8
Application of Deep Learning in Text Mining

The current chapter we will cover the following topics:

- Performing preprocessing of textual data and extraction of sentiments
- Analyzing documents using tf-idf
- Performing sentiment prediction using LSTM network
- Application using text2vec examples

Performing preprocessing of textual data and extraction of sentiments

In this section, we will use Jane Austen's bestselling novel Pride and Prejudice, published in 1813, for our textual data preprocessing analysis. In R, we will use the `tidytext` package by Hadley Wickham to perform tokenization, stop word removal, sentiment extraction using predefined sentiment lexicons, **term frequency - inverse document frequency** (**tf-idf**) matrix creation, and to understand pairwise correlations among *n*-grams.

In this section, instead of storing text as a string or a corpus or a **document term matrix** (**DTM**), we process them into a tabular format of one token per row.

How to do it...

Here is how we go about preprocessing:

1. Load the required packages:

```
load_packages=c("janeaustenr","tidytext","dplyr","stringr","ggplot2
","wordcloud","reshape2","igraph","ggraph","widyr","tidyr")
lapply(load_packages, require, character.only = TRUE)
```

2. Load the `Pride and Prejudice` dataset. The `line_num` attribute is analogous to the line number printed in the book:

```
Pride_Prejudice <- data.frame("text" = prideprejudice,
                              "book" = "Pride and Prejudice",
                              "line_num" =
1:length(prideprejudice),
                              stringsAsFactors=F)
```

3. Now, perform tokenization to restructure the one-string-per-row format to a one-token-per-row format. Here, the token can refer to a single word, a group of characters, co-occurring words (*n*-grams), sentences, paragraphs, and so on. Currently, we will tokenize sentence into singular words:

```
Pride_Prejudice <- Pride_Prejudice %>% unnest_tokens(word,text)
```

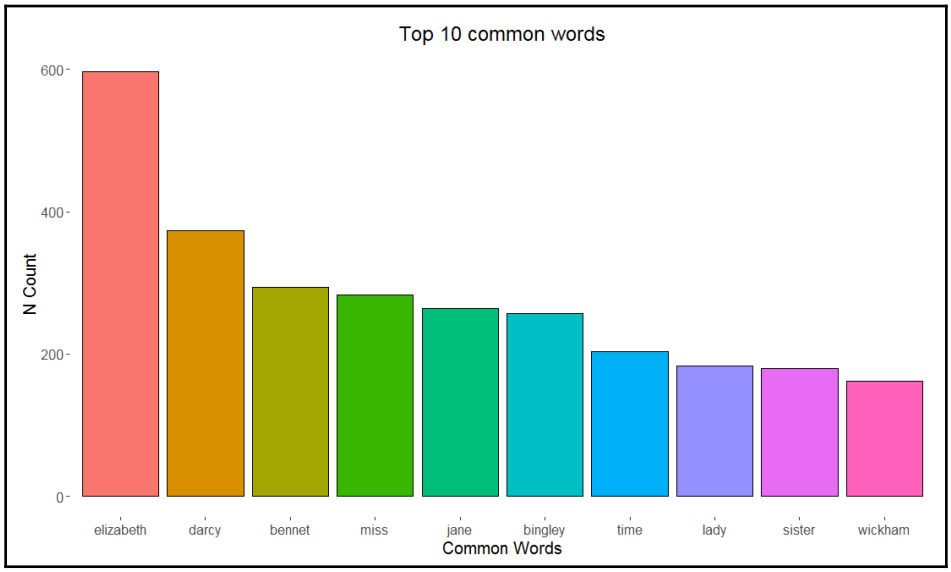
4. Then, remove the commonly occurring words such as *the*, *and*, *for*, and so on using the `stop words` removal corpus:

```
data(stop_words)
Pride_Prejudice <- Pride_Prejudice %>% anti_join(stop_words,
by="word")
```

5. Extract the most common textual words used:

```
most.common <- Pride_Prejudice %>% dplyr::count(word, sort = TRUE)
```

6. Visualize the top 10 common occurring words, as shown in the following figure:



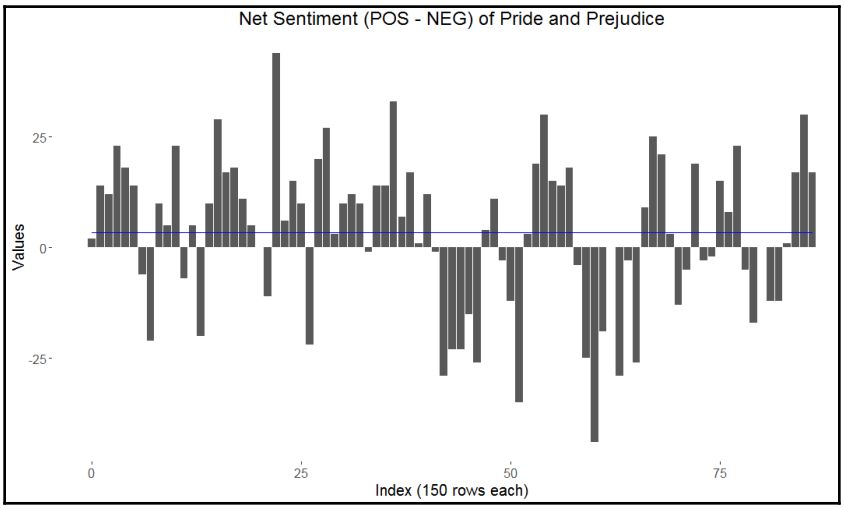
Top 10 common words

```
most.common$word  <- factor(most.common$word , levels =
most.common$word)
ggplot(data=most.common[1:10,], aes(x=word, y=n, fill=word)) +
  geom_bar(colour="black", stat="identity")+
  xlab("Common Words") + ylab("N Count")+
  ggtitle("Top 10 common words")+
  guides(fill=FALSE)+
  theme(plot.title = element_text(hjust = 0.5))+
  theme(text = element_text(size = 10))+
  theme(panel.background = element_blank(), panel.grid.major =
element_blank(),panel.grid.minor = element_blank())
```

7. Then, extract sentiments at a higher level (that is positive or negative) using the `bing` lexicon.

```
Pride_Prejudice_POS_NEG_sentiment <- Pride_Prejudice %>%
inner_join(get_sentiments("bing"), by="word") %>%
dplyr::count(book, index = line_num %/% 150, sentiment) %>%
spread(sentiment, n, fill = 0) %>% mutate(net_sentiment = positive
- negative)
```

8. Visualize the sentiments across small sections (150 words) of text, as shown in the following figure:



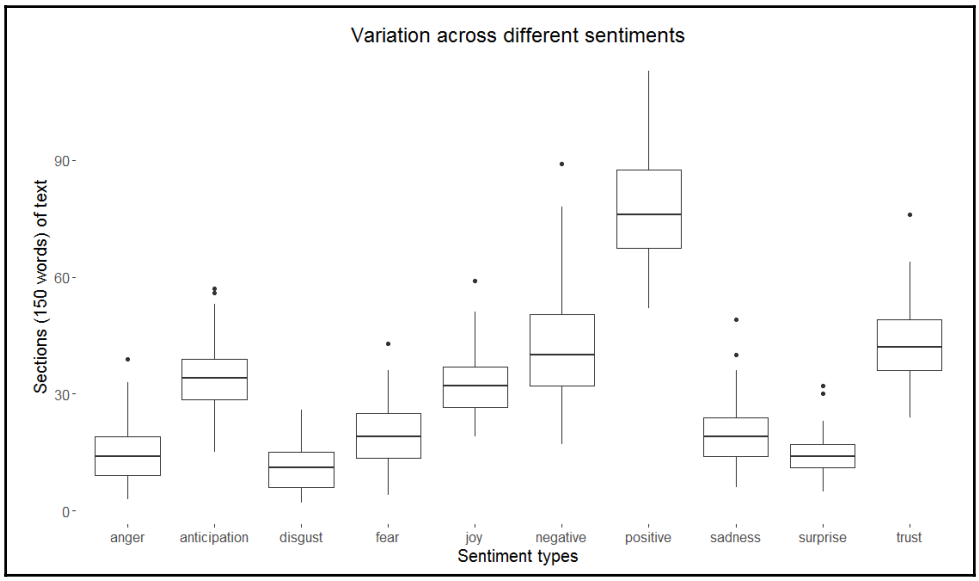
Distribution of the number of positive and negative words across sentences of 150 words each

```
ggplot(Pride_Prejudice_POS_NEG_sentiment, aes(index,
net_sentiment))+
  geom_col(show.legend = FALSE) +
  geom_line(aes(y=mean(net_sentiment)),color="blue")+
  xlab("Section (150 words each)") + ylab("Values")+
  ggtitle("Net Sentiment (POS - NEG) of Pride and Prejudice")+
  theme(plot.title = element_text(hjust = 0.5))+
  theme(text = element_text(size = 10))+
  theme(panel.background = element_blank(), panel.grid.major =
element_blank(),panel.grid.minor = element_blank())
```

9. Now extract sentiments at a granular level (namely positive, negative, anger, disgust, surprise, trust, and so on.) using the `nrc` lexicon:

```
Pride_Prejudice_GRAN_sentiment <- Pride_Prejudice %>%
inner_join(get_sentiments("nrc"), by="word") %>% dplyr::count(book,
index = line_num %/% 150, sentiment) %>% spread(sentiment, n, fill
= 0)
```

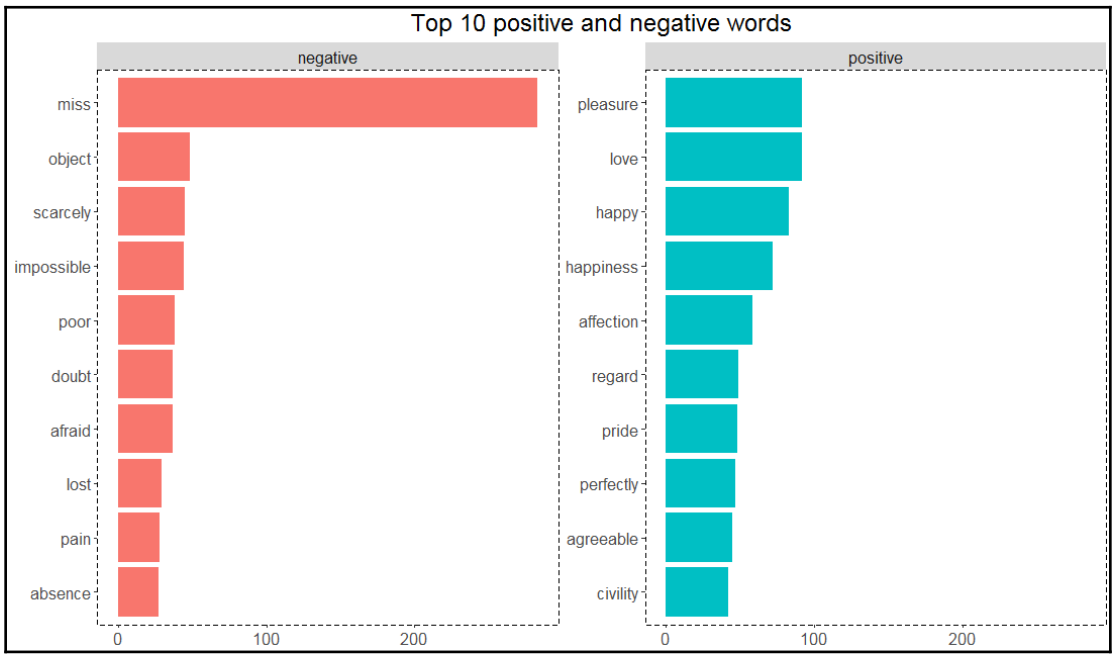
10. Visualize the variation across different sentiments defined, as shown in the following figure:



Variation across different types of sentiments

```
ggplot(stack(Pride_Prejudice_GRAN_sentiment[,3:12]), aes(x = ind, y
= values)) +
  geom_boxplot()+
  xlab("Sentiment types") + ylab("Sections (150 words) of text")+
  ggtitle("Variation across different sentiments")+
  theme(plot.title = element_text(hjust = 0.5))+
  theme(text = element_text(size = 15))+
  theme(panel.background = element_blank(), panel.grid.major =
element_blank(),panel.grid.minor = element_blank())
```

11. Extract the most occurring positive and negative words based on the `bing` lexicon, and visualize them as shown in the following figure:



Top 10 positive and negative words in the novel Pride and Prejudice

```
POS_NEG_word_counts <- Pride_Prejudice %>%
inner_join(get_sentiments("bing"), by="word") %>%
dplyr::count(word, sentiment, sort = TRUE) %>% ungroup()
POS_NEG_word_counts %>% group_by(sentiment) %>% top_n(10) %>%
ungroup() %>% mutate(word = reorder(word, n)) %>% ggplot(aes(word,
n, fill = sentiment)) + geom_col(show.legend = FALSE) +
facet_wrap(~sentiment, scales = "free_y") + ggtitle("Top 10
positive and negative words")+ coord_flip() + theme(plot.title =
element_text(hjust = 0.5))+ theme(text = element_text(size = 15))+
labs(y = NULL, x = NULL)+ theme(panel.background =
element_blank(),panel.border = element_rect(linetype = "dashed",
fill = NA))
```

12. Generate a sentiment word cloud as shown in the following figure:

Word cloud of positive and negative words

```
Prejudice %>%
inner_join(get_sentiments("bing"), by = "word") %>%
dplyr::count(word, sentiment, sort = TRUE) %>% acast(word ~
sentiment, value.var = "n", fill = 0) %>% comparison.cloud(colors =
c("red", "green"), max.words = 100,title.size=2, use.r.layout=TRUE,
random.order=TRUE, scale=c(6,0.5)
```

13. Now analyze sentiments across the chapters of the book:

    1. Extract the chapters, and perform tokenization:

```
austen_books_df <-
as.data.frame(austen_books(),stringsAsFactors=F)
austen_books_df$book <- as.character(austen_books_df$book)
Pride_Prejudice_chapters <- austen_books_df %>%
group_by(book) %>% filter(book == "Pride & Prejudice") %>%
mutate(chapter = cumsum(str_detect(text, regex("^chapter
[\\divxlc]", ignore_case = TRUE)))) %>% ungroup() %>%
unnest_tokens(word, text)
```

    2. Extract the set `positive` and `negative` words from the `bing` lexicon:

```
bingNEG <- get_sentiments("bing") %>%
  filter(sentiment == "negative")
bingPOS <- get_sentiments("bing") %>%
  filter(sentiment == "positive")
```

3. Get the count of words for each chapter:

```
wordcounts <- Pride_Prejudice_chapters %>%
group_by(book, chapter) %>%
dplyr::summarize(words = n())
```

4. Extract the ratio of positive and negative words:

```
POS_NEG_chapter_distribution <- merge (
Pride_Prejudice_chapters %>%
semi_join(bingNEG, by="word") %>%
group_by(book, chapter) %>%
dplyr::summarize(neg_words = n()) %>%
left_join(wordcounts, by = c("book", "chapter")) %>%
mutate(neg_ratio = round(neg_words*100/words,2)) %>%
filter(chapter != 0) %>%
ungroup(),
Pride_Prejudice_chapters %>%
semi_join(bingPOS, by="word") %>%
group_by(book, chapter) %>%
dplyr::summarize(pos_words = n()) %>%
left_join(wordcounts, by = c("book", "chapter")) %>%
mutate(pos_ratio = round(pos_words*100/words,2)) %>%
filter(chapter != 0) %>%
ungroup() )
```

5. Generate a sentiment flag for each chapter based on the proportion of positive and negative words:

```
POS_NEG_chapter_distribution$sentiment_flag <-
ifelse(POS_NEG_chapter_distribution$neg_ratio >
POS_NEG_chapter_distribution$pos_ratio,"NEG","POS")
table(POS_NEG_chapter_distribution$sentiment_flag)
```

How it works...

As mentioned earlier, we have used Jane Austen's famous novel *Pride and Prejudice* in this section, detailing the steps involved in tidying the data, and extracting sentiments using (publicly) available lexicons.

Steps 1 and 2 show the loading of the required `cran` packages and the required text. Steps 3 and 4 perform unigram tokenization and stop word removal. Steps 5 and 6 extract and visualize the top 10 most occurring words across all the 62 chapters. Steps 7 to 12 demonstrate high and granular-level sentiments using two widely used lexicons `bing` and `nrc`.

Both the lexicons contains a list of widely used English words that are tagged to sentiments. In `bing`, each word is tagged to one of the high level binary sentiments (positive or negative), and in `nrc`, each word is tagged to one of the granular-level multiple sentiments (positive, negative, anger, anticipation, joy, fear, disgust, trust, sadness, and surprise).

Each 150-word-long sentence is tagged to a sentiment, and the same has been shown in the figure showing the *Distribution of number of positive and negative words across sentences of 150 words each*. In step 13, chapter-wise sentiment tagging is performed using maximum occurrence of positive or negative words from the `bing` lexicon. Out of 62 chapters, 52 have more occurrences of positive lexicons, and 10 have more occurrences of negative lexicons.

Analyzing documents using tf-idf

In this section, we will learn how to analyze documents quantitatively. A simple way is to look at the distribution of unigram words across the document and their frequency of occurrence, also termed as **term frequency** (**tf**). The words with higher frequency of occurrence generally tend to dominate the document.

However, one would disagree in case of generally occurring words such as the, is, of, and so on. Hence, these are removed by stop word dictionaries. Apart from these stop words, there might be some specific words that are more frequent with less relevance. Such kinds of words are penalized using their **inverse document frequency** (**idf**) values. Here, the words with higher frequency of occurrence are penalized.

The statistic tf-idf combines these two quantities (by multiplication) and provides a measure of importance or relevance of each word for a given document across multiple documents (or a corpus).

In this section, we will generate a tf-idf matrix across chapters of the book *Pride and Prejudice*.

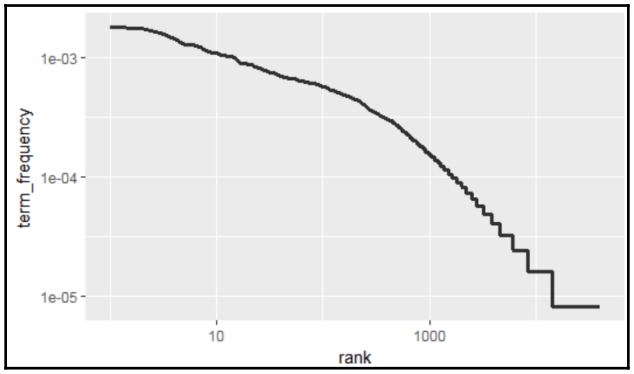
How to do it...

Here is how we go about analyzing documents using tf-idf:

1. Extract the text of all 62 chapters in the book `Pride and Prejudice`. Then, return chapter-wise occurrence of each word. The total words in the book are approx 1.22M.

```
Pride_Prejudice_chapters <- austen_books_df %>%
group_by(book) %>%
filter(book == "Pride & Prejudice") %>%
mutate(linenumber = row_number(),
chapter = cumsum(str_detect(text, regex("^chapter [\\divxlc]",
                                        ignore_case = TRUE)))) %>%
ungroup() %>%
unnest_tokens(word, text) %>%
count(book, chapter, word, sort = TRUE) %>%
ungroup()
```

2. Calculate the rank of words such that the most frequently occurring words have lower ranks. Also, visualize the term frequency by rank, as shown in the following figure:



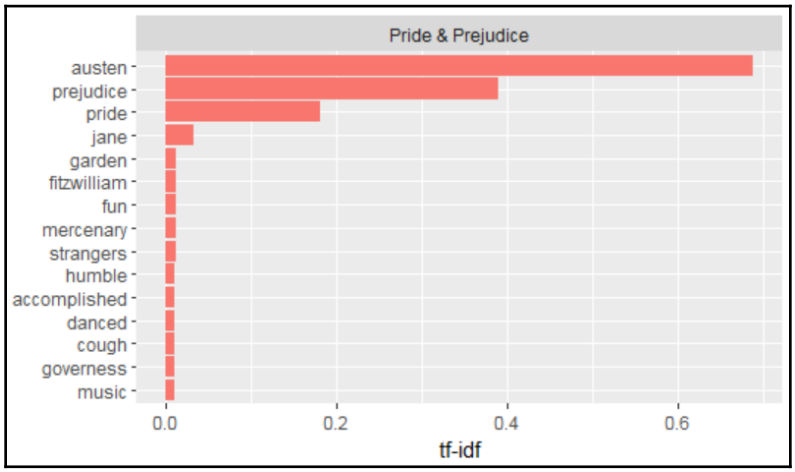
This figure shows lower ranks for words with higher term-frequency (ratio) value

```
freq_vs_rank <- Pride_Prejudice_chapters %>%
mutate(rank = row_number(),
       term_frequency = n/totalwords)
freq_vs_rank %>%
  ggplot(aes(rank, term_frequency)) +
  geom_line(size = 1.1, alpha = 0.8, show.legend = FALSE) +
  scale_x_log10() +
  scale_y_log10()
```

3. Calculate the `tf-idf` value for each word using the `bind_tf-idf` function:

```
Pride_Prejudice_chapters <- Pride_Prejudice_chapters %>%
bind_tf_idf(word, chapter, n)
```

4. Extract and visualize the top 15 words with higher values of tf-idf, as shown in the following figure:



tf-idf values of top 15 words

```
Pride_Prejudice_chapters %>%
  select(-totalwords) %>%
  arrange(desc(tf_idf))

Pride_Prejudice_chapters %>%
  arrange(desc(tf_idf)) %>%
  mutate(word = factor(word, levels = rev(unique(word)))) %>%
  group_by(book) %>%
  top_n(15) %>%
  ungroup %>%
  ggplot(aes(word, tf_idf, fill = book)) +
  geom_col(show.legend = FALSE) +
  labs(x = NULL, y = "tf-idf") +
  facet_wrap(~book, ncol = 2, scales = "free") +
  coord_flip()
```

How it works...

As mentioned earlier, one can observe that the tf-idf scores of very common words such as *the* are close to zero and those of fewer occurrence words such as the proper noun *Austen* is close to one.

Performing sentiment prediction using LSTM network

In this section, we will use LSTM networks to perform sentiment analysis. Along with the word itself, the LSTM network also accounts for the sequence using recurrent connections, which makes it more accurate than a traditional feed-forward neural network.

Here, we shall use the `movie reviews` dataset `text2vec` from the `cran` package. This dataset consists of 5,000 IMDb movie reviews, where each review is tagged with a binary sentiment flag (positive or negative).

How to do it...

Here is how you can proceed with sentiment prediction using LSTM:

1. Load the required packages and movie reviews dataset:

```
load_packages=c("text2vec","tidytext","tensorflow")
lapply(load_packages, require, character.only = TRUE)
data("movie_review")
```

2. Extract the movie reviews and labels as a dataframe and matrix respectively. In movie reviews, add an additional attribute `"Sno"` denoting the review number. In the labels matrix, add an additional attribute related to `negative flag`.

```
reviews <- data.frame("Sno" = 1:nrow(movie_review),
                          "text"=movie_review$review,
                          stringsAsFactors=F)

labels <- as.matrix(data.frame("Positive_flag" =
movie_review$sentiment,"negative_flag" = (1
                     movie_review$sentiment)))
```

3. Extract all the unique words across the reviews, and get their count of occurrences (*n*). Also, tag each word with a unique integer (`orderNo`). Thus, each word is encoded using a unique integer, which shall be later used in the LSTM network.

```
reviews_sortedWords <- reviews %>% unnest_tokens(word,text) %>%
dplyr::count(word, sort = TRUE)
reviews_sortedWords$orderNo <- 1:nrow(reviews_sortedWords)
reviews_sortedWords <- as.data.frame(reviews_sortedWords)
```

4. Now, assign the tagged words back to the reviews based on their occurrences:

```
reviews_words <- reviews %>% unnest_tokens(word,text)
reviews_words <-
plyr::join(reviews_words,reviews_sortedWords,by="word")
```

5. Using the outcome of step 4, create a list of reviews with each review transformed into a set of encoded numbers representing those words:

```
reviews_words_sno <- list()
for(i in 1:length(reviews$text))
{
  reviews_words_sno[[i]] <- c(subset(reviews_words,Sno==i,orderNo))
}
```

6. In order to facilitate equal-length sequences to the LSTM network, let's restrict the review length to 150 words. In other words, reviews longer than 150 words will be truncated to the first 150, whereas shorter reviews will be made 150 words long by prefixing with the required number of zeroes. Thus, we now add in a new word **0**.

```
reviews_words_sno <- lapply(reviews_words_sno,function(x)
{
  x <- x$orderNo
  if(length(x)>150)
  {
    return (x[1:150])
  }
  else
  {
  return(c(rep(0,150-length(x)),x))
  }
})
```

7. Now split the 5,000 reviews into training and testing reviews using a 70:30 split ratio. Also, bind the list of train and test reviews row wise into a matrix format, with rows representing reviews and columns representing the position of a word:

```
train_samples <-
caret::createDataPartition(c(1:length(labels[1,1])),p =
0.7)$Resample1

train_reviews <- reviews_words_sno[train_samples]
test_reviews <- reviews_words_sno[-train_samples]

train_reviews <- do.call(rbind,train_reviews)
test_reviews <- do.call(rbind,test_reviews)
```

8. Similarly, also split the labels into train and test accordingly:

```
train_labels <- as.matrix(labels[train_samples,])
test_labels <- as.matrix(labels[-train_samples,])
```

9. Reset the graph, and start an interactive TensorFlow session:

```
tf$reset_default_graph()
sess<-tf$InteractiveSession()
```

10. Define model parameters such as size of input pixels (`n_input`), step size (`step_size`), number of hidden layers (`n.hidden`), and number of outcome classes (`n.classes`):

```
n_input<-15
step_size<-10
n.hidden<-2
n.class<-2
```

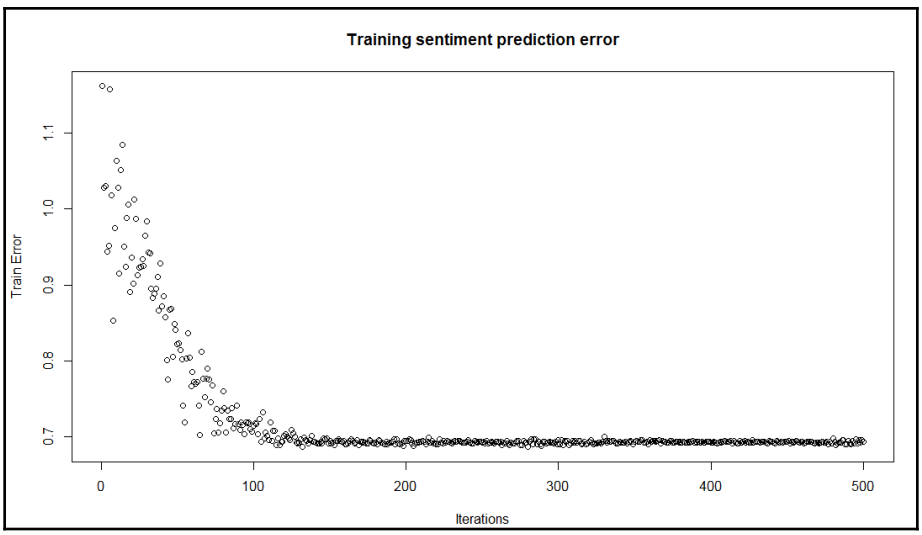
11. Define training parameters such as learning rate (`lr`), number of inputs per batch run (`batch`), and number of iterations (`iteration`):

```
lr<-0.01
batch<-200
iteration = 500
```

12. Based on the RNN and LSTM functions defined in `Chapter 6`, *Recurrent Neural Networks,* from the section *Run the optimization post initializing a session using global variables initializer*.

```
sess$run(tf$global_variables_initializer())
train_error <- c()
for(i in 1:iteration){
  spls <- sample(1:dim(train_reviews)[1],batch)
  sample_data<-train_reviews[spls,]
  sample_y<-train_labels[spls,]
  # Reshape sample into 15 sequence with each of 10 elements
  sample_data=tf$reshape(sample_data, shape(batch, step_size,
n_input))
  out<-optimizer$run(feed_dict = dict(x=sample_data$eval(session =
sess), y=sample_y))
  if (i %% 1 == 0){
    cat("iteration - ", i, "Training Loss - ",  cost$eval(feed_dict
= dict(x=sample_data$eval(), y=sample_y)), "\n")
  }
  train_error <-  c(train_error,cost$eval(feed_dict =
dict(x=sample_data$eval(), y=sample_y)))
}
```

13. Plot the reduction in training errors across iterations as shown in the following figure:



Distribution of sentiment prediction error of training dataset

```
plot(train_error, main="Training sentiment prediction error",
xlab="Iterations", ylab = "Train Error")
```

14. Get the error of test data:

```
test_data=tf$reshape(test_reviews, shape(-1, step_size, n_input))
cost$eval(feed_dict=dict(x= test_data$eval(), y=test_labels))
```

How it works...

In steps 1 to 8, the movie reviews dataset is loaded, processed, and transformed into a set of train and test matrices, which can be directly used to train an LSTM network. Steps 9 to 14 are used to run LSTM using TensorFlow, as described in `Chapter 6`, *Recurrent Neural Networks*. The figure *Distribution of sentiment prediction error of training dataset* shows the decline in training errors across 500 iterations.

Application using text2vec examples

In this section, we will analyze the performance of logistic regression on various examples of `text2vec`.

How to do it...

Here is how we apply `text2vec`:

1. Load the required packages and dataset:

```
library(text2vec)
library(glmnet)
data("movie_review")
```

2. Function to perform Lasso logistic regression, and return the train and test `AUC` values:

```
logistic_model <- function(Xtrain,Ytrain,Xtest,Ytest)
{
  classifier <- cv.glmnet(x=Xtrain, y=Ytrain,
  family="binomial", alpha=1, type.measure = "auc",
  nfolds = 5, maxit = 1000)
  plot(classifier)
  vocab_test_pred <- predict(classifier, Xtest, type = "response")
```

```
  return(cat("Train AUC : ", round(max(classifier$cvm), 4),
  "Test AUC : ",glmnet:::auc(Ytest, vocab_test_pred),"\n"))
}
```

3. Split the movies review data into train and test in an 80:20 ratio:

```
train_samples <-
caret::createDataPartition(c(1:length(labels[1,1])),p =
0.8)$Resample1
train_movie <- movie_review[train_samples,]
test_movie <- movie_review[-train_samples,]
```

4. Generate a DTM of all vocabulary words (without any stop word removal), and asses its performance using Lasso logistic regression:

```
train_tokens <- train_movie$review %>% tolower %>% word_tokenizer
test_tokens <- test_movie$review %>% tolower %>% word_tokenizer

vocab_train <-
create_vocabulary(itoken(train_tokens,ids=train$id,progressbar =
FALSE))

# Create train and test DTMs
vocab_train_dtm <- create_dtm(it =
itoken(train_tokens,ids=train$id,progressbar = FALSE),
                              vectorizer =
vocab_vectorizer(vocab_train))
vocab_test_dtm <- create_dtm(it =
itoken(test_tokens,ids=test$id,progressbar = FALSE),
                              vectorizer =
vocab_vectorizer(vocab_train))

dim(vocab_train_dtm)
dim(vocab_test_dtm)

# Run LASSO (L1 norm) Logistic Regression
logistic_model(Xtrain = vocab_train_dtm,
               Ytrain = train_movie$sentiment,
               Xtest = vocab_test_dtm,
               Ytest = test_movie$sentiment)
```

5. Perform pruning using a list of stop words, and then assess the performance using Lasso logistic regression:

```
data("stop_words")
vocab_train_prune <-
create_vocabulary(itoken(train_tokens,ids=train$id,progressbar =
FALSE),
```

```
                                              stopwords = stop_words$word)

vocab_train_prune <-
prune_vocabulary(vocab_train_prune,term_count_min = 15,
                                          doc_proportion_min = 0.0005,
                                          doc_proportion_max = 0.5)

vocab_train_prune_dtm <- create_dtm(it =
itoken(train_tokens,ids=train$id,progressbar = FALSE),
                                 vectorizer =
vocab_vectorizer(vocab_train_prune))
vocab_test_prune_dtm <- create_dtm(it =
itoken(test_tokens,ids=test$id,progressbar = FALSE),
                                vectorizer =
vocab_vectorizer(vocab_train_prune))

logistic_model(Xtrain = vocab_train_prune_dtm,
               Ytrain = train_movie$sentiment,
               Xtest = vocab_test_prune_dtm,
               Ytest = test_movie$sentiment)
```

6. Generate a DTM using *n*-grams (uni and bigram words), and then assess the performance using Lasso logistic regression:

```
vocab_train_ngrams <-
create_vocabulary(itoken(train_tokens,ids=train$id,progressbar =
FALSE),
                                           ngram = c(1L, 2L))

vocab_train_ngrams <-
prune_vocabulary(vocab_train_ngrams,term_count_min = 10,
                                          doc_proportion_min = 0.0005,
                                          doc_proportion_max = 0.5)

vocab_train_ngrams_dtm <- create_dtm(it =
itoken(train_tokens,ids=train$id,progressbar = FALSE),
                                       vectorizer =
vocab_vectorizer(vocab_train_ngrams))
vocab_test_ngrams_dtm <- create_dtm(it =
itoken(test_tokens,ids=test$id,progressbar = FALSE),
                                     vectorizer =
vocab_vectorizer(vocab_train_ngrams))


logistic_model(Xtrain = vocab_train_ngrams_dtm,
               Ytrain = train_movie$sentiment,
               Xtest = vocab_test_ngrams_dtm,
               Ytest = test_movie$sentiment)
```

7. Perform feature hashing, and then asses the performance using Lasso logistic regression:

```
vocab_train_hashing_dtm <- create_dtm(it =
itoken(train_tokens,ids=train$id,progressbar = FALSE),
                                      vectorizer =
hash_vectorizer(hash_size = 2^14, ngram = c(1L, 2L)))
vocab_test_hashing_dtm <- create_dtm(it =
itoken(test_tokens,ids=test$id,progressbar = FALSE),
                                     vectorizer =
hash_vectorizer(hash_size = 2^14, ngram = c(1L, 2L)))

logistic_model(Xtrain = vocab_train_hashing_dtm,
               Ytrain = train_movie$sentiment,
               Xtest = vocab_test_hashing_dtm,
               Ytest = test_movie$sentiment)
```

8. Using tf-idf transformation on full vocabulary DTM, assess the performance using Lasso logistic regression:

```
vocab_train_tfidf <- fit_transform(vocab_train_dtm, TfIdf$new())
vocab_test_tfidf <- fit_transform(vocab_test_dtm, TfIdf$new())

logistic_model(Xtrain = vocab_train_tfidf,
               Ytrain = train_movie$sentiment,
               Xtest = vocab_test_tfidf,
               Ytest = test_movie$sentiment)
```

How it works...

Steps 1 to 3 loads necessary packages, datasets, and functions required to assess different examples of `text2vec`. Logistic regression is implemented using the `glmnet` package with L1 penalty (Lasso regularization). In step 4, a DTM is created using all the vocabulary words present in the train movie reviews, and the test `auc` value is 0.918. In step 5, the train and test DTMs are pruned using stop words and frequency of occurrence.

The test `auc` value is observed as 0.916, not much decrease compared to using all the vocabulary words. In step 6, along with single words (or uni-grams), bi-grams are also added to the vocabulary. The test `auc` value increases to 0.928. Feature hashing is then performed in step 7, and the test `auc` value is 0.895. Though the `auc` value reduced, hashing is meant to improve run-time performance of larger datasets. Feature hashing is widely popularized by Yahoo. Finally, in step 8, we perform tf-idf transformation, which returns a test `auc` value of 0.907.