    1          1     1       0   2     0    2   0    0
2    0          1     1       0   1     0    1   1    1
3    0          0     1       1   0     1    0   0    0

In [380]: display_features(new_doc_features, feature_names)
   and  beautiful  blue  cheese  is  love  sky  so  the
0    0          0     1       0   0     0    1   0    0
```

That makes things much clearer, right? Consider the second document of `CORPUS`, represented in the preceding in row 1 of the first table. You can see that `'sky is blue and sky is beautiful'` has value 2 for the feature `sky`, 1 for `beautiful`, and so on. Values of 0 are assigned for words not present in the document. Note that for the new document `new_doc`, there is no feature for the words `today`, `this`, or `loving` in the sentence. The reason for this is what I mentioned before—that the feature-extraction process, model, and vocabulary are always based on the training data and will never change or get influenced on newer documents, which it will predict later as a part of testing or otherwise. You might have guessed that this is because a model is always trained on some training data and is never influenced by newer documents unless we plan on rebuilding that model. Hence, the features in this model are always limited based on the document vector space of the training corpus.

You have now started to get an idea of how to extract meaningful vector-based features from text data, which previously seemed impossible. Try out the preceding functions by setting `ngram_range` to `(1, 3)` and see the outputs.

TF-IDF Model

The Bag of Words model is good, but the vectors are completely based on absolute frequencies of word occurrences. This has some potential problems where words that may tend to occur a lot across all documents in the corpus will have higher frequencies and will tend to overshadow other words that may not occur as frequently but may be more interesting and effective as features to identify specific categories for the documents. This is where TF-IDF comes into the picture. TF-IDF stands for Term Frequency-Inverse Document Frequency, a combination of two metrics: *term frequency* and *inverse document frequency*. This technique was originally developed as a metric for ranking functions for showing search engine results based on user queries and has come to be a part of information retrieval and text feature extraction now.

Let us formally define TF-IDF now and look at the mathematical representations for it before diving into its implementation. Mathematically, TF-IDF is the product of two metrics and can be represented as $tfidf = tf \times idf$, where term frequency *(tf)* and inverse-document frequency *(idf)* represent the two metrics.

Term frequency denoted by *tf* is what we had computed in the Bag of Words model. Term frequency in any document vector is denoted by the raw frequency value of that term in a particular document. Mathematically it can be represented as $tf\left(w, D\right) = f_{w_D}$, where f_{w_D} denotes frequency for word *w* in document *D*, which becomes the term

frequency *(tf)*. There are various other representations and computations for term frequency, such as converting frequency to a binary feature where 1 means the term has occurred in the document and 0 means it has not. Sometimes you can also normalize the absolute raw frequency using logarithms or averaging the frequency. We will be using the raw frequency in our computations.

Inverse document frequency denoted by *idf* is the inverse of the document frequency for each term. It is computed by dividing the total number of documents in our corpus by the document frequency for each term and then applying logarithmic scaling on the result. In our implementation we will be adding 1 to the document frequency for each term just to indicate that we also have one more document in our corpus that essentially has every term in the vocabulary. This is to prevent potential division-by-zero errors and smoothen the inverse document frequencies. We also add 1 to the result of our *idf* computation to avoid ignoring terms completely that might have zero *idf*. Mathematically our implementation for *idf* can be represented by

$$idf(t) = 1 + \log \frac{C}{1 + df(t)}$$

where $idf(t)$ represents the *idf* for the term t, C represents the count of the total number of documents in our corpus, and $df(t)$ represents the frequency of the number of documents in which the term t is present.

Thus the term frequency-inverse document frequency can be computed by multiplying the above two measures together. The final TF-IDF metric we will be using is a normalized version of the *tfidf* matrix we get from the product of *tf* and *idf*. We will normalize the *tfidf* matrix by dividing it with the L2 norm of the matrix, also known as the *Euclidean norm*, which is the square root of the sum of the square of each term's *tfidf* weight. Mathematically we can represent the final *tfidf* feature vector as $tfidf = \frac{tfidf}{\|tfidf\|}$,

where $\|tfidf\|$ represents the Euclidean L2 norm for the *tfidf* matrix.

The following code snippet shows an implementation of getting the *tfidf*-based feature vectors, considering we have our Bag of Words feature vectors we obtained in the previous section:

```
from sklearn.feature_extraction.text import TfidfTransformer

def tfidf_transformer(bow_matrix):

    transformer = TfidfTransformer(norm='l2',
                                   smooth_idf=True,
                                   use_idf=True)
    tfidf_matrix = transformer.fit_transform(bow_matrix)
    return transformer, tfidf_matrix
```

You can see that we have used the L2 norm option in the parameters and also made sure we smoothen the *idf*s to give weightages also to terms that may have zero *idf* so that we do not ignore them. We can see this function in action in the following code snippet:

```
import numpy as np
from feature_extractors import tfidf_transformer
feature_names = bow_vectorizer.get_feature_names()

# build tfidf transformer and show train corpus tfidf features
In [388]: tfidf_trans, tdidf_features = tfidf_transformer(bow_features)
     ...: features = np.round(tdidf_features.todense(), 2)
     ...: display_features(features, feature_names)
    and  beautiful  blue  cheese    is  love   sky    so   the
0  0.00       0.00  0.40    0.00  0.49  0.00  0.49  0.00  0.60
1  0.44       0.35  0.23    0.00  0.56  0.00  0.56  0.00  0.00
2  0.00       0.43  0.29    0.00  0.35  0.00  0.35  0.55  0.43
3  0.00       0.00  0.35    0.66  0.00  0.66  0.00  0.00  0.00

# show tfidf features for new_doc using built tfidf transformer
In [389]: nd_tfidf = tfidf_trans.transform(new_doc_features)
     ...: nd_features = np.round(nd_tfidf.todense(), 2)
     ...: display_features(nd_features, feature_names)
   and  beautiful  blue  cheese   is  love   sky   so  the
0  0.0        0.0  0.63     0.0  0.0   0.0  0.77  0.0  0.0
```

Thus the preceding outputs show the *tfidf* feature vectors for all our sample documents. We use the `TfidfTransformer` class, which helps us in computing the *tfidf*s for each document based on the equations described earlier.

Now we will show how the internals of this class work. You will also see how to implement the mathematical equations described earlier to compute the *tfidf*-based feature vectors. This section is dedicated to ML experts (and curious readers who are interested in how things work behind the scenes). We will start with loading necessary dependencies and computing the term frequencies (*TF*) by reusing our Bag of Words-based features for our sample corpus, which can also act as the term frequencies for our training `CORPUS`:

```
import scipy.sparse as sp
from numpy.linalg import norm
feature_names = bow_vectorizer.get_feature_names()

# compute term frequency
tf = bow_features.todense()
tf = np.array(tf, dtype='float64')

# show term frequencies
In [391]: display_features(tf, feature_names)
   and  beautiful  blue  cheese   is  love  sky   so  the
0  0.0        0.0   1.0     0.0  1.0   0.0  1.0  0.0  1.0
1  1.0        1.0   1.0     0.0  2.0   0.0  2.0  0.0  0.0
2  0.0        1.0   1.0     0.0  1.0   0.0  1.0  1.0  1.0
3  0.0        0.0   1.0     1.0  0.0   1.0  0.0  0.0  0.0
```

We will now compute our document frequencies (*DF*) for each term based on the number of documents in which it occurs. The following snippet shows how to obtain it from our Bag of Words feature matrix:

```
# build the document frequency matrix
df = np.diff(sp.csc_matrix(bow_features, copy=True).indptr)
df = 1 + df # to smoothen idf later

# show document frequencies
In [403]: display_features([df], feature_names)
   and  beautiful  blue  cheese  is  love  sky  so  the
0    2          3     5       2   4     2    4   2    3
```

This tells us the document frequency (*DF*) for each term and you can verify it with the documents in CORPUS. Remember that we have added 1 to each frequency value to smoothen the *idf* values later and prevent division-by-zero errors by assuming we have a document (imaginary) that has all the terms once. Thus, if you check in the CORPUS, you will see that blue occurs 4(+1) times, sky occurs 3(+1) times, and so on, considering (+1) for our smoothening.

Now that we have the document frequencies, we will compute the inverse document frequency (*idf*) using our formula defined earlier. Remember to add 1 to the total count of documents in the corpus to add the document that we had assumed earlier to contain all the terms at least once for smoothening the *idf*s:

```
# compute inverse document frequencies
total_docs = 1 + len(CORPUS)
idf = 1.0 + np.log(float(total_docs) / df)

# show inverse document frequencies
In [406]: display_features([np.round(idf, 2)], feature_names)
    and  beautiful  blue  cheese    is  love   sky    so   the
0  1.92       1.51   1.0    1.92  1.22  1.92  1.22  1.92  1.51

# compute idf diagonal matrix
total_features = bow_features.shape[1]
idf_diag = sp.spdiags(idf, diags=0, m=total_features, n=total_features)
idf = idf_diag.todense()

# print the idf diagonal matrix
In [407]: print np.round(idf, 2)
[[ 1.92  0.    0.    0.    0.    0.    0.    0.    0.  ]
 [ 0.    1.51  0.    0.    0.    0.    0.    0.    0.  ]
 [ 0.    0.    1.    0.    0.    0.    0.    0.    0.  ]
 [ 0.    0.    0.    1.92  0.    0.    0.    0.    0.  ]
 [ 0.    0.    0.    0.    1.22  0.    0.    0.    0.  ]
 [ 0.    0.    0.    0.    0.    1.92  0.    0.    0.  ]
 [ 0.    0.    0.    0.    0.    0.    1.22  0.    0.  ]
 [ 0.    0.    0.    0.    0.    0.    0.    1.92  0.  ]
 [ 0.    0.    0.    0.    0.    0.    0.    0.    1.51]]
```

You can now see the *idf* matrix that we created based on our mathematical equation, and we also convert it to a diagonal matrix, which will be helpful later on when we want to compute the product with term frequency.

Now that we have our *tf*s and *idf*s, we can compute the *tfidf* feature matrix using matrix multiplication, as shown in the following snippet:

```
# compute tfidf feature matrix
tfidf = tf * idf

# show tfidf feature matrix
In [410]: display_features(np.round(tfidf, 2), feature_names)
    and  beautiful  blue  cheese    is  love   sky    so   the
0  0.00       0.00   1.0    0.00  1.22  0.00  1.22  0.00  1.51
1  1.92       1.51   1.0    0.00  2.45  0.00  2.45  0.00  0.00
2  0.00       1.51   1.0    0.00  1.22  0.00  1.22  1.92  1.51
3  0.00       0.00   1.0    1.92  0.00  1.92  0.00  0.00  0.00
```

We now have our *tfidf* feature matrix, but wait! It is not yet over. We have to divide it with the L2 norm, if you remember from our equations depicted earlier. The following snippet computes the *tfidf* norms for each document and then divides the *tfidf* weights with the norm to give us the final desired *tfidf* matrix:

```
# compute L2 norms
norms = norm(tfidf, axis=1)

# print norms for each document
In [412]: print np.round(norms, 2)
[ 2.5   4.35  3.5   2.89]

# compute normalized tfidf
norm_tfidf = tfidf / norms[:, None]

# show final tfidf feature matrix
In [415]: display_features(np.round(norm_tfidf, 2), feature_names)
    and  beautiful  blue  cheese    is  love   sky    so   the
0  0.00       0.00  0.40    0.00  0.49  0.00  0.49  0.00  0.60
1  0.44       0.35  0.23    0.00  0.56  0.00  0.56  0.00  0.00
2  0.00       0.43  0.29    0.00  0.35  0.00  0.35  0.55  0.43
3  0.00       0.00  0.35    0.66  0.00  0.66  0.00  0.00  0.00
```

Compare the preceding obtained *tfidf* feature matrix for the documents in `CORPUS` to the feature matrix obtained using `TfidfTransformer` earlier. Note they are exactly the same, thus verifying that our mathematical implementation was correct—and in fact this very same implementation is adopted by `scikit-learn`'s `TfidfTransformer` behind the scenes using some more optimizations. Now, suppose we want to compute the *tfidf*-based feature matrix for our new document `new_doc`. We can do it using the following snippet. We reuse the `new_doc_features` Bag of Words vector from before for the term frequencies:

```
# compute new doc term freqs from bow freqs
nd_tf = new_doc_features
nd_tf = np.array(nd_tf, dtype='float64')

# compute tfidf using idf matrix from train corpus
nd_tfidf = nd_tf*idf
nd_norms = norm(nd_tfidf, axis=1)
norm_nd_tfidf = nd_tfidf / nd_norms[:, None]

# show new_doc tfidf feature vector
In [418]: display_features(np.round(norm_nd_tfidf, 2), feature_names)
   and  beautiful  blue  cheese   is  love   sky   so  the
0  0.0        0.0  0.63     0.0  0.0   0.0  0.77  0.0  0.0
```

The preceding output depicts the *tfidf*-based feature vector for `new_doc`, and you can see it is the same as the one obtained by `TfidfTransformer`.

Now that we know how the internals work, we are going to implement a generic function that can directly compute the *tfidf*-based feature vectors for documents from the raw documents themselves. The following snippet depicts the same:

```
from sklearn.feature_extraction.text import TfidfVectorizer

def tfidf_extractor(corpus, ngram_range=(1,1)):

    vectorizer = TfidfVectorizer(min_df=1,
                                 norm='l2',
                                 smooth_idf=True,
                                 use_idf=True,
                                 ngram_range=ngram_range)
    features = vectorizer.fit_transform(corpus)
    return vectorizer, features
```

The preceding function makes use of the `TfidfVectorizer`, which directly computes the *tfidf* vectors by taking the raw documents themselves as input and internally computing the term frequencies as well as the inverse document frequencies, eliminating the need to use the `CountVectorizer` for computing the term frequencies based on the Bag of Words model. Support is also present for adding n-grams to the feature vectors. We can see the function in action in the following snippet:

```
# build tfidf vectorizer and get training corpus feature vectors
In [425]: tfidf_vectorizer, tdidf_features = tfidf_extractor(CORPUS)
     ...: display_features(np.round(tdidf_features.todense(), 2), feature_
          names)
    and  beautiful  blue  cheese    is  love   sky    so   the
0  0.00       0.00  0.40    0.00  0.49  0.00  0.49  0.00  0.60
1  0.44       0.35  0.23    0.00  0.56  0.00  0.56  0.00  0.00
2  0.00       0.43  0.29    0.00  0.35  0.00  0.35  0.55  0.43
3  0.00       0.00  0.35    0.66  0.00  0.66  0.00  0.00  0.00
```

```
# get tfidf feature vector for the new document
In [426]: nd_tfidf = tfidf_vectorizer.transform(new_doc)
     ...: display_features(np.round(nd_tfidf.todense(), 2), feature_names)
   and  beautiful  blue  cheese   is  love   sky   so  the
0  0.0        0.0  0.63     0.0  0.0   0.0  0.77  0.0  0.0
```

You can see from the preceding outputs that the *tfidf* feature vectors match to the ones we obtained previously. This brings us to the end of our discussion on feature extraction using *tfidf*. Now we will look at some advanced word vectorization techniques.

Advanced Word Vectorization Models

There are various approaches to creating more advanced word vectorization models for extracting features from text data. Here we will discuss a couple of them that use Google's popular word2vec algorithm. The word2vec model, released in 2013 by Google, is a neural network–based implementation that learns distributed vector representations of words based on continuous Bag of Words and skip-gram–based architectures. The word2vec framework is much faster than other neural network–based implementations and does not require manual labels to create meaningful representations among words. You can find more details on Google's word2vec project at https://code.google.com/archive/p/word2vec/. You can even try out some of the implementations yourself if you are interested.

We will be using the gensim library in our implementation, which is Python implementation for word2vec that provides several high-level interfaces for easily building these models. The basic idea is to provide a corpus of documents as input and get feature vectors for them as output. Internally, it constructs a vocabulary based on the input text documents and learns vector representations for words based on various techniques mentioned earlier, and once this is complete, it builds a model that can be used to extract word vectors for each word in a document. Using various techniques like average weighting or *tfidf* weighting, we can compute the averaged vector representation of a document using its word vectors. You can get more details about the interface for gensim's word2vec implementation at http://radimrehurek.com/gensim/models/word2vec.html.

We will be mainly focusing on the following parameters when we build our model from our sample training corpus:

- size: This parameter is used to set the size or dimension for the word vectors and can range from tens to thousands. You can try out various dimensions to see which gives the best result.
- window: This parameter is used to set the context or window size. which specifies the length of the window of words that should be considered for the algorithm to take into account as context when training.
- min_count: This parameter specifies the minimum word count needed across the corpus for the word to be considered in the vocabulary. This helps in removing very specific words that may not have much significance because they occur very rarely in the documents.

- `sample`: This parameter is used to downsample effects of occurrence of frequent words. Values between 0.01 and 0.0001 are usually ideal.

Once we build a model, we will define and implement two techniques of combining word vectors together in text documents based on certain weighing schemes. We will implement two techniques mentioned as follows.

- Averaged word vectors
- TF-IDF weighted word vectors

Let us start the feature-extraction process by building our `word2vec` model on our sample training corpus before going into further implementations. The following code snippet shows how:

```
import gensim
import nltk

# tokenize corpora
TOKENIZED_CORPUS = [nltk.word_tokenize(sentence)
                    for sentence in CORPUS]
tokenized_new_doc = [nltk.word_tokenize(sentence)
                     for sentence in new_doc]

# build the word2vec model on our training corpus
model = gensim.models.Word2Vec(TOKENIZED_CORPUS, size=10, window=10,
                               min_count=2, sample=1e-3)
```

As you can see, we have built the model using the parameters described earlier; you can play around with these and also look at other parameters from the documentation to change the architecture type, number of workers, and so on. Now that we have our model ready, we can start implementing our feature extraction techniques.

Averaged Word Vectors

The preceding model creates a vector representation for each word in the vocabulary. We can access them by just typing in the following code:

```
In [430]: print model['sky']
[ 0.01608407 -0.04819566  0.04227461 -0.03011346  0.0254148   0.01728328
  0.0155535   0.00774884 -0.02752112  0.01646519]

In [431]: print model['blue']
[-0.0472235   0.01662185 -0.01221706 -0.04724348 -0.04384995  0.00193343
 -0.03163504 -0.03423524  0.02661656  0.03033725]
```

Each word vector is of length 10 based on the size parameter specified earlier. But when we deal with sentences and text documents, they are of unequal length, and we must carry out some form of combining and aggregation operations to make sure the

number of dimensions of the final feature vectors are the same, regardless of the length of the text document, number of words, and so on. In this technique, we will use an average weighted word vectorization scheme, where for each text document we will extract all the tokens of the text document, and for each token in the document we will capture the subsequent word vector if present in the vocabulary. We will sum up all the word vectors and divide the result by the total number of words matched in the vocabulary to get a final resulting averaged word vector representation for the text document. This can be mathematically represented using the equation

$$AWV(D) = \frac{\sum_{1}^{n} wv(w)}{n}$$

where $AVW(D)$ is the averaged word vector representation for document D, containing words $w_1, w_2, \ldots, w_n$, and $wv(w)$ is the word vector representation for the word w.

The following snippet shows the pseudocode for the algorithm just described:

```
model := the word2vec model we built
vocabulary := unique_words(model)
document := [words]
matched_word_count := 0
vector := []

for word in words:
        if word in vocabulary:
                vector := vector + model[word]
                matched_word_count :=  matched_word_count + 1

averaged_word_vector := vector / matched_word_count
```

That snippet shows the flow of operations in a better way that is easier to understand. We will now implement our algorithm in Python using the following code snippet:

```
import numpy as np

# define function to average word vectors for a text document
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

# generalize above function for a corpus of documents
def averaged_word_vectorizer(corpus, model, num_features):
    vocabulary = set(model.index2word)
    features = [average_word_vectors(tokenized_sentence, model, vocabulary,
    num_features)
                    for tokenized_sentence in corpus]
    return np.array(features)
```

The average_word_vectors() function must seem familiar to you—it is the concrete implementation of our algorithm shown using our pseudocode earlier. We also create a generic function averaged_word_vectorizer() to perform averaging of word vectors for a corpus of documents. The following snippet shows our function in action on our sample corpora:

```
# get averaged word vectors for our training CORPUS
In [445]: avg_word_vec_features = averaged_word_vectorizer(corpus=TOKENIZED_
CORPUS,
     ...:                                                  model=model,
     ...:                                                  num_features=10)
     ...: print np.round(avg_word_vec_features, 3)
[[ 0.006 -0.01   0.015 -0.014  0.004 -0.006 -0.024 -0.007 -0.001  0.   ]
 [-0.008 -0.01   0.021 -0.019 -0.002 -0.002 -0.011  0.002  0.003 -0.001]
 [-0.003 -0.007  0.008 -0.02  -0.001 -0.004 -0.014 -0.015  0.002 -0.01 ]
 [-0.047  0.017 -0.012 -0.047 -0.044  0.002 -0.032 -0.034  0.027  0.03 ]]

# get averaged word vectors for our test new_doc
In [447]: nd_avg_word_vec_features = averaged_word_
vectorizer(corpus=tokenized_new_doc,
     ...:                                                     model=model,
     ...:                                                     num_
                                                              features=10)
     ...: print np.round(nd_avg_word_vec_features, 3)
[[-0.016 -0.016  0.015 -0.039 -0.009  0.01  -0.008 -0.013  0.     0.023]]
```

From the preceding outputs, you can see that we have uniformly sized averaged word vectors for each document in the corpus, and these feature vectors can be used later for classification by feeding it to the ML algorithms.

TF-IDF Weighted Averaged Word Vectors

Our previous vectorizer simply sums up all the word vectors pertaining to any document based on the words in the model vocabulary and calculates a simple average by dividing with the count of matched words. This section introduces a new and novel technique

of weighing each matched word vector with the word TF-TDF score and summing up all the word vectors for a document and dividing it by the sum of all the TF-IDF weights of the matched words in the document. This would basically give us a TF-IDF weighted averaged word vector for each document.

This can be mathematically represented using the equation

$$TWA(D) = \frac{\sum_{1}^{n} wv(w) \times tfidf(w)}{n}$$

where $TWA(D)$ is the TF-IDF weighted averaged word vector representation for document D, containing words $w_1, w_2, \ldots, w_n$, where $wv(w)$ is the word vector representation and $tfidf(w)$ is the TF-IDF weight for the word w. The following snippet shows the pseudocode for this algorithm:

```
model := the word2vec model we built
vocabulary := unique_words(model)
document := [words]
tfidfs := [tfidf(word) for each word in words]
matched_word_wts := 0
vector := []

for word in words:
        if word in vocabulary:
                word_vector := model[word]
                weighted_word_vector := tfidfs[word] x word_vector
                vector := vector + weighted_word_vector
                matched_word_wts :=  matched_word_wts + tfidfs[word]

tfidf_wtd_avgd_word_vector := vector / matched_word_wts
```

That pseudocode gives structure to our algorithm and shows how to implement the algorithm from the mathematical formula we defined earlier.

The following code snippet implements this algorithm in Python so we can use it for feature extraction:

```
# define function to compute tfidf weighted averaged word vector for a document
def tfidf_wtd_avg_word_vectors(words, tfidf_vector, tfidf_vocabulary, model,
num_features):

    word_tfidfs = [tfidf_vector[0, tfidf_vocabulary.get(word)]
                   if tfidf_vocabulary.get(word)
                   else 0 for word in words]
    word_tfidf_map = {word:tfidf_val for word, tfidf_val in zip(words, word_
    tfidfs)}

    feature_vector = np.zeros((num_features,),dtype="float64")
```

```
    vocabulary = set(model.index2word)
    wts = 0.
    for word in words:
        if word in vocabulary:
            word_vector = model[word]
            weighted_word_vector = word_tfidf_map[word] * word_vector
            wts = wts + word_tfidf_map[word]
            feature_vector = np.add(feature_vector, weighted_word_vector)
    if wts:
        feature_vector = np.divide(feature_vector, wts)

    return feature_vector

# generalize above function for a corpus of documents
def tfidf_weighted_averaged_word_vectorizer(corpus, tfidf_vectors,
                                   tfidf_vocabulary, model, num_features):

    docs_tfidfs = [(doc, doc_tfidf)
                   for doc, doc_tfidf
                   in zip(corpus, tfidf_vectors)]
    features = [tfidf_wtd_avg_word_vectors(tokenized_sentence, tfidf, tfidf_
    vocabulary,
                                   model, num_features)
                    for tokenized_sentence, tfidf in docs_tfidfs]
    return np.array(features)
```

The `tfidf_wtd_avg_word_vectors()` function helps us in getting the TF-IDF weighted averaged word vector representation for a document. We also create a corresponding generic function `tfidf_weighted_averaged_word_vectorizer()` to perform TF-IDF weighted averaging of word vectors for a corpus of documents. We can see our implemented function in action on our sample corpora using the following snippet:

```
# get tfidf weights and vocabulary from earlier results and compute result
In [453]: corpus_tfidf = tdidf_features
     ...: vocab = tfidf_vectorizer.vocabulary_
     ...: wt_tfidf_word_vec_features = tfidf_weighted_averaged_word_
vectorizer(corpus=TOKENIZED_CORPUS, tfidf_vectors=corpus_tfidf,
     ...:                               tfidf_vocabulary=vocab, model=model,
                                        num_features=10)
```

```
     ...: print np.round(wt_tfidf_word_vec_features, 3)
[[ 0.011 -0.011  0.014 -0.011  0.007 -0.007 -0.024 -0.008 -0.004 -0.004]
 [ 0.    -0.014  0.028 -0.014  0.004 -0.003 -0.012  0.011 -0.001 -0.002]
 [-0.001 -0.008  0.007 -0.019  0.001 -0.004 -0.012 -0.018  0.001 -0.014]
 [-0.047  0.017 -0.012 -0.047 -0.044  0.002 -0.032 -0.034  0.027  0.03 ]]

# compute avgd word vector for test new_doc
In [454]: nd_wt_tfidf_word_vec_features = tfidf_weighted_averaged_word_
vectorizer(corpus=tokenized_new_doc, tfidf_vectors=nd_tfidf, tfidf_
vocabulary=vocab, model=model, num_features=10)
     ...: print np.round(nd_wt_tfidf_word_vec_features, 3)
[[-0.012 -0.019  0.018 -0.038 -0.006  0.01  -0.006 -0.011 -0.003  0.023]]
```

From the preceding results, you can see how we can converted each document into TF-IDF weighted averaged numeric vectors. We also used our TF-IDF weights and vocabulary, obtained earlier when we implemented TF-IDF–based feature vector extraction from documents.

Now you have a good grasp on how to extract features from text data that can be used for training a classifier.

Classification Algorithms

Classification algorithms are supervised ML algorithms that are used to classify, categorize, or label data points based on what it has observed in the past. Each classification algorithm, being a supervised learning algorithm, requires training data. This training data consists of a set of training observations where each observation is a pair consisting of an input data point, usually a feature vector like we observed earlier, and a corresponding output outcome for that input observation. There are mainly three processes classification algorithms go through:

- *Training* is the process where the supervised learning algorithm analyzes and tries to infer patterns out of training data such that it can identify which patterns lead to a specific outcome. These outcomes are often known as the class labels/class variables/response variables. We usually carry out the process of feature extraction or feature engineering to derive meaningful features from the raw data before training. These feature sets are fed to an algorithm of our choice, which then tries to identify and learn patterns from them and their corresponding outcomes. The result is an inferred function known as a model or a classification model. This model is expected to be generalized enough from learning patterns in the training set such that it can predict the classes or outcomes for new data points in the future.

- *Evaluation* involves trying to test the prediction performance of our model to see how well it has trained and learned on the training dataset. For this we usually use a validation dataset and test the performance of our model by predicting on that dataset and testing our predictions against the actual class labels, also called as the *ground truth*. Often we also use cross-validation, where the data is divided into *folds* and a chunk of it is used for training, with the remainder used to validate the trained model. Note that we also tune the model based on the validation results to get to an optimal configuration that yields maximum accuracy and minimum error. We also evaluate our model against a holdout or test dataset, but we never tune our model against that dataset because that would lead to it being biased or overfit against very specific features from the dataset. The holdout or test dataset is something of a representative sample of what new, real data samples might look like for which the model will generate predictions and how it might perform on these new data samples. Later we will look at various metrics that are typically used to evaluate and measure model performance.
- *Tuning*, also known as *hyperparameter tuning* or *optimization*, is where we focus on trying to optimize a model to maximize its prediction power and reduce errors. Each model is at heart a mathematical function with several parameters that determine model complexity, learning capability, and so on. These are known as hyperparameters because they cannot be learned directly from data and must be set prior to running and training the model. Hence, the process of choosing an optimal set of model hyperparameters such that the performance of the model yields good prediction accuracy is known as *model tuning*, and we can carry it out in various ways, including randomized search and grid search. We will not be covering this in our implementations since this is more inclined towards core machine learning and is out of our current scope as the models we will be building work well with default hyperparameter configurations. But there are plenty of resources on the Web if you are interested in model tuning and optimization.

There are various types of classification algorithms, but we will not be venturing into each one in detail. Our focus remains text classification, and I do not want to bore everyone with excessive mathematical derivations for each algorithm. However, I will touch upon a couple of algorithms that are quite effective for text classification and try to explain them, keeping the mathematical formulae to the base essentials. These algorithms are the following:

- Multinomial Naïve Bayes
- Support vector machines

There are also several other algorithms besides these you can look up, including logistic regression, decision trees, and neural networks. And ensemble techniques use a collection or ensemble of models to learn and predict outcomes that include random forests and gradient boosting, but they often don't perform very well for text classification because they are very prone to overfitting. I recommend you be careful if you plan on experimenting with them. Besides these, deep learning–based techniques have also recently become popular. They use multiple hidden layers and combine several neural network models to build a complex classification model.

We will now briefly look at some of the concepts surrounding multinomial naïve Bayes and support vector machines before using them for our classification problem.

Multinomial Naïve Bayes

This algorithm is a special case of the popular naïve Bayes algorithm, which is used specifically for prediction and classification tasks where we have more than two classes. Before looking at multinomial naïve Bayes, let us look at the definition and formulation of the naïve Bayes algorithm. The naïve Bayes algorithm is a supervised learning algorithm that puts into action the very popular Bayes' theorem. However, there is a "naïve" assumption here that each feature is independent of the others. Mathematically we can formulate this as follows: Given a response class variable y and a set of n features in the form of a feature vector $\{x_1, x_2, \ldots, x_n\}$, using Bayes' theorem we can denote the probability of the occurrence of y given the features as

$$P(y \mid x_1, x_2, \ldots, x_n) = \frac{P(y) \times P(x_1, x_2, \ldots, x_n \mid y)}{P(x_1, x_2, \ldots, x_n)}$$

under the assumption that $P(x_i \mid y, x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = P(x_i \mid y)$, and for all i we can represent this as

$$P(y \mid x_1, x_2, \ldots, x_n) = \frac{P(y) \times \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, x_2, \ldots, x_n)}$$

where i ranges from 1 to n. In simple terms, this can be written as $posterior = \frac{prior \times likelihood}{evidence}$ and now, since $P(x_1, x_2, \ldots, x_n)$ is constant, the model can be expressed like this:

$$P(y \mid x_1, x_2, \ldots, x_n) \propto P(y) \times \prod_{i=1}^{n} P(x_i \mid y)$$

This means that under the previous assumptions of independence among the features where each feature is conditionally independent of every other feature, the conditional distribution over the class variable which is to be predicted, y can be represented using the following mathematical equation as

$$P(y \mid x_1, x_2, \ldots, x_n) = \frac{1}{Z} P(y) \times \prod_{i=1}^{n} P(x_i \mid y)$$

where the evidence measure, $Z = p(x)$ is a constant scaling factor dependent on the feature variables. From this equation, we can build the naïve Bayes classifier by combining it with a rule known as the *MAP decision rule*, which stands for *maximum a posteriori*. Going into the statistical details would be impossible in the current scope, but by using it, the classifier can be represented as a mathematical function that can assign a predicted class label $\hat{y} = C_k$ for some *k* using the following representation:

$$\hat{y} = \underset{k \in \{1, 2, \ldots, K\}}{argmax}\, P(C_k) \times \prod_{i=1}^{n} P(x_i \mid C_k)$$

This classifier is often said to be simple, quite evident from its name and also because of several assumptions we make about our data and features that might not be so in the real world. Nevertheless, this algorithm still works remarkably well in many use cases related to classification, including multi-class document classification, spam filtering, and so on. They can train really fast compared to other classifiers and also work well even when we do not have sufficient training data. Models often do not perform well when they have a lot of features, and this phenomenon is known as the *curse of dimensionality*. Naïve Bayes takes care of this problem by decoupling the class variable–related conditional feature distributions, thus leading to each distribution being independently estimated as a single dimension distribution.

Multinomial naïve Bayes is an extension of the preceding algorithm for predicting and classifying data points, where the number of distinct classes or outcomes is more than two. In this case the feature vectors are usually assumed to be word counts from the Bag of Words model, but TF-IDF–based weights will also work. One limitation is that negative weight-based features can't be fed into this algorithm. This distribution can be represented as $p_y = \{p_{y1}, p_{y2}, \ldots, p_{yn}\}$ for each class label *y*, and the total number of features is *n*, which could be represented as the total vocabulary of distinct words or terms in text analytics. From the preceding equation, $p_{yi} = P(x_i \mid y)$ represents the probability of feature *i* in any observation sample that has an outcome or class*y*. The parameter p_y can be estimated with a smoothened version of maximum likelihood estimation (with relative frequency of occurrences), and represented as

$$\hat{p}_{yi} = \frac{F_{yi} + \alpha}{F_y + \alpha n}$$

where $F_{yi} = \sum_{x \in TD} x_i$ is the frequency of occurrence for the feature *i* in a sample for class label *y* in our training dataset *TD*, and $F_y = \sum_{i=1}^{|TD|} F_{yi}$ is the total frequency of all features for the class label *y*. There is some amount of smoothening one with the help of priors $\alpha \geq 0$,

which accounts for the features that are not present in the learning data points and helps in getting rid of zero-probability–related issues. Some specific settings for this parameter are used quite often. The value of $\alpha = 1$ is known as Laplace smoothing, and $\alpha < 1$ is known as Lidstone smoothing. The `scikit-learn` library provides an excellent implementation for multinomial naïve Bayes in the class `MultinomialNB`, which we will be leveraging when we build our text classifier later on.

Support Vector Machines

In machine learning, *support vector machines* (SVM) are supervised learning algorithms used for classification, regression, novelty, and anomaly or outlier detection. Considering a binary classification problem, if we have training data such that each data point or observation belongs to a specific class, the SVM algorithm can be trained based on this data such that it can assign future data points into one of the two classes. This algorithm represents the training data samples as points in space such that points belonging to either class can be separated by a wide gap between them, called a *hyperplane*, and the new data points to be predicted are assigned classes based on which side of this hyperplane they fall into. This process is for a typical linear classification process. However, SVM can also perform non-linear classification by an interesting approach known as a *kernel trick*, where kernel functions are used to operate on high-dimensional feature spaces that are non-linear separable. Usually, inner products between data points in the feature space help achieve this.

The SVM algorithm takes in a set of training data points and constructs a hyperplane of a collection of hyperplanes for a high dimensional feature space. The larger the margins of the hyperplane, the better the separation, so this leads to lower generalization errors of the classifier. Let us represent this formally and mathematically. Consider a training dataset of *n* data points $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)$ such that the class variable $y_i \in \{-1, 1\}$ where each value indicates the class corresponding to the point $\vec{x}_i$. Each data point $\vec{x}_i$ is a feature vector. The objective of the SVM algorithm is to find the max-margin hyperplane that separates the set of data points having class label of $y_i = 1$ from the set of data points having class label $y_i = -1$ such that the distance between the hyperplane and sample data points from either class nearest to it is maximized. These sample data points are known as the support vectors. Figure 4-3, courtesy of Wikipedia, shows what the vector space with the hyperplane looks like.

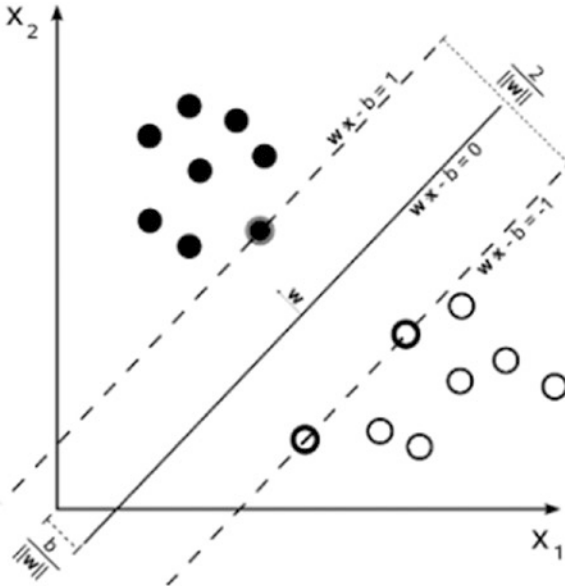

Figure 4-3. *Two-class SVM depicting hyperplane and support vectors (courtesy: Wikipedia)*

You can clearly see the hyperplane and the support vectors in the figure. The hyperplane can be defined as the set of points $\vec{x}$ which satisfy $\overrightarrow{w} \cdot \overrightarrow{x} + b = 0$ where $\vec{w}$ is the normal vector to the hyperplane, as shown in Figure 4-3, and $\frac{b}{\|\vec{w}\|}$ gives us the offset of the hyperplane from the origin toward the support vectors highlighted in the figure. There are two main types of margins that help in separating out the data points belonging to the different classes.

When the data is linearly separable, as in Figure 4-3, we can have hard margins that are basically represented by the two parallel hyperplanes depicted by the dotted lines, which help in separating the data points belonging to the two different classes. This is done taking into account that the distance between them is as large as possible. The region bounded by these two hyperplanes forms the margin with the max-margin hyperplane being in the middle. These hyperplanes are shown in the figure having the equations $\overrightarrow{w} \cdot \overrightarrow{x} + b = 1$ and $\overrightarrow{w} \cdot \overrightarrow{x} + b = -1$.

Often the data points are not linearly separable, for which we can use the *hinge loss* function, which can be represented as $\max(0, 1 - y_i(\overrightarrow{w} \cdot \overrightarrow{x_i} + b)$ and in fact the `scikit-learn` implementation of SVM can be found in `SVC`, `LinearSVC`, or `SGDClassifier` where we will use the `'hinge'` loss function (set by default) defined previously to optimize and build the model. This loss function helps us in getting the soft margins and is often known as a *soft-margin SVM*.

For a multi-class classification problem, if we have *n* classes, for each class a binary classifier is trained and learned that helps in separating between each class and the other *n-1* classes. During prediction, the *scores* (distances to hyperplanes) for each classifier are computed, and the maximum score is chosen for selecting the class label. Also often stochastic gradient descent is used for minimizing the loss function in SVM algorithms. Figure 4-4 shows how three classifiers are trained in total for a three-class SVM problem over the very popular iris dataset. This figure is built using a scikit-learn model and is obtained from the official documentation available at http://scikit-learn.org.

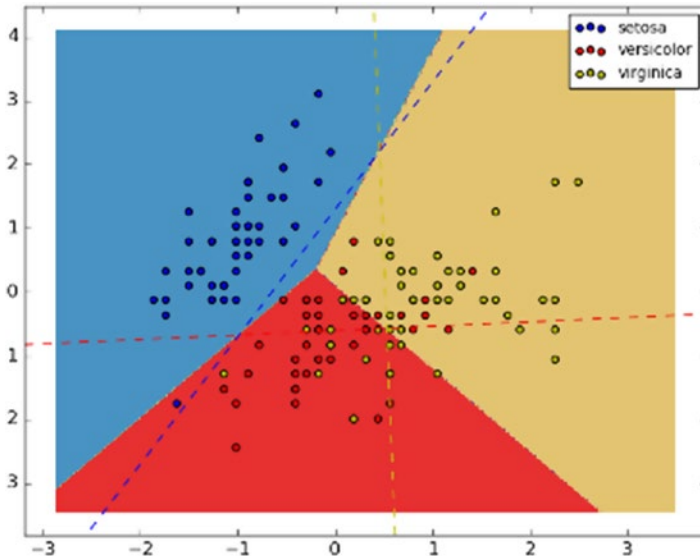

Figure 4-4. *Multi-class SVM on three classes (courtesy: scikit-learn.org)*

In Figure 4-4 you can clearly see that a total of three SVM classifiers have been trained for each of the three classes and are then combined for the final predictions so that data points belonging to each class can be labeled correctly. There are a lot of resources and books dedicated entirely towards supervised ML and classification. Interested readers should check them out to gain more in-depth knowledge on how these techniques work and how they can be applied to various problems in analytics.

Evaluating Classification Models

Training, tuning, and building models are an important part of the whole analytics lifecycle, but even more important is knowing how well these models are performing. Performance of classification models is usually based on how well they predict outcomes for new data points. Usually this performance is measured against a test or holdout dataset that consists of data points which was not used to influence or train the classifier in any way. This test dataset usually has several observations and corresponding labels.

We extract features in the same way as it was followed when training the model. These features are fed to the already trained model, and we obtain predictions for each data point. These predictions are then matched with the actual labels to see how well or how accurately the model has predicted.

Several metrics determine a model's prediction performance, but we will mainly focus on the following metrics:

- Accuracy
- Precision
- Recall
- F1 score

Let us look at a practical example to see how these metrics can be computed. Consider a binary classification problem of classifying emails as either `'spam'` or `'ham'`. Assuming we have a total of 20 emails, for which we already have the actual manual labels, we pass it through our built classifier to get predicted labels for each email. This gives us 20 predicted labels. Now we want to measure the classifier performance by comparing each prediction with its actual label. The following code snippet sets up the initial dependencies and the actual and predicted labels:

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

actual_labels = ['spam', 'ham', 'spam', 'spam', 'spam',
                 'ham',  'ham', 'spam', 'ham',  'spam',
                 'spam', 'ham', 'ham',  'ham',  'spam',
                 'ham',  'ham', 'spam', 'spam', 'ham']

predicted_labels = ['spam', 'spam', 'spam', 'ham',  'spam',
                    'spam', 'ham',  'ham',  'spam', 'spam',
                    'ham',  'ham',  'spam', 'ham',  'ham',
                    'ham',  'spam', 'ham',  'spam', 'spam']

ac = Counter(actual_labels)
pc = Counter(predicted_labels)
```

Let us now see the total number of emails belonging to either `'spam'` or `'ham'` based on the actual labels and our predicted labels using the following snippet:

```
In [517]: print 'Actual counts:', ac.most_common()
     ...: print 'Predicted counts:', pc.most_common()
Actual counts: [('ham', 10), ('spam', 10)]
Predicted counts: [('spam', 11), ('ham', 9)]
```

Thus we see that there are a total of 10 emails that are `'spam'` and 10 emails that are `'ham'`. Our classifier has predicted a total of 11 emails as `'spam'` and 9 as `'ham'`. How do we now compare which email was actually `'spam'` and what it was classified as? A confusion matrix is an excellent way to measure this performance across the two classes. A *confusion matrix* is a tabular structure that helps visualize the performance of classifiers. Each column in the matrix represents classified instances based on predictions, and each row of the matrix represents classified instances based on the actual class labels. (It can be vice-versa if needed.) We usually have a class label defined as the *positive class*, which could be typically the class of our interest. Figure 4-5 shows a typical two-class confusion matrix where (p) denotes the positive class and (n) denotes the negative class.

	p' (Predicted)	n' (Predicted)
p (Actual)	True Positive	False Negative
n (Actual)	False Positive	True Negative

Figure 4-5. *A confusion matrix from a two-class classification problem*

You can see some terms in the matrix depicted in Figure 4-5. True Positive (TP) indicates the number of correct hits or predictions for our positive class. False Negative (FN) indicates the number of instances we missed for that class by predicting it falsely as the negative class. False Positive (FP) is the number of instances we predicted wrongly as the positive class when it was actually not. True Negative (TN) is the number of instances we correctly predicted as the negative class.

The following code snippet constructs a confusion matrix with our data:

```
In [519]: cm = metrics.confusion_matrix(y_true=actual_labels,
     ...:                          y_pred=predicted_labels,
     ...:                          labels=['spam','ham'])
     ...: print pd.DataFrame(data=cm,
     ...:                    columns=pd.MultiIndex(levels=[['Predicted:'],
     ...:                                                  ['spam','ham']],
     ...:                                          labels=[[0,0],[0,1]]),
     ...:                    index=pd.MultiIndex(levels=[['Actual:'],
```

```
     ...:                                                      ['spam','ham']],
     ...:                                              labels=[[0,0],[0,1]]))
             Predicted:
                   spam ham
Actual: spam          5   5
        ham           6   4
```

We now get a confusion matrix similar to the figure. In our case, let us consider `'spam'` to be the positive class. We can now define the preceding metrics in the following snippet:

```
positive_class = 'spam'

true_positive = 5.
false_positive = 6.
false_negative = 5.
true_negative = 4.
```

Now that we have the necessary values from the confusion matrix, we can calculate our four performance metrics one by one. We have taken the values from earlier as floats to help with computations involving divisions. We will use the `metrics` module from `scikit-learn`, which is very powerful and helps in computing these metrics with a single function. And we will define and compute these metrics manually so that you can understand them clearly and see what goes on behind the scenes of those functions from the `metrics` module.

Accuracy is defined as the overall accuracy or proportion of correct predictions of the model, which can be depicted by the formula

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

where we have our correct predictions in the numerator divided by all the outcomes in the denominator. The following snippet shows the computations for accuracy:

```
In [522]: accuracy = np.round(
     ...:                  metrics.accuracy_score(y_true=actual_labels,
     ...:                                         y_pred=predicted_labels),2)
     ...: accuracy_manual = np.round(
     ...:                      (true_positive + true_negative) /
     ...:                        (true_positive + true_negative +
     ...:                         false_negative + false_positive),2)
     ...: print 'Accuracy:', accuracy
     ...: print 'Manually computed accuracy:', accuracy_manual
Accuracy: 0.45
Manually computed accuracy: 0.45
```

Precision is defined as the number of predictions made that are actually correct or relevant out of all the predictions based on the positive class. This is also known as *positive predictive value* and can be depicted by the formula

$$Precision = \frac{TP}{TP + FP}$$

where we have our correct predictions in the numerator for the positive class divided by all the predictions for the positive class including the false positives. The following snippet shows the computations for precision:

```
In [523]: precision = np.round(
     ...:                  metrics.precision_score(y_true=actual_labels,
     ...:                                          y_pred=predicted_labels,
     ...:                                          pos_label=positive_
                                                   class),2)
     ...: precision_manual = np.round(
     ...:                          (true_positive) /
     ...:                          (true_positive + false_positive),2)
     ...: print 'Precision:', precision
     ...: print 'Manually computed precision:', precision_manual
Precision: 0.45
Manually computed precision: 0.45
```

Recall is defined as the number of instances of the positive class that were correctly predicted. This is also known as *hit rate*, *coverage*, or *sensitivity* and can be depicted by the formula

$$Recall = \frac{TP}{TP + FN}$$

where we have our correct predictions for the positive class in the numerator divided by correct and missed instances for the positive class, giving us the hit rate. The following snippet shows the computations for recall:

```
In [524]: recall = np.round(
     ...:              metrics.recall_score(y_true=actual_labels,
     ...:                                   y_pred=predicted_labels,
     ...:                                   pos_label=positive_class),2)
     ...: recall_manual = np.round(
     ...:                      (true_positive) /
     ...:                      (true_positive + false_negative),2)
     ...: print 'Recall:', recall
     ...: print 'Manually computed recall:', recall_manual
Recall: 0.5
Manually computed recall: 0.5
```

F1 score is another accuracy measure that is computed by taking the harmonic mean of the precision and recall and can be represented as follows:

$$F1\ Score = \frac{2 \times\ Precision \times Recall}{Precision + Recall}$$

We can compute the same using the following code snippet:

```
In [526]: f1_score = np.round(
     ...:                 metrics.f1_score(y_true=actual_labels,
     ...:                                 y_pred=predicted_labels,
     ...:                                 pos_label=positive_class),2)
     ...: f1_score_manual = np.round(
     ...:                     (2 * precision * recall) /
     ...:                     (precision + recall),2)
     ...: print 'F1 score:', f1_score
     ...: print 'Manually computed F1 score:', f1_score_manual
F1 score: 0.48
Manually computed F1 score: 0.47
```

This should give you a pretty good idea about the main metrics used most often when evaluating classification models. We will be measuring the performance of our models using the very same metrics, and you may remember seeing these metrics from Chapter 3, when we were building some of our taggers and parsers.

Building a Multi-Class Classification System

We have gone through all the steps necessary for building a classification system, from normalization to feature extraction, model building, and evaluation. In this section, we will be putting everything together and applying it on some real-world data to build a multi-class text classification system. For this, we will be using the 20 newsgroups dataset available for download using `scikit-learn`. The 20 newsgroups dataset comprises around 18,000 newsgroups posts spread across 20 different categories or topics, thus making this a 20-class classification problem! Remember the more classes, the more complex or difficult trying to build an accurate classifier gets. It is recommended that you remove the headers, footers, and quotes from the text documents to prevent the model from overfitting or not generalizing well due to certain specific headers or email addresses, so we will make sure we take care of this. We will also remove documents that are empty or have no content after removing these three items because it would be pointless to try and extract features from empty documents.

Let us start with loading the necessary dataset and defining functions for building the training and testing datasets:

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.cross_validation import train_test_split
```

```
def get_data():
    data = fetch_20newsgroups(subset='all',
                              shuffle=True,
                              remove=('headers', 'footers', 'quotes'))
    return data

def prepare_datasets(corpus, labels, test_data_proportion=0.3):
    train_X, test_X, train_Y, test_Y = train_test_split(corpus, labels,
                                                        test_size=0.33,
random_state=42)
    return train_X, test_X, train_Y, test_Y

def remove_empty_docs(corpus, labels):
    filtered_corpus = []
    filtered_labels = []
    for doc, label in zip(corpus, labels):
        if doc.strip():
            filtered_corpus.append(doc)
            filtered_labels.append(label)

    return filtered_corpus, filtered_labels
```

We can now get the data, see the total number of classes in our dataset, and split our data into training and test datasets using the following snippet (in case you do not have the data downloaded, feel free to connect to the Internet and take some time to download the complete corpus):

```
# get the data
In [529]: dataset = get_data()

# print all the classes
In [530]: print dataset.target_names
['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys.ibm.
pc.hardware', 'comp.sys.mac.hardware', 'comp.windows.x', 'misc.forsale',
'rec.autos', 'rec.motorcycles', 'rec.sport.baseball', 'rec.sport.hockey',
'sci.crypt', 'sci.electronics', 'sci.med', 'sci.space', 'soc.religion.
christian', 'talk.politics.guns', 'talk.politics.mideast', 'talk.politics.
misc', 'talk.religion.misc']

# get corpus of documents and their corresponding labels
In [531]: corpus, labels = dataset.data, dataset.target
     ...: corpus, labels = remove_empty_docs(corpus, labels)

# see sample document and its label index, name
In [548]: print 'Sample document:', corpus[10]
     ...: print 'Class label:',labels[10]
     ...: print 'Actual class label:', dataset.target_names[labels[10]]
Sample document: the blood of the lamb.
```

```
This will be a hard task, because most cultures used most animals
for blood sacrifices. It has to be something related to our current
post-modernism state. Hmm, what about used computers?

Cheers,
Kent
Class label: 19
Actual class label: talk.religion.misc

# prepare train and test datasets
In [549]: train_corpus, test_corpus, train_labels, test_labels = prepare_
datasets(corpus,
     ...:                                                        labels, test_
data_proportion=0.3)
```

You can see from the preceding snippet how a sample document and label looks. Each document has its own class label, which is one of the 20 topics it is categorized into. The labels obtained are numbers, but we can easily map it back to the original category name if needed using the preceding snippet. We also split our data into train and test datasets, where the test dataset is 30 percent of the total data. We will build our model on the training data and test its performance on the test data. In the following snippet, we will use the normalization module we built earlier to normalize our datasets:

```
from normalization import normalize_corpus

norm_train_corpus = normalize_corpus(train_corpus)
norm_test_corpus = normalize_corpus(test_corpus)
```

Remember, a lot of normalization steps take place that we implemented earlier for each document in the corpora, so it may take some time to complete. Once we have normalized documents, we will use our feature extractor module built earlier to start extracting features from our documents. We will build models for Bag of Words, TF-IDF, averaged word vector, and TF-IDF weighted averaged word vector features separately and compare their performances.

The following snippet extracts necessary features based on the different techniques:

```
from feature_extractors import bow_extractor, tfidf_extractor
from feature_extractors import averaged_word_vectorizer
from feature_extractors import tfidf_weighted_averaged_word_vectorizer
import nltk
import gensim

# bag of words features
bow_vectorizer, bow_train_features = bow_extractor(norm_train_corpus)
bow_test_features = bow_vectorizer.transform(norm_test_corpus)
```

```
# tfidf features
tfidf_vectorizer, tfidf_train_features = tfidf_extractor(norm_train_corpus)
tfidf_test_features = tfidf_vectorizer.transform(norm_test_corpus)

# tokenize documents
tokenized_train = [nltk.word_tokenize(text)
                   for text in norm_train_corpus]
tokenized_test = [nltk.word_tokenize(text)
                   for text in norm_test_corpus]
# build word2vec model
model = gensim.models.Word2Vec(tokenized_train,
                               size=500,
                               window=100,
                               min_count=30,
                               sample=1e-3)

# averaged word vector features
avg_wv_train_features = averaged_word_vectorizer(corpus=tokenized_train,
                                                 model=model,
                                                 num_features=500)
avg_wv_test_features = averaged_word_vectorizer(corpus=tokenized_test,
                                                model=model,
                                                num_features=500)

# tfidf weighted averaged word vector features
vocab = tfidf_vectorizer.vocabulary_
tfidf_wv_train_features =
tfidf_weighted_averaged_word_vectorizer(corpus=tokenized_train,

tfidf_vectors=tfidf_train_features,

tfidf_vocabulary=vocab, model=model,

num_features=500)
tfidf_wv_test_features =
tfidf_weighted_averaged_word_vectorizer(corpus=tokenized_test,

tfidf_vectors=tfidf_test_features,

tfidf_vocabulary=vocab, model=model,

num_features=500)
```

Once we extract all the necessary features from our text documents using the preceding feature extractors, we define a function that will be useful for evaluation our classification models based on the four metrics discussed earlier, as shown in the following snippet:

```
from sklearn import metrics
import numpy as np

def get_metrics(true_labels, predicted_labels):
```

```
    print 'Accuracy:', np.round(
                        metrics.accuracy_score(true_labels,
                                               predicted_labels),
                        2)
    print 'Precision:', np.round(
                        metrics.precision_score(true_labels,
                                               predicted_labels,
                                               average='weighted'),
                        2)
    print 'Recall:', np.round(
                        metrics.recall_score(true_labels,
                                               predicted_labels,
                                               average='weighted'),
                        2)
    print 'F1 Score:', np.round(
                        metrics.f1_score(true_labels,
                                               predicted_labels,
                                               average='weighted'),
                        2)
```

We now define a function that trains the model using an ML algorithm and the training data, performs predictions on the test data using the trained model, and then evaluates the predictions using the preceding function to give us the model performance:

```
def train_predict_evaluate_model(classifier,
                                 train_features, train_labels,
                                 test_features, test_labels):
    # build model
    classifier.fit(train_features, train_labels)
    # predict using model
    predictions = classifier.predict(test_features)
    # evaluate model prediction performance
    get_metrics(true_labels=test_labels,
                predicted_labels=predictions)
    return predictions
```

We now import two ML algorithms (discussed in detail earlier) so that we can start building our models with them based on our extracted features. We will be using `scikit-learn` as mentioned to import the necessary classification algorithms, saving us the time and effort that would have been spent otherwise reinventing the wheel:

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import SGDClassifier

mnb = MultinomialNB()
svm = SGDClassifier(loss='hinge', n_iter=100)
```

Now we will train, predict, and evaluate models for all the different types of features using both multinomial naïve Bayes and support vector machines using the following snippet:

```
# Multinomial Naive Bayes with bag of words features
In [558]: mnb_bow_predictions = train_predict_evaluate_model(classifier=mnb,
     ...:                                                     train_features=bow_
                                                              train_features,
     ...:                                                     train_labels=train_
                                                              labels,
     ...:                                                     test_features=bow_test_
                                                              features,
     ...:                                                     test_labels=test_
                                                              labels)
Accuracy: 0.67
Precision: 0.72
Recall: 0.67
F1 Score: 0.65

# Support Vector Machine with bag of words features
In [559]: svm_bow_predictions = train_predict_evaluate_model(classifier=svm,
     ...:                                                     train_features=bow_
                                                              train_features,
     ...:                                                     train_labels=train_
                                                              labels,
     ...:                                                     test_features=bow_test_
                                                              features,
     ...:                                                     test_labels=test_
                                                              labels)
Accuracy: 0.61
Precision: 0.66
Recall: 0.61
F1 Score: 0.62

# Multinomial Naive Bayes with tfidf features
In [560]: mnb_tfidf_predictions = train_predict_evaluate_
model(classifier=mnb,
     ...:                                                     train_features=tfidf_
                                                              train_features,
     ...:                                                     train_labels=train_
                                                              labels,
     ...:                                                     test_features=tfidf_
                                                              test_features,
     ...:                                                     test_labels=test_
                                                              labels)
Accuracy: 0.72
```

```
Precision: 0.78
Recall: 0.72
F1 Score: 0.7

# Support Vector Machine with tfidf features
In [561]: svm_tfidf_predictions = train_predict_evaluate_
model(classifier=svm,
     ...:                                            train_features=tfidf_
                                                     train_features,
     ...:                                            train_labels=train_
                                                     labels,
     ...:                                            test_features=tfidf_
                                                     test_features,
     ...:                                            test_labels=test_
                                                     labels)
Accuracy: 0.77
Precision: 0.77
Recall: 0.77
F1 Score: 0.77

# Support Vector Machine with averaged word vector features
In [562]: svm_avgwv_predictions = train_predict_evaluate_
model(classifier=svm,
     ...:                                            train_features=avg_wv_
                                                     train_features,
     ...:                                            train_labels=train_
                                                     labels,
     ...:                                            test_features=avg_wv_
                                                     test_features,
     ...:                                            test_labels=test_
                                                     labels)
Accuracy: 0.55
Precision: 0.55
Recall: 0.55
F1 Score: 0.52

# Support Vector Machine with tfidf weighted averaged word vector features
In [563]: svm_tfidfwv_predictions = train_predict_evaluate_model(classifier
=svm,
     ...:
train_features=tfidf_wv_train_features,
     ...:
train_labels=train_labels, test_features=tfidf_wv_test_features,
     ...:                        test_labels=test_labels)
Accuracy: 0.53
Precision: 0.55
Recall: 0.53
F1 Score: 0.52
```

We built a total of six models using various types of extracted features and evaluated the performance of the model on the test data. From the preceding results, we can see that the SVM-based model built using TF-IDF features yielded the best results of 77 percent accuracy as well as precision, recall, and F1 score. We can build the confusion matrix for our SVM TF-IDF–based model to get an idea of the classes for which our model might not be performing well:

```
In [597]: import pandas as pd
     ...: cm = metrics.confusion_matrix(test_labels, svm_tfidf_predictions)
     ...: pd.DataFrame(cm, index=range(0,20), columns=range(0,20))
Out[597]:
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	157	3	0	1	1	0	2	3	4	1	4	4	1	4	5	34	3	7	7	22
1	1	225	8	7	8	14	8	0	2	1	0	2	5	4	4	1	4	0	3	0
2	1	20	219	19	9	18	8	1	0	0	0	3	5	2	3	2	1	1	2	0
3	1	11	25	223	9	5	8	2	1	1	1	2	6	3	1	0	1	0	0	0
4	0	4	7	15	228	6	5	2	3	1	0	3	9	3	3	1	1	0	1	0
5	0	21	18	1	2	272	0	1	1	0	0	0	4	3	1	0	0	1	0	0
6	0	2	7	11	12	2	269	10	3	2	1	1	10	1	4	0	2	1	1	0
7	1	5	2	2	2	3	4	247	19	1	3	2	9	3	2	0	3	3	4	1
8	3	1	0	4	2	2	5	27	252	3	4	2	1	4	1	3	2	2	4	0
9	2	1	1	0	2	3	4	3	6	277	12	2	1	1	2	4	2	0	2	0
10	0	0	0	0	0	0	1	3	2	4	282	1	2	1	4	1	0	1	1	0
11	3	5	3	3	1	2	2	2	2	3	0	259	6	2	0	1	5	2	5	0
12	1	6	6	15	7	2	13	10	8	4	4	2	211	4	5	1	1	1	0	1
13	2	4	0	1	2	4	3	0	2	1	1	1	7	268	4	2	3	0	3	0
14	0	5	3	0	2	4	2	5	4	1	2	0	8	3	264	2	4	1	3	1
15	11	1	0	0	1	1	0	0	4	1	2	2	1	7	5	291	4	4	3	5
16	4	1	0	0	0	4	2	1	7	2	2	11	3	2	4	2	227	3	13	3
17	6	0	1	0	1	3	0	2	3	2	4	6	1	3	1	6	5	259	10	2
18	10	1	2	1	0	1	2	1	5	3	3	7	0	9	6	4	33	7	164	3
19	21	5	0	1	0	2	4	3	7	2	1	1	0	11	3	57	22	7	3	63

Figure 4-6. *20-class confusion matrix for our SVM based model*

From the confusion matrix shown in Figure 4-6, we can see a large number of documents for class label 0 that got misclassified to class label 15, and similarly for class label 18, many documents got misclassified into class label 16. Many documents for class label 19 got misclassified into class label 15. On printing the class label names for them, we can observe the following output:

```
In [600]: class_names = dataset.target_names
     ...: print class_names[0], '->', class_names[15]
     ...: print class_names[18], '->', class_names[16]
     ...: print class_names[19], '->', class_names[15]
alt.atheism -> soc.religion.christian
talk.politics.misc -> talk.politics.guns
talk.religion.misc -> soc.religion.christian
```

From the preceding output we can see that the misclassified categories are not vastly different from the actual correct category. Christian, religion, and atheism are based on some concepts related to the existence of God and religion and possibly have similar features. Talks about miscellaneous issues and guns related to politics also must be

having similar features. We can further analyze and look at the misclassified documents in detail using the following snippet (due to space constraints I only include the first few misclassified documents in each case):

```
In [621]: import re
     ...: num = 0
     ...: for document, label, predicted_label in zip(test_corpus, test_
labels, svm_tfidf_predictions):
     ...:     if label == 0 and predicted_label == 15:
     ...:         print 'Actual Label:', class_names[label]
     ...:         print 'Predicted Label:', class_names[predicted_label]
     ...:         print 'Document:-'
     ...:         print re.sub('\n', ' ', document)
     ...:         print
     ...:         num += 1
     ...:         if num == 4:
     ...:             break
     ...:
     ...:
Actual Label: alt.atheism
Predicted Label: soc.religion.christian
Document:-
I would like a list of Bible contadictions from those of you who dispite
being free from Christianity are well versed in the Bible.

Actual Label: alt.atheism
Predicted Label: soc.religion.christian
Document:-
  They spent quite a bit of time on the wording of the Constitution.  They
picked words whose meanings implied the intent.  We have already looked in
the dictionary to define the word.  Isn't this sufficient?   But we were
discussing it in relation to the death penalty.  And, the Constitution need
not define each of the words within.  Anyone who doesn't know what cruel is
can look in the dictionary (and we did).

Actual Label: alt.atheism
Predicted Label: soc.religion.christian
Document:-
Our Lord and Savior David Keresh has risen!     He has been seen
alive!         Spread the word!     ----------------------------
-----------------------------------------------------------------

Actual Label: alt.atheism
Predicted Label: soc.religion.christian
Document:-
  "This is your god" (from John Carpenter's "They Live," natch)

In [623]: num = 0
```

```
    ...: for document, label, predicted_label in zip(test_corpus, test_
labels, svm_tfidf_predictions):
    ...:     if label == 18 and predicted_label == 16:
    ...:         print 'Actual Label:', class_names[label]
    ...:         print 'Predicted Label:', class_names[predicted_label]
    ...:         print 'Document:-'
    ...:         print re.sub('\n', ' ', document)
    ...:         print
    ...:         num += 1
    ...:         if num == 4:
    ...:             break
    ...:
    ...:
Actual Label: talk.politics.misc
Predicted Label: talk.politics.guns
Document:-
After the initial gun battle was over, they had 50 days to come out
peacefully. They had their high priced lawyer, and judging by the posts here
they had some public support. Can anyone come up with a rational explanation
why the didn't come out (even after they negotiated coming out after the
radio sermon) that doesn't include the Davidians wanting to commit suicide/
murder/general mayhem?

Actual Label: talk.politics.misc
Predicted Label: talk.politics.guns
Document:-
Yesterday, the FBI was saying that at least three of the bodies had gunshot
wounds, indicating that they were shot trying to escape the fire.  Today's
paper quotes the medical examiner as saying that there is no evidence of
gunshot wounds in any of the recovered bodies.  At the beginning of this
siege, it was reported that while Koresh had a class III (machine gun)
license, today's paper quotes the government as saying, no, they didn't have
a license.  Today's paper reports that a number of the bodies were found
with shoulder weapons next to them, as if they had been using them while
dying -- which doesn't sound like the sort of action I would expect from a
suicide.  Our government lies, as it tries to cover over its incompetence
and negligence.  Why should I believe the FBI's claims about anything else,
when we can see that they are LYING?  This system of government is beyond
reform.

Actual Label: talk.politics.misc
Predicted Label: talk.politics.guns
Document:-
  Well, for one thing most, if not all the Dividians (depending on whether
they could show they acted in self-defense and there were no illegal
weapons), could have gone on with their life as they were living it. No one
was forcing them to give up their religion or even their legal weapons. The
Dividians had survived a change in leadership before so even if Koresch
```

```
himself would have been convicted and sent to jail, they still could have
carried on.   I don't think the Dividians were insane, but I don't see a
reason for mass suicide (if the fire was intentional set by some of the
Dividians.) We also don't know that, if the fire was intentionally set from
inside, was it a generally know plan or was this something only an inner
circle knew about, or was it something two or three felt they had to do
with or without Koresch's knowledge/blessing, etc.? I don't know much about
Masada. Were some people throwing others over? Did mothers jump over with
their babies in their arms?

Actual Label: talk.politics.misc
Predicted Label: talk.politics.guns
Document:-
rja@mahogany126.cray.com (Russ Anderson) writes...      The fact is that
Koresh and his followers involved themselves   in a gun battle to control
the Mt Carmel complex. That is not   in dispute. From what I remember of the
trial, the authories    couldn't reasonably establish who fired first, the
big reason   behind the aquittal. Mitchell S Todd
```

Thus you can see how to analyze and look at documents that have been misclassified and then maybe go back and tune our feature extraction methods by removing certain words or weighing words differently to reduce or give prominence.

This brings us to the end of our discussion and implementation of our text classification system. Feel free to implement more models using other innovative feature-extraction techniques or supervised learning algorithms and compare their performance.

Applications and Uses

Text classification and categorization is used in several real-world scenarios and applications, including the following:

- News articles categorization
- Spam filtering
- Music or movie genre categorization
- Sentiment analysis
- Language detection

The possibilities with text data are indeed endless, and with a little effort you can apply classification to solve various problems and automate otherwise time-consuming operations and scenarios.

Summary

Text classification is indeed a powerful tool, and we have covered almost all aspects related to it in this chapter. We started off our journey with look at the definition and scope of text classification. Next, we defined automated text classification as a supervised learning problem and looked at the various types of text classification. We also briefly covered some ML concepts related to the various types of algorithms. A typical text classification system blueprint was also defined to describe the various modules and steps involved when building an end-to-end text classifier. Each module in the blueprint was then expanded upon. Normalization was touched upon in detail in Chapter 3, and we built a normalization module here specially for text classification. Various feature-extraction techniques were explored in detail, including Bag of Words, TF-IDF, and advanced word vectorization techniques.

You should now be clear about not only the mathematical representations and concepts but also ways to implement them using our code samples. Various supervised learning methods were discussed with focus on multinomial naïve Bayes and support vector machines, which work well with text data, and we looked at ways to evaluate classification model performance and even implemented those metrics. Finally, we put everything we learned together into building a robust 20-class text classification system on real data, evaluated various models, and analyzed model performance in detail. We wrapped up our discussion by looking at some areas where text classification is used frequently.

We have just scratched the surface of text analytics here with classification. We will be looking at more ways to analyze and derive insights from textual data in future chapters.