Chapter 8
Text Mining

8.1 Introduction

In this chapter we consider the problem of analyzing text data. Text is perhaps the most ubiquitous form of data. The amount of text data available is staggering: from books or articles, to web pages, to social media. All of these form rich sources of data which we can exploit to gain valuable information. Analysis of text data has many practical applications including information retrieval, social network analysis, spam filtering, and sentiment analysis.

Text mining, or text analytics as it is alternatively known, is the task of extracting information from text data. In the previous chapters, we looked at using R for performing data analysis in various settings. There was a common element in all of the settings: the input data that we used was almost always *structured*, i.e., consisting of data points for a given set of variables either numeric or categorical. Text data is *unstructured*. Although text written in any human language is governed by the rules of grammar, it lacks the clearly defined variables that we have when analyzing structured data. Apart from linguistic rules, text carries information, such as describing an object or presenting a point of view. In turn, the goal of text mining is to extract this latent information from text data as objectively as possible.

Text mining is an umbrella term for different types of analysis that we perform over text. The analysis tasks range from simple word-frequency based analysis to more complex tasks such as classification. In fact, we can easily adapt classical data analysis techniques such as exploratory data analysis and classification when working with text data. We only need to use the proper representation of text so that the unstructured text data can be encoded as structured or semi-structured data. In this chapter we will look at various data structures to represent text documents.

The text mining functionality in R is built around the `tm` package which we review in this chapter.[1] The `tm` package has a well-designed data structures and utility

[1] Different versions of the tm package have slight differences in function names. In this chapter we use tm package version 0.5–10.

M. A. Pathak, *Beginning Data Science with R,*
DOI 10.1007/978-3-319-12066-9_8

functions for representing text data. Another benefit of `tm` is that it is convenient to export the data so that it can be used in other R functions and packages.

8.2 Dataset

In this chapter, we look at a collection of reviews for 2000 movies.[2] The reviews are classified into positive and negative according to their overall sentiment. There are equal number of positive and negative reviews. Due to the sentiment labeling, this dataset leads us to various interesting analysis tasks including identifying the most positive and negative words, and classifying if a given document has a positive or negative sentiment.

The movie review dataset is located in the directory `review_polarity`. All of the movie reviews are in the form of flat text files. The positive movie reviews are located in the directory `review_polarity/txt_sentoken/pos/`, while the negative movie reviews are located in the directory `review_polarity/txt_sentoken/neg/`.

Before we begin our analysis, we load the `tm` package using the library function.

```
> library(tm)
```

8.3 Reading Text Input Data

The fundamental data structure of tm is the TextDocument object, which, as the name suggests, represents a text document. After that, we have the Corpus object which represents a collection of text documents.[3] Finally, we have the TextRepository object which is a collection of multiple Corpus objects.

Text documents are available in heterogeneous data formats: from ubiquitous plain text, PDF, HTML formats to more special purpose formats such as RCV1. Plain text is the easiest format to process as it contains text alone without additional metadata. In many other text processing packages, we first need to use external tools to convert data into plain text. One of the strengths of the tm package is its support to read data from various file formats. This functionality is available through the reader functions for each file format. We can obtain the list of reader functions using getReaders().

[2] We use the polarity dataset v2.0 available for download at http://www.cs.cornell.edu/people/pabo/movie-review-data/.

[3] In linguistics, corpus, which literally means a body, is the term used to refer to a collection of documents.

```
> getReaders()
[1] "readDOC"
[2] "readPDF"
[3] "readReut21578XML"
[4] "readReut21578XMLasPlain"
[5] "readPlain"
[6] "readRCV1"
[7] "readRCV1asPlain"
[8] "readTabular"
[9] "readXML"
```

Apart from file formats, text documents are available from different sources: text files, web pages with a URL, or even in existing R data frame or vector objects. The data frames are useful for intermediate data storage when we are using an external source such as an SQL database. In tm, the data source is encapsulated as the Source object. We have different types of Source objects for each specialized data source. We can see the list of the available Source objects using getSources().

```
> getSources()
[1] "DataframeSource" "DirSource"
[3] "ReutersSource"   "URISource"
[5] "VectorSource"
```

We use DirSource when we have a directory containing text documents as the case is with our dataset.

To create a Corpus object, we require a Source object and a reader function. If we do not specify a reader function, readPlain() is used by default. We create a corpus of positive documents from our dataset directory `review_polarity/txt_sentoken/pos`. As we discussed above, this directory contains 1000 text files.

```
> source.pos = DirSource('review_polarity/txt_
               sentoken/pos')
> corpus.pos = Corpus(source.pos)
```

Similarly, we create the corpus of negative documents from the directory `'review_polarity/txt_sentoken/neg`.

```
> source.neg = DirSource('review_polarity/txt_
               sentoken/neg')
> corpus.neg = Corpus(source.neg)
```

The corpus object `corpus.pos` as well as `corpus.neg` contain the 1000 text documents. Printing the corpus.pos object confirms this with a human-readable message.

```
> corpus.pos
A corpus with 1000 text documents
```

The corpus object is indexable as a vector, so we can use the length() function to determine its length.

```
> length(corpus.pos)
[1] 1000
```

Similarly, we use vector indexing to obtain individual text documents in the corpus object.

```
> corpus.pos[1]
A corpus with 1 text document
```

Printing the text document `corpus.pos[1]` on the console is equivalent to using the show() function.

```
> show(corpus.pos[1])
A corpus with 1 text document
```

The inspect() function allows us to view the contents of the text document along with its metadata. As we have not yet assigned the metadata to this document, we can only see the text contents after the file name `$cv000_29590.txt`. This is the same text that we can see if we opened the file `review_polarity/txt_sentoken/pos/cv000_29590.txt` in a text editor.

```
> inspect(corpus.pos[1])
A corpus with 1 text document

The metadata consists of 2 tag-value pairs and a data frame
Available tags are:
  create_date creator
Available variables in the data frame are:
  MetaID

$cv000_29590.txt
films adapted from comic books have had plenty of success,
whether they're about superheroes (batman, superman, spawn), or
geared toward kids (casper) or the arthouse crowd (ghost world),
but there's never really been a comic book like from hell before.
for starters, it was created by alan moore (and eddie campbell),
who brought the medium to a whole new level in the mid '80s with a
12-part series called the watchmen.
to say moore and campbell thoroughly researched the subject of jack
the ripper would be like saying michael jackson is starting to look a
little odd.
...
```

8.4 Common Text Preprocessing Tasks

Once we have our text documents loaded into R, there are a few preprocessing tasks that we need to do before we can begin our analysis. In most cases, these tasks are a set of transformations that we apply on the text document. For example, when we are trying to find the frequencies of word occurences in a corpus, it makes sense to remove the punctuation symbols from the text, otherwise commonly occurring punctuation symbols such as full-stop and comma would also start appearing in our word counts.

The tm package has the `tm_map()` function that applies a transformation to a corpus or a text document and also takes a transformation function as input. To remove the punctuation symbols, we use the `removePunctuation()` function.

```
> doc = tm_map(corpus.pos[1],removePunctuation)
> inspect(doc)
A corpus with 1 text document

The metadata consists of 2 tag-value pairs and a data frame
Available tags are:
  create_date creator
Available variables in the data frame are:
  MetaID

$cv000_29590.txt
films adapted from comic books have had plenty of success whether
theyre about superheroes batman superman spawn or geared toward
kids casper or the arthouse crowd ghost world but theres never
really been a comic book like from hell before
for starters it was created by alan moore and eddie campbell who
brought the medium to a whole new level in the mid 80s with a 12part
series called the watchmen
to say moore and campbell thoroughly researched the subject of jack
the ripper would be like saying michael jackson is starting to look a
little odd
...
```

As compared to the output of `inspect(corpus.pos[1])`, all of the punctuation symbols are now removed. The `tm_map()` function does not make any changes to the input document but returns another text document with the modified text.

The `tm_map()` function supports many other transformations: we can see the list of available transformations using the getTransformations() function.

```
> getTransformations()
[1] "as.PlainTextDocument"
[2] "removeNumbers"
[3] "removePunctuation"
[4] "removeWords"
```

```
[5] "stemDocument"
[6] "stripWhitespace"
```

Two other important transformations are stop word removal using the removeWords() transformation and stemming using the stemDocument() transformation.

8.4.1 Stop Word Removal

All words in a text document are not equally important: while some words carry meaning, others play only a supporting role. The words in the latter category are called stop words. The list of stop words is of course specific to a language. Examples of stop words in English include "the," "is," "a," "an." As these words occur very frequently in all documents, it is better to first remove them from the text.

There are lists of stop words compiled for most languages. The `tm` package makes such lists available for English and a few other European languages through the stopwords() function. We need to specify the language name as an argument to this function, but it returns the list for English by default.

```
> stopwords()
  [1] "i"          "me"
  [3] "my"         "myself"
  [5] "we"         "our"
  [7] "ours"       "ourselves"
  [9] "you"        "your"
 [11] "yours"      "yourself"
 [13] "yourselves" "he"
 [15] "him"        "his"
 [17] "himself"    "she"
 [19] "her"        "hers"
 [21] "herself"    "it"
...
```

To remove the stop words from a text document, we use the removeWords() transformation in the `tm_map()` function. We also need to provide the list of stopwords as the third argument.

```
> doc = tm_map(doc,removeWords,stopwords())
> inspect(doc)
A corpus with 1 text document
...
$cv000_29590.txt
films adapted comic books plenty success whether theyre
superheroes batman superman spawn geared toward kids casper
arthouse crowd ghost world theres never really comic book like
hell starters created alan moore eddie campbell brought medium
 whole new level mid 80s 12part series called watchmen
 say moore campbell thoroughly researched subject jack ripper
```

```
 like saying michael jackson starting look little odd
 book graphic novel will 500 pages long includes nearly 30
 consist nothing footnotes words dont dismiss film source
...
```

As we can see, the words such as "from," "have," "had" have been removed from the text.

In the example above, we used the generic list of English stop words. In many scenarios, this is not ideal as we have *domain-specific* stop words. In a corpus of movie reviews, words such as "movie" or "film" occur fairly often, but are devoid of useful content, as most of the documents would contain them. We therefore need to do a frequency analysis to identify such commonly occuring words in a corpus. We shall see an example of frequency analysis further in the chapter.

It is straightforward to remove our own list of stopwords in addition to the generic list available in the stopwords() function. We simply concatenate the two lists using the c() function.

```
> doc = tm_map(doc,removeWords,
               c(stopwords(),'film','films','movie','movies'))
> inspect(doc)
A corpus with 1 text document
...
$cv000_29590.txt
 adapted comic books plenty success whether theyre superheroes
batman superman spawn geared toward kids casper arthouse
crowd ghost world theres ever really comic book like hell
 starters created alan moore eddie campbell brought medium
whole new level mid 80s 12part series called watchmen
 say moore campbell thoroughly researched subject jack ripper
like saying michael jackson starting look little odd
...
```

8.4.2 Stemming

In the above example of stopword removal, we need to specify multiple forms of the same word such as "film" and "films." As an alternative to this, we can use a stemming transformation on the document. In linguistics, stemming is the task of replacing each word in a document to its stem or basic form, e.g., "films" by "film." Note that multiple words typically stem to the same word, e.g., the words "filmed," "filming," and "filmings" all would stem to the same word "film."

In most cases, the words and their stemmed counterpart carry a similar meaning or at least share something in common; in the above example, they have something to do with a "film." Stemming greatly reduces the redundancy in text. This finds applications in many NLP tasks: in the case of word frequency analysis over a stemmed corpus, the counts of individual words get added up in the counts for their stemmed form. If the word "film" occurs 100 times in the original corpus, and "films" occurs 50 times, in the stemmed corpus, the count for the word "films" will

be 150, which is a more representative count. Stemming plays an important role in information retrieval: most search engines perform stemming on the search queries and text documents so that we can find documents containing different forms of the same words that occur in the query. A search using the word "cats" should also match a document with the word "cat."

Automatic stemming algorithms or stemmers have more than a 50 year history. The most popular stemmer for English is the Porter stemmer. There are software implementations of the Porter stemmer for most programming environments. In the tm package, the Porter stemmer is available through the stemDocument() transformation.

We use the stemDocument() function as an argument to `tm_map()`.

```
> doc = tm_map(doc,stemDocument)
> inspect(doc)
A corpus with 1 text document
...
$cv000_29590.txt
adapt comic book plenti success whether theyr superhero batman
superman spawn gear toward kid casper arthous crowd ghost
world there never realli comic book like hell
starter creat alan moor eddi campbel brought medium whole
new level mid 80s 12part seri call watchmen
say moor campbel thorough research subject jack ripper like say
michael jackson start look littl odd
...
```

As we can see words are replaced by their stemmed versions, e.g., "adapted" by "adapt" and "books" by "book."

8.5 Term Document Matrix

The term document matrix is a fundamental data structure in text mining. In natural language processing, the term document matrix is also referred to as *bag of words*. Term document matrix is the data representation where the text data is stored with some form of structure, making it the bridge between unstructured data and structured data. Most of the text mining algorithms involve performing some computation on this representation.

As the name suggests, the term document matrix is a matrix containing individual words or terms as rows and documents as columns. The number of rows in the matrix is the number of unique words in the corpus, and the number of columns is the number of documents. The entry in the matrix is determined by the weighting function. By default, we use the term frequency, that is the count of how many times the respective word occurs in the document. If a document A contains the text "this is a cat, that is a cat.", the column for document A in the term document matrix will contain entries with value 1 each for "this" and "that," and entries with value 2 for "is," "a," and "cat." In this way, the term document matrix only contains the word

occurence counts and ignores the order in which these words occur. An alternative formulation is the document term matrix, with the documents as rows and individual terms as columns. In this section, we only consider term document matrices.

In tm, the term document matrix is represented by the TermDocumentMatrix object. We create a term document matrix for either a document or a corpus using its constructor function TermDocumentMatrix(). We create the term document matrix for the corpus of positive sentiment documents `corpus.pos` below.

```
> tdm.pos = TermDocumentMatrix(corpus.pos)
> tdm.pos
A term-document matrix (36469 terms, 1000 documents)

Non-/sparse entries: 330003/36138997
Sparsity           : 99%
Maximal term length: 79
Weighting          : term frequency (tf)
```

As with TextDocument objects, printing a TermDocumentMatrix object displays summary information. There are 36,469 terms in corpus with 1000 documents. The nonsparse entries refer to the number of entries in the matrix with nonzero values. Sparsity is the percentage of those entries over all the entries in the matrix. As we can see the sparsity is 99 %. This is a common phenomenon with text corpora as most of them are very sparse. Most documents of our corpus contain only a few hundred terms, so the remaining 36,000 or so entries for each document are zero. Maximal term length is the length of the longest word. As we mentioned above, we are using the term frequency weighting function.

We view the contents of the TermDocumentMatrix object using the inspect() function. As displaying the TermDocumentMatrix for the entire corpus of 1000 documents takes a lot of space, we only display the TermDocumentMatrix computed over the corpus of the three documents here.

```
> tdm.pos3 = TermDocumentMatrix(corpus.pos[1:3])
> inspect(tdm.pos3)
A term-document matrix (760 terms, 3 documents)

Non-/sparse entries: 909/1371
Sparsity           : 60%
Maximal term length: 16
Weighting          : term frequency (tf)

              Docs
Terms          cv000_29590.txt cv001_18431.txt cv002_15918.txt
  _election                  0               2               0
  _election_                 0               7               0
  _ferris                    0               1               0
  _rushmore_                 0               6               0
  '80s                       1               0               0
  102                        1               0               0
  12-part                    1               0               0
```

```
1888                    1               0               0
500                     1               0               0
abberline               2               0               0
ably                    1               0               0
about                   4               0               2
absinthe                1               0               0
absolutely              0               0               1
accent                  2               0               0
across                  0               0               1
acting                  1               0               1
acts                    1               0               0
actually                1               0               1
adapted                 1               0               0
add                     0               1               0
adding                  0               0               1
...
```

We see that the stop words are present in the term document matrix. There are two ways of removing stopwords and performing additional transformations like stemming and punctuation removal. We can either apply these transformations over the corpus object using `tm_map()` before calling `TermDocumentMatrix()`. The second option is to pass a list of transformation in the `control` parameter of `TermDocumentMatrix()`. We create a term document matrix from the first three documents of `corpus.pos` after removing the stop words, punctuations, and applying stemming.

```
> tdm.pos3 = TermDocumentMatrix(corpus.pos[1:3],
                control=list(stopwords = T,
                removePunctuation = T, stemming = T))

> inspect(tdm.pos3)

A term-document matrix (613 terms, 3 documents)

Non-/sparse entries: 701/1138
Sparsity           : 62%
Maximal term length: 14
Weighting          : term frequency (tf)


              Docs
Terms          cv000_29590.txt cv001_18431.txt cv002_15918.txt
  102                      1               0               0
  12part                   1               0               0
  1888                     1               0               0
  500                      1               0               0
  80s                      1               0               0
  abberlin                 2               0               0
  abli                     1               0               0
  absinth                  1               0               0
  absolut                  0               0               1
  accent                   2               0               0
  across                   0               0               1
```

```
act                    2              0              1
actual                 1              0              1
adapt                  1              0              0
add                    0              1              0
...
```

As we can see, this term document matrix does not contain stopwords, punctuation words starting with _ and only contains stemmed words. We apply the same transformations to the entire corpus object below.

```
> tdm.pos = TermDocumentMatrix(corpus.pos,
                control = list(stopwords = T,
                removePunctuation = T, stemming = T))
> tdm.pos
A term-document matrix (22859 terms, 1000 documents)

Non-/sparse entries: 259812/22599188
Sparsity           : 99%
Maximal term length: 61
Weighting          : term frequency (tf)
```

With these transformations, we see that the number of terms in `tdm.pos` has decreased from 330,003 to 259,812: a 21.27 % reduction. This alone, however, does not significantly reduce the sparsity of the term document matrix.

8.5.1 TF-IDF Weighting Function

In the above term document matrices, we are using the term frequency (TF) weighting function. Another popular weighting function is term frequency-inverse document frequency (TF-IDF) which is widely used in information retrieval and text mining.

As we mentioned above, TF is the count of how many times a term occurs in a document. More often a word appears in the document, higher its TF score. The idea behind inverse document frequency (IDF) is that all words in the corpus are not equally important. IDF is a function of the fraction of documents in which the term appears in. If a corpus has N documents, and a term appears in d of them, the IDF of the term will be $\log(N/d)$.[4] A term appearing in most of the documents will have a low IDF as the ratio N/d will be close to 1. A term that appears in only a handful of documents will have a high IDF score as the N/d ratio will be large.

The TF-IDF weighting score for a term-document pair is given by its term frequency multiplied by the IDF of the term. If a term that infrequently occurs in the corpus occurs in a given document multiple times, that term will get a higher score

[4] log is used as a smoothing function. Alternative formulations of IDF use other such functions.

for the given document. Alternatively, terms that are frequently occuring in the corpus such as stop words will have a low IDF score as they appear in most documents. Even if such words have a high TF score for a document, their TF-IDF score will be relatively low. Due to this, TF-IDF weighting function has an effect similar to stop word removal.

We specify the TF-IDF weighting function `weightTfIdf()` in the control parameter of `TermDocumentMatrix()`.

```
> tdm.pos3 = TermDocumentMatrix(corpus.pos[1:3],
                 control = list(weighting = weightTfIdf,
                 stopwords = T, removePunctuation = T,
                 stemming = T))
> inspect(tdm.pos3)
A term-document matrix (613 terms, 3 documents)

Non-/sparse entries: 665/1174
Sparsity           : 64%
Maximal term length: 14
Weighting          : term frequency - inverse document
                     frequency (normalized)

               Docs
Terms           cv000_29590.txt cv001_18431.txt cv002_15918.txt
  102               0.004032983     0.000000000     0.000000000
  12part            0.004032983     0.000000000     0.000000000
  1888              0.004032983     0.000000000     0.000000000
  500               0.004032983     0.000000000     0.000000000
  80s               0.004032983     0.000000000     0.000000000
  abberlin          0.008065967     0.000000000     0.000000000
  abli              0.004032983     0.000000000     0.000000000
  absinth           0.004032983     0.000000000     0.000000000
  absolut           0.000000000     0.000000000     0.007769424
  accent            0.008065967     0.000000000     0.000000000
  across            0.000000000     0.000000000     0.007769424
  act               0.002976908     0.000000000     0.002867463
  actual            0.001488454     0.000000000     0.002867463
  adapt             0.004032983     0.000000000     0.000000000
  add               0.000000000     0.004620882     0.000000000
  ...
```

The entries in the term document matrix are all fractional due to the TF-IDF weighting function.

Some of the other commonly used weighting functions include binary weighting where we assign a score of 0 or 1 to a term-document pair depending on whether the term appears in the document while ignoring the frequency of occurence. This weighting function is available through `weightBin()`.

8.6 Text Mining Applications

With these data structures at hand, we now look at a few algorithms for text mining.

8.6.1 Frequency Analysis

The first and simplest text mining algorithm is based on counting the number of words in a corpus. Despite its simplicity, frequency analysis provides a good high-level overview of the corpus contents. The most frequently occuring words in the corpus usually indicate what the corpus is about.

We can determine the frequency of occurence for a word simply by summing up the corresponding row in the term document matrix. The `findFreqTerms()` function does a similar computation. It also takes a lower and upper threshold limits as inputs and returns the words with frequencies between those limits.

We first construct the term document matrices for the two corpus objects `corpus.pos` and `corpus.neg`.

```
> tdm.pos = TermDocumentMatrix(corpus.pos,
                  control = list(stopwords = T,
                  removePunctuation = T, stemming = T))

> tdm.neg = TermDocumentMatrix(corpus.neg,
                  control = list(stopwords = T,
                  removePunctuation = T, stemming = T))
```

We apply `findFreqTerms()` to the term document matrix of the corpus of positive reviews `tdm.pos` below. We use a lower threshold of 800 and do not specify the upper threshold. By default, the upper threshold is set to infinity.

```
> findFreqTerms(tdm.pos,800)
 [1] "also"    "best"    "can"     "charact" "come"    "end"
 [7] "even"    "film"    "first"   "get"     "good"    "just"
[13] "life"    "like"    "look"    "love"    "make"    "movi"
[19] "much"    "one"     "perform" "play"    "scene"   "see"
[25] "seem"    "stori"   "take"    "thing"   "time"    "two"
[31] "way"     "well"    "will"    "work"    "year"
```

The `findFreqTerms()` function returns a vector of words sorted in alphabetical order. For the term document matrix `tdm.pos`, the frequent words includes words with positive sentiments such as “best,” “good,” and “well.” This list also includes filler words such as “also,” “can,” “even” that do not carry too much meaning by themselves. These words are present in the term document matrix even after we applied stop word removal, so it appears that these words are not present in the stop word list. Frequency analysis is a useful tool to augment the stop word list for a corpus.

Similarly, we apply `findFreqTerms()` to the term document matrix of the corpus of negative reviews `tdm.neg`.

```
> findFreqTerms(tdm.neg,800)
 [1] "bad"     "can"     "charact" "come"    "end"     "even"
 [7] "film"    "first"   "get"     "good"    "just"    "like"
[13] "look"    "make"    "movi"    "much"    "one"     "play"
[19] "plot"    "scene"   "see"     "seem"    "stori"   "take"
[25] "thing"   "time"    "two"     "way"     "will"
```

We see that the frequent terms contain words with negative sentiment such as "bad," but also contain words such as "good." This indicates that frequency analysis is not alone suitable to determine if a given review is positive or negative.

We can also use frequency analysis to identify frequently co-occuring words. Such words are refered to as word associations. The `findAssocs()` function takes a term document matrix object along with an input word and returns its word associations along with correlation scores. The correlation score is the degree of confidence in the word association. When using `findAssocs()`, we also need to specify a lower limit for the correlation score so that the function only returns the word associations with correlations greater than or equal to that limit. We find the associations for the word "star" with the correlation limit 0.5.

```
> findAssocs(tdm.pos,'star', 0.5)
                         star
trek                     0.63
enterpris                0.57
picard                   0.57
insurrect                0.56
jeanluc                  0.54
androidwishingtobehuman 0.50
anij                     0.50
bubblebath               0.50
crewmat                  0.50
dougherti                0.50
everbut                  0.50
harkonnen                0.50
homeworld                0.50
iith                     0.50
indefin                  0.50
mountaintop              0.50
plunder                  0.50
reassum                  0.50
ruafro                   0.50
soran                    0.50
unsuspens                0.50
verdant                  0.50
youthrestor              0.50
```

We see that the strongest associations for the word "star" are "trek" and "enterpris" arising from the movie Star Trek.

8.6.2 Text Classification

Text classification is the task of classifying text documents into two or more classes based on their content. Similar to the classification techniques we discussed in Chap. 7, we learn a classifier from a set of text documents annotated with class labels. Text classification finds application in a wide variety of settings: spam filtering, where we classify emails into spam or not spam, document categorization, where we classify documents into a prespecified taxonomy, and sentiment analysis. Our movie reviews dataset is a good setting where we can apply sentiment analysis. Using the positive and negative movie reviews, we learn a classifier to predict if a new movie review is positive or negative.

There are many ways to learn a text classifier. We only need to represent the corpus in a way that can be easily consumed by the classifier functions. The document term matrix is a good representation for this purpose as the rows are documents and columns are the words occurring in the corpus. This is very similar to data frames we used in previous chapters, where the data points are the rows and variables are the columns. Using a document term matrix, we can apply all classification algorithms towards learning a text classifier that we studied in Chap. 7.

8.6.2.1 Creating the Corpus and Document Term Matrix

To learn a document classifier, we need to have both positive and negative documents in the same corpus object. This is important because when we construct the document term matrix, we need the documents of both classes represented using the same set of words. If we have separate corpora for positive and negative documents, the respective document term matrices would only contain words occurring in that corpus. It is not convenient to obtain the global document term matrix from the two matrices.

We first obtain a source object for the base directory. By using the argument `recursive=T`, the source object contains the text documents in all the subdirectories.

```
> source = DirSource('review_polarity/',recursive=T)
```

The `source` object contains text files listed in an alphabetical order. We can see this by printing `source`. Due to this, the files containing negative reviews in the directory `review_polarity/txt_sentoken/neg/` are listed before the files containing positive reviews in the directory `review_polarity/txt_sentoken/pos/`. There are 1000 text files in each of these directories.

We obtain the corpus object from `source` below.

```
> corpus = Corpus(source)
```

The first 1000 text documents in `corpus` are negative reviews and the next 1000 text documents are positive reviews.

We obtain the document term matrix for the `corpus` object with the standard preprocessing options: removing stop words and punctuations, stemming, and applying TF-IDF weighting function.

```
> dtm = DocumentTermMatrix(corpus,
              control = list(weighting = weightTfIdf,
              stopwords = T, removePunctuation = T,
              stemming = T))
```

Most classification algorithms do not directly work with document term matrices. We first need to convert these into data frames. We first convert the document term matrix into a regular matrix and then into a data frame `x.df`.

```
> x.df = as.data.frame(as.matrix(dtm))
```

The variables or columns of the data frame are the words of the corpus, whereas the rows are the individual documents. We see this by printing the dimensions of `x.df`.

```
> dim(x.df)
[1]  2000 31255
```

We add an additional variable for the class label of the document: 1 and 0 representing positive and negative documents respectively.[5] As the first 1000 documents are negative reviews and the next 1000 documents are positive reviews, we create a vector of `'neg'` repeated 1000 times followed by `'pos'` repeated 1000 times.

```
> x.df$class_label = c(rep(0,1000),rep(1,1000))
```

8.6.2.2 Creating Train and Test Datasets

Similar to the examples in Chap. 7, we split our data `x.df` into train and test datasets so that we measure how good our classifier is. A simple way to randomly split a data frame is using the `split()` function that takes a variable with true/false values as input.

We construct an `index` variable containing a permutation of the row numbers of the data frame.

```
> index = sample(nrow(x.df))
```

[5] We could also use a factor variable with values "pos" and "neg." We are using 1 and 0 because it is convenient to train a logistic regression model with these class labels.

As `x.df` contains 2000 rows, `index` contains the elements in the sequence 1:2000 in random order.

```
> index
   [1] 1206 1282  576 1243 1063  386 1125 1222 ...
```

We split the data frame so that training dataset contains 1500 documents and test dataset contains 500 documents. For that we use the `split()` function with the argument `index > 500`.

```
> x.split = split(x.df, index > 1500)
```

The output `x.split` is a list of data frames containing the training and test datasets. We assign them respectively to the two data frames `train` and `test`.

```
> train = x.split[[1]]
> test = x.split[[2]]
```

We can check to see if the dimensions of these data frames are as expected.

```
> dim(train)
[1]  1500 31256
> dim(test)
[1]   500 31256
```

Our original data frame had 1000 positive and 1000 negative text documents. As we obtain `train` and `test` data frames by a random split, the distribution of class labels will not necessarily be equal. We use the `table()` function to see the distribution of positive and negative documents in `train`.

```
> table(train$class_label)

  0   1
760 740
```

We see that there are 760 negative documents and 740 positive documents in `train`. Not too far off from a 750-750 split. Similarly, we see that `test` contains the remaining 240 negative and 260 positive documents.

```
> table(test$class_label)

  0   1
240 260
```

8.6.2.3 Learning the Text Classifier

Once we have the data frames for the training data `train`, it is easy to learn the text classifier using any of the classification functions we looked at in Chap. 7. In this section, we train a text classifier using logistic regression and support vector machines (SVMs).

When learning any classification model over text data, one recurrent issue is data dimensionality which is equal to the number of words in the corpus. The `train` data frame is very high dimensional: it contains 31,256 variables, which means there are $1500 \times 31{,}255$ entries in the data frame. We can learn a classifier in R over such a large data frame, but it is fairly expensive in terms of computation time and memory.

A good approximation is to use a subset of features when learning the text classifier. It is an approximation because the classifier using subset of features might not be as good as the classifier learned using the complete set of variables. On the other hand, if we choose the variables in a smart way, this classifier is likely to have an accuracy close to the latter one.

Choosing variables in a classification model falls under the area of feature selection, which is a deep area of machine learning by itself. Here, we use a simple heuristic to select the variables: by using the frequent terms of the corpus. As we have already removed stop words, and are using TF-IDF weighting function, the frequently occuring terms in the corpus are the ones that are the most informative. We find the frequently occuring terms using the `findFreqTerms()` function.

```
> s = findFreqTerms(dtm,4)
> s
  [1] "act"          "action"       "actor"
  [4] "actual"       "alien"        "almost"
  ...
```

We empirically set a lower bound of 4 so that we end up with a reasonable number of variables. This factor is a trade-off between accuracy and computational cost.[6] By using this lower bound, we have 141 frequently occuring terms.

```
> length(s)
[1] 141
```

We add the variable `class_label` to `s` as we would need it to learn the text classifier.

```
> s = c(s,'class_label')
```

We obtain the data frame containing the variables as determined by `s` using `train[,s]`.

We first learn a logistic regression classifier using the `glm()` function. We specify `train[,s]` as the second argument.

```
> model.glm = glm(class_label ~ ., train[,s],
                  family='binomial')
```

Let us look at some of the model coefficients using the `coef()` function.

[6] In practice, the returns on using more number of variables are diminishing; after a point, we only see a marginal improvement in accuracy by adding more variables.

```
> coef(model.glm)
 (Intercept)          act       action        actor
  -0.2245104  -14.4608005   -7.6335646  -33.9385114

      actual        alien       almost         also
 -31.1075874   11.4464657   14.3002912  104.8285575

    although     american         anim        anoth
  43.0003199   42.7002215   10.4214973  -10.1282664

      around      audienc         back          bad
  -5.4040998  -25.4028306   38.8096867 -161.1490557

       becom        begin         best       better
 -10.9462855   -8.9060229   12.3624355  -38.3186012

         big      brother          can         cast
 -22.5387636  -10.1029090  -55.2868626    9.7951972

     charact         come       comedi          day
 -19.1775238  -12.8608680   13.9084740    0.3077193

      ...
```

We see that the word "bad" has a coefficient of −161.15, which is the lowest. According to our model, having one instance "bad" of the text would make the review negative. Similarly, the word "best" has a coefficient of 12.36. This is not true for all positive sentiment words as the word "better" has a negative coefficient of −38.32.

We see how well does this classifier do on our test dataset. We obtain the predictions of our model using the `predict()` function. We normalize these values to 0 and 1, which we then compare to the `class_label` variable of `test`.

```
> p = ifelse(predict(model.glm,test) < 0,0,1)
> table(p == test$class_label)/nrow(test)

FALSE  TRUE
0.268 0.732
```

We see that our model `model.glm` achieves 73.2 % accuracy.

As we discussed earlier, natural language is inherently ambiguous, which makes it hard to do text classification and text mining in general with high accuracy. It is interesting to look at some of the misclassified documents. We see that the model assigns a very low score of −4.15 to one of the positive reviews in the `test` dataset: `pos/cv107_24319.txt`. Using the sigmoid function (Sect. 7.2.2), we see that this score implies that the probability of the review being positive is 0.0155 or 1.55 %. Why does our model classify this document so incorrectly?

The answer to this question is clear when we see the contents of this document.

```
pos/cv107_24319.txt

is evil dead ii a bad movie?
it's full of terrible acting, pointless violence,
and plot holes yet it remains a cult classic nearly
fifteen years after its release.
```

The document contains the words "bad," "terrible," and "pointless," and then goes on to say that the movie is actually good. The presence of such negative sentiment words cause the classifier to label this document with a strong negative score, while the postive words are not sufficient to compensate for this.

Finally, we use support vector machines to learn the text classifier. We use the `svm()` function of the `e1071` package (Sect. 7.2.3). We reuse the `train[,s]` data frame containing the frequently occuring terms as variables. We use the Gaussian or radial kernel with cost = 2.

```
> model.svm = svm(class_label ~ ., train[,s],
                  kernel = 'radial', cost=2)
> model.svm

Call:
svm(formula = class_label ~ ., data = train[, s],
    kernel = "radial", cost = 2)


Parameters:
   SVM-Type:  eps-regression
 SVM-Kernel:  radial
       cost:  2
      gamma:  0.007142857
    epsilon:  0.1
Number of Support Vectors:  1410
```

We can see that the SVM classifier contains a large number of support vectors: 1410.

We also measure the accuracy of this text classifier on the `test` data.

```
> p.svm = ifelse(predict(model.svm,test) < 0.5,0,1)
> table(p.svm == test$class_label)/nrow(test)

FALSE  TRUE
0.256 0.744
```

We see that the SVM classifier has a slightly higher accuracy of 74.4 % as compared to that of the logistic regression classifier.

8.7 Chapter Summary

In this chapter we looked at analyzing text data with R using the `tm` package. When using this package, the first task is to load the text dataset in a Corpus object where each documents is represented by a TextDocument object. Before we begin analyzing the text dataset, it is important to perform certain preprocessing tasks to remove the noise from the data. The preprocessing tasks include: removal of punctuations, stopwords, and performing stemming to replace individual words or terms by their stem or basic form.

For most text analysis tasks, it is useful to represent the corpus as a term document matrix, where the individual terms of the documents are rows and documents are columns. The values stored in the matrix is given by its weighting function; the default weighting function is term frequently (TF) where we store the frequency of occurence of the term in the document. A more powerful weighting function is TF-IDF, which accounts for the importance of terms based on how infrequently the term occurs in the corpus.

The first text mining application we looked at was frequency analysis, where we find frequently occuring terms in a corpus as a glimpse into its overall sentiment. We also looked at finding frequently co-occuring terms for a given term to identify word associations. We also looked at text classification where we learn a classifier to label text documents. Using the term document matrix representation, we can apply all of the classification techniques that we have discussed in the previous chapters.