Chapter 3: Representing Text – Capturing Semantics

Representing the meaning of words, phrases, and sentences in a form that's understandable to computers is one of the pillars of NLP processing. Machine learning, for example, represents each data point as a fixed-size vector, and we are faced with the question of how to turn words and sentences into vectors. Almost any NLP task starts with representing the text in some numeric form, and this chapter will show several ways of doing that. Once you've learned how to represent text as vectors, you will be able to perform tasks such as classification, which will be described in later chapters.

We will also learn how to turn phrases such as **fried chicken** into vectors, how to train a **`word2vec`** model, and how to create a small search engine with semantic search.

The following recipes will be covered in this chapter:

- Putting documents into a bag of words
- Constructing the N-gram model
- Representing texts with TF-IDF
- Using word embeddings
- Training your own embeddings model
- Representing phrases – phrase2vec
- Using BERT instead of word embeddings
- Getting started with semantic search

Let's get started!

Technical requirements

The code for this chapter is located at **https://github.com/PacktPublishing/Python-Natural-Language-Processing-Cookbook/tree/master/Chapter03**. In this chapter, we will need additional packages. The installation instructions for Anaconda are as follows:

```
pip install sklearn
pip install gensim
pip install pickle
pip install langdetect
conda install pytorch torchvision cudatoolkit=10.2 -c
pytorch
pip install transformers
pip install -U sentence-transformers
pip install whoosh
```

In addition, we will use the models and datasets located at the following URLs:

- **http://vectors.nlpl.eu/repository/20/40.zip**
- **https://www.kaggle.com/currie32/project-gutenbergs-top-20-books**
- **https://www.yelp.com/dataset**
- **https://www.kaggle.com/PromptCloudHQ/imdb-data**

Putting documents into a bag of words

A bag of words is the simplest way of representing text. We treat our text as a collection of documents, where documents are anything from sentences to book chapters to whole books. Since we usually compare different documents to each other or use them in a larger context of other documents, typically, we work with a collection of documents, not just a single document.

The bag of words method uses a training text that provides it with a list of words that it should consider. When encoding new sentences, it counts the number of occurrences each word makes in the document, and the final vector includes those counts for each word in the vocabulary. This representation can then be fed into a machine learning algorithm.

The decision of what represents a document lies with the engineer, and in many cases will be obvious. For example, if you are working on classifying tweets as belonging to a particular topic, a single tweet will be your document. If, on the other hand, you would like to find out which of the chapters of a book are most similar to a book you already have, then chapters are documents.

In this recipe, we will create a bag of words for the beginning of the Sherlock Holmes text. Our documents will be the sentences in the text.

Getting ready

For this recipe, we will be using the **CountVectorizer** class from the **sklearn** package. To install the package, use the following command:

```
pip install sklearn
```

Let's begin.

How to do it...

Our code will take a set of documents – sentences, in this case – and represent them as a matrix of vectors. We will use the file **sherlock_holmes_1.txt** for this task:

1. Import the **CountVectorizer** class and helper functions from ***Chapter 1***, *Learning NLP Basics*:
```
from sklearn.feature_extraction.text import
CountVectorizer
from Chapter01.dividing_into_sentences import
read_text_file,\
preprocess_text, divide_into_sentences_nltk
```
2. Define the **get_sentences** function, which will read in the text file, preprocess the text, and divide it into sentences:
```
def get_sentences(filename):
    sherlock_holmes_text = read_text_file(filename)
    sherlock_holmes_text = \
    preprocess_text(sherlock_holmes_text)
    sentences = \
    divide_into_sentences_nltk(sherlock_holmes_text
)
    return sentences
```
3. Create a function that will return the vectorizer and the final matrix:
```
def create_vectorizer(sentences):
    vectorizer = CountVectorizer()
    X = vectorizer.fit_transform(sentences)
    return (vectorizer, X)
```
4. Now, use the aforementioned functions on the **sherlock_holmes_1.txt** file:

```
sentences = get_sentences("sherlock_holmes_1.txt")
(vectorizer, X) = create_vectorizer(sentences)
```

5. We will now print the matrix representation of the text:

```
print(X)
```

6. The resulting matrix is a **`scipy.sparse.csr.csr_matrix`** object, and the beginning of its printout looks like this:

```
  (0, 114)      1
  (0, 99)       1
  (0, 47)       1
  (0, 98)       1
  (0, 54)       1
  (0, 10)       1
  (0, 0)        1
  (0, 124)      1
   …
```

7. It can also be turned into a **`numpy.matrixlib.defmatrix.matrix`** object, where each sentence is a vector. These sentence vectors can be used our machine learning algorithms later:

```
denseX = X.todense()
```

8. Let's print the resulting matrix:

```
print(denseX)
```

9. Its printout looks like this:

```
[[1 0 0 ... 0 0 0]
[0 0 0 ... 0 0 0]
[0 0 0 ... 0 0 0]
...
[0 0 0 ... 0 0 0]
[0 0 0 ... 0 1 0]
[0 0 0 ... 0 0 1]]
```

10. We can see all the words that were used in the document set:

```
print(vectorizer.get_feature_names())
```

11. The result will be as follows:

```
['_the_', 'abhorrent', 'actions', 'adjusted',
'adler', 'admirable', 'admirably', 'admit', 'akin',
'all', 'always', 'and', 'any', 'as', 'balanced',
'be', 'but', 'cold', 'crack', 'delicate',
'distracting', 'disturbing', 'doubt', 'drawing',
'dubious', 'eclipses', 'emotion', 'emotions',
'excellent', 'eyes', 'factor', 'false', 'felt',
'finely', 'for', 'from', 'gibe', 'grit', 'has',
'have', 'he', 'heard', 'her', 'high', 'him',
```

```
'have', 'he', 'heard', 'her', 'high', 'him',
'himself', 'his', 'holmes', 'in', 'instrument',
'into', 'introduce', 'intrusions', 'irene', 'is',
'it', 'late', 'lenses', 'love', 'lover', 'machine',
'memory', 'men', 'mental', 'mention', 'might',
'mind', 'more', 'most', 'motives', 'name',
'nature', 'never', 'not', 'observer', 'observing',
'of', 'one', 'or', 'other', 'own', 'particularly',
'passions', 'perfect', 'placed', 'position',
'power', 'precise', 'predominates', 'questionable',
'reasoner', 'reasoning', 'results', 'save', 'seen',
'seldom', 'sensitive', 'sex', 'she', 'sherlock',
'sneer', 'softer', 'spoke', 'strong', 'such',
'take', 'temperament', 'than', 'that', 'the',
'there', 'they', 'things', 'throw', 'to',
'trained', 'under', 'upon', 'veil', 'was', 'were',
'which', 'whole', 'with', 'woman', 'world',
'would', 'yet']
```

12. We can now also use the **`CountVectorizer`** object to represent new sentences that were not in the original document set. We will use the sentence *I had seen little of Holmes lately*, which is the next sentence after the excerpt in **`sherlock_holmes_1.txt`**. The **`transform`** function expects a list of documents, so we will create a new list with the sentence as the only element:

```
new_sentence = "I had seen little of Holmes
lately."
new_sentence_vector =
vectorizer.transform([new_sentence])
```

13. We can now print the sparse and dense matrices of this new sentence:

```
print(new_sentence_vector)
print(new_sentence_vector.todense())
```

14. The result will be as follows:

```
  (0, 47)       1
  (0, 76)       1
  (0, 94)       1
[[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0
```

```
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

IMPORTANT NOTE
When running the code from this book's GitHub repository, run it like **`python -m Chapter03.bag_of_words`**. *This will ensure the code you've imported from previous chapters works properly.*

How it works...

In *step 1*, we import the **`CountVectorizer`** class and helper functions. In *step 2*, we define the **`get_sentences`** function, which reads the text of a file into a string, preprocesses it, and divides into sentences. In *step 3*, we define the **`create_vectorizer`** function, which takes in a list of sentences and returns the **`vectorizer`** object and the final matrix representation of the sentences. We will use the **`vectorizer`** object later on to encode new, unseen sentences. In *step 4*, we use the preceding two functions on the **`sherlock_holmes_1.txt`** file. In *step 5*, we print out the matrix that was created from encoding the input text.

The **`CountVectorizer`** object makes a representation of each document by looking at whether each word is in its vocabulary or all the words seen in all the documents are present in a particular document. This can be seen in the sparse representation, where each tuple is a pair, with the document number and word number and the corresponding number, which is the number of times the word appears in that particular document. For example, the sixth entry in the sparse matrix is as follows:

```
  (0, 0)        1
```

This means that there is one word with the number 0 in the document number 0. Document number 0 is the first sentence from our text: **To Sherlock Holmes she is always _the_ woman**. The word number 0 is the first word from the vocabulary listing, which is **_the_**. The ordering of words is alphabetical according to their appearance in the whole text. There is one word, **_the_**, in this sentence, and it is also the sixth word, corresponding to the sixth entry in the sparse matrix.

In *step 6*, we turn the sparse matrix into a dense one, where each sentence is represented with a vector. In *step 7*, we print the resulting matrix out, and we see that it is a nested list, where each individual list is a vector that represents each sentence in the text. In this form, the sentence representations are suitable to use in machine learning algorithms.

In *step 8*, we print out all the words that were used to create the **`vector-`**

In *step 8*, we print out all the words that were used to create the **vectorizer** object. This list is sometimes needed to see which words are in the vocabulary, and which are not.

In *step 9*, we create a string variable with a new sentence that was not used to create the **vectorizer** object, and then apply the transformation to it. In *step 10*, we print out the sparse and dense matrices for this new sentence. It shows that three words from the vocabulary are present, which are **seen**, **of**, and **Holmes**. The other words were not present in the sentences the vectorizer was created on, and so they are absent from the new vectors.

There's more...

The **CountVectorizer** class contains several useful features, such as showing the results of a sentence analysis that shows only the words that are going to be used in the vector representation, excluding words that are very frequent, or excluding stopwords from an included list. We will explore these features here:

1. Import the **CountVectorizer** class and helper functions from ***Chapter 1***, *Learning NLP Basics* :
```
from sklearn.feature_extraction.text import
CountVectorizer
from Chapter01.dividing_into_sentences import
read_text_file, preprocess_text,
divide_into_sentences_nltk
```
2. Read in the text file, preprocess the text, and divide it into sentences:
```
filename="sherlock_holmes_1.txt"
sherlock_holmes_text = read_text_file(filename)
sherlock_holmes_text =
preprocess_text(sherlock_holmes_text)
sentences =
divide_into_sentences_nltk(sherlock_holmes_text)
```
3. Create a new **vectorizer** class. This time, use the **stop_words** argument:
```
vectorizer = CountVectorizer(stop_words='english')
```
4. Use the **vectorizer** object to get the matrix:
```
X = vectorizer.fit_transform(sentences)
```
5. We can print the vocabulary like so:
```
print(vectorizer.get_feature_names())
```
6. The result will be a smaller set with very frequent words, such as **of**,

6. The result will be a similar set with very frequent words, such as **a**, **the**, **to**, and so on (stopwords), missing:

```
['_the_', 'abhorrent', 'actions', 'adjusted',
'adler', 'admirable', 'admirably', 'admit', 'akin',
'balanced', 'cold', 'crack', 'delicate',
'distracting', 'disturbing', 'doubt', 'drawing',
'dubious', 'eclipses', 'emotion', 'emotions',
'excellent', 'eyes', 'factor', 'false', 'felt',
'finely', 'gibe', 'grit', 'heard', 'high',
'holmes', 'instrument', 'introduce', 'intrusions',
'irene', 'late', 'lenses', 'love', 'lover',
'machine', 'memory', 'men', 'mental', 'mention',
'mind', 'motives', 'nature', 'observer',
'observing', 'particularly', 'passions', 'perfect',
'placed', 'position', 'power', 'precise',
'predominates', 'questionable', 'reasoner',
'reasoning', 'results', 'save', 'seen', 'seldom',
'sensitive', 'sex', 'sherlock', 'sneer', 'softer',
'spoke', 'strong', 'temperament', 'things',
'throw', 'trained', 'veil', 'woman', 'world']
```

7. We can now apply the new vectorizer to one of the sentences in the original set and use the **`build_analyzer`** function to see the analysis of the sentence more clearly:

```
new_sentence = "And yet there was but one woman to
him, and that woman was the late Irene Adler, of
dubious and questionable memory."
new_sentence_vector =
vectorizer.transform([new_sentence])
analyze = vectorizer.build_analyzer()
print(analyze(new_sentence))
```

We can see that **and**, **yet**, **there**, **was**, **but**, **one**, **to**, **him**, **that**, **the**, and **of** are all missing from the printout of the sentence analysis:

```
['woman', 'woman', 'late', 'irene', 'adler',
'dubious', 'questionable', 'memory']
```

8. We can print out the sparse vector of the sentence like so:

```
print(new_sentence_vector)
```

9. The result will contain seven words:

```
  (0, 4)        1
  (0, 17)       1
  (0, 35)       1
  (0, 36)       1
  (0, 41)       1
```

```
  (0, 41)       1
  (0, 58)       1
  (0, 77)       2
```

10. Instead of using a prebuilt stopwords list, we can limit the vocabulary by specifying either an absolute or relative maximum document frequency. We can specify this using the **`max_df`** argument, where we either provide it with an integer for the absolute document frequency or a float for the maximum percentage of documents. Our document set is very small, so it won't have an effect, but in a larger set, you would build the **`CountVectorizer`** object with a maximum document frequency, as follows:

```
vectorizer = CountVectorizer(max_df=0.8)
```

In this case, the vectorizer will consider words that appear in less than 80% of all documents.

Constructing the N-gram model

Representing a document as a bag of words is useful, but semantics is about more than just words in isolation. To capture word combinations, an n-gram model is useful. Its vocabulary consists not just of words, but word sequences, or n-grams. We will build a bigram model in this recipe, where bigrams are sequences of two words.

Getting ready

The **`CountVectorizer`** class is very versatile and allows us to construct n-gram models. We will use it again in this recipe. We will also explore how to build character n-gram models using this class.

How to do it...

Follow these steps:

1. Import the **`CountVectorizer`** class and helper functions from ***Chapter 1***, *Learning NLP Basics*, from the *Putting documents into a bag of words* recipe:

```
from sklearn.feature_extraction.text import
CountVectorizer
from Chapter01.dividing_into_sentences import
read_text_file, preprocess_text,
```

```
divide_into_sentences_nltk
from Chapter03.bag_of_words import get_sentences,
get_new_sentence_vector
```

2. Get the sentences from the **sherlock_holmes_1.txt** file:

```
sentences = get_sentences("sherlock_holmes_1.txt")
```

3. Create a new **vectorizer** class. In this case, we will use the **n_gram** argument:

```
bigram_vectorizer = CountVectorizer(ngram_range=(1,
2))
```

4. Use the **vectorizer** object to get the matrix:

```
X = bigram_vectorizer.fit_transform(sentences)
```

5. Print the result:

```
print(X)
```

6. The resulting matrix is a **scipy.sparse.csr.csr_matrix** object, and the beginning of its printout looks like this:

```
  (0, 269)      1
  (0, 229)      1
  (0, 118)      1
  (0, 226)      1
  (0, 136)      1
  (0, 20)       1
  (0, 0)        1
  (0, 299)      1
  (0, 275)      1
  (0, 230)      1
  (0, 119)      1
  (0, 228)      1
   …
```

7. To get a **numpy.matrixlib.defmatrix.matrix** object, where each sentence is a vector, use the **todense()** function:

```
denseX = X.todense()
print(denseX)
```

8. The printout looks like this:

```
[[1 1 0 ... 0 0 0]
[0 0 0 ... 0 0 0]
[0 0 0 ... 0 0 0]
...
[0 0 0 ... 0 0 0]
[0 0 0 ... 1 0 0]
[0 0 0 ... 0 1 1]]
```

9. Let's look at the vocabulary that the model uses:

3. Let's look at the vocabulary that the model uses.

```
print(bigram_vectorizer.get_feature_names())
```

The resulting vocabulary includes each word and each bigram:

```
['_the_', '_the_ woman', 'abhorrent', 'abhorrent
to', 'actions', 'adjusted', 'adjusted temperament',
'adler', 'adler of', 'admirable', 'admirable
things', 'admirably', 'admirably balanced',
'admit', 'admit such', 'akin', 'akin to', 'all',
'all emotions', 'all his', 'always', 'always
_the_', 'and', 'and actions', 'and finely', 'and
observing', 'and predominates', 'and questionable',
'and sneer', 'and that', 'and yet', 'any', 'any
emotion', 'any other', 'as', 'as his', 'as lover',
'balanced', 'balanced mind', 'be', 'be more',
'but', 'but admirably', 'but as', 'but for', 'but
one', 'cold', 'cold precise', 'crack', 'crack in',
'delicate', 'delicate and', 'distracting',
'distracting factor', 'disturbing', 'disturbing
than', 'doubt', 'doubt upon', 'drawing', 'drawing
the', 'dubious', 'dubious and', 'eclipses',
'eclipses and', 'emotion', 'emotion akin', 'emotion
in', 'emotions', 'emotions and', 'excellent',
'excellent for', 'eyes', 'eyes she', 'factor',
'factor which', 'false', 'false position', 'felt',
'felt any', 'finely', 'finely adjusted', 'for',
'for drawing', 'for irene', 'for the', 'from',
'from men', 'gibe', 'gibe and', 'grit', 'grit in',
'has', 'has seen', 'have', 'have placed', 'have
seldom', 'he', 'he felt', 'he never', 'he was', 'he
would', 'heard', 'heard him', 'her', 'her sex',
'her under', 'high', 'high power', 'him', 'him
and', 'him mention', 'himself', 'himself in',
'his', 'his cold', 'his eyes', 'his mental', 'his
own', 'holmes', 'holmes she', 'in', 'in false', 'in
his', 'in nature', 'in one', 'in sensitive',
'instrument', 'instrument or', 'into', 'into his',
'introduce', 'introduce distracting', 'intrusions',
'intrusions into', 'irene', 'irene adler', 'is',
'is always', 'it', 'it the', 'it was', 'late',
'late irene', 'lenses', 'lenses would', 'love',
'love for', 'lover', 'lover he', 'machine',
'machine that', 'memory', 'men', 'men motives',
```

```
'machine that', 'memory', 'men', 'men motives',
'mental', 'mental results', 'mention', 'mention
her', 'might', 'might throw', 'mind', 'more', 'more
disturbing', …]
```

10. We can now also use the **`CountVectorizer`** object to represent new sentences that were not in the original document set. We will use the sentence *I had seen little of Holmes lately*, which is the next sentence after the excerpt in **`sherlock_holmes_1.txt`**. The **`transform`** function expects a list of documents, so we will create a new list where the sentence is the only element:

```
new_sentence = "I had seen little of Holmes
lately."
new_sentence_vector = \
bigram_vectorizer.transform([new_sentence])
```

11. We can now print the sparse and dense representations of this new sentence:

```
print(new_sentence_vector)
print(new_sentence_vector.todense())
```

The result will be as follows:

```
  (0, 118)      1
  (0, 179)      1
  (0, 219)      1
[[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
  0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

12. Let's compare the preceding representation with the representation of a sentence from the original input text; that is, *And yet there was but*

one woman to him, and that woman was the late Irene Adler, of dubious and questionable memory:

```
new_sentence1 = " And yet there was but one woman
to him, and that woman was the late Irene Adler, of
dubious and questionable memory."
new_sentence_vector1 =
vectorizer.transform([new_sentence])
```

13. We will print the sparse and dense representations of this sentence:

```
print(new_sentence_vector1)
print(new_sentence_vector1.todense())
```

14. The result will be as follows:

```
  (0, 7)        1
  (0, 8)        1
  (0, 22)       3
  (0, 27)       1
  (0, 29)       1
  …
[[0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0
0 0 1 0 1 1 0 0 0 0 0
  0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
  1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 1 1 0
  0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1
  1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 1
  1 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 1 0 0 0 0
0 0 0 0 0 0 0 2 1 0 0
  1 0 0 0 0 0 0 0 0 0 0 2 1 1 0 0 0 0 0 1 1]]
```

How it works…

In *step 1*, we import the necessary objects and functions, and in *step 2*, we create a list of sentences from the **`sherlock_holmes_1.txt`** file. In *step 3*, we create a new **`CountVectorizer`** object that has an extra argument, **`ngram_range`**. The **`CountVectorizer`** class, when the **`ngram_range`** ar-

gument is set, counts not only individual words, but also word combinations, where the number of words in the combinations depends on the numbers provided to the **`ngram_range`** argument. We provided **`ngram_range=(1,2)`** as the argument, which means that the number of words in the combinations ranges from 1 to 2, so unigrams and bigrams are counted.

In *step 4*, we use the **`bigram_vectorizer`** object and create the matrix, and in *step 5*, we print the matrix. The result looks very similar to the matrix output shown in the *Putting documents into a bag of words* recipe, with the only difference that the output should now be longer as it includes not just individual words, but also bigrams, or sequences of two words.

In *step 6*, we create a dense matrix and print it out. In *step 7*, we print out the vocabulary of the vectorizer and we see that it includes both individual words and bigrams.

In *step 8*, we transform an unseen sentence using the vectorizer and in *step 9*, we print out both the sparse and dense vectors for the sentence. In *step 10*, we transform a sentence that was part of the original text and in *step 11*, we print out its sparse and dense representations. Once we analyze the new sentence, we see that only three words (or bigrams) from the vectorizer's original vocabulary are present in the sentence, while the sentence that was part of the original dataset has a lot more words counted. This shows how sentences that are very different from the original sentence set the vectorizer was fitted on will be poorly represented, since most of the words and word combinations will be missing from the vectorizer's vocabulary.

There's more...

We can use trigrams, quadrigrams, and more in the vectorizer by providing the corresponding tuple to the **`ngram_range`** argument. The downside of this is the ever-expanding vocabulary and the growth of sentence vectors, since each sentence vector has to have an entry for each word in the input vocabulary.

Representing texts with TF-IDF

We can go one step further and use the TF-IDF algorithm to count words

We can go one step further and use the TF-IDF algorithm to count words and ngrams in incoming documents. **TF-IDF** stands for **term frequency-inverse document frequency** and gives more weight to words that are unique to a document than to words that are frequent, but repeated throughout most documents. This allows us to give more weight to words uniquely characteristic to particular documents. You can find out more at **https://scikit-learn.org/stable/modules/feature_extraction.html#tfidf-term-weighting**.

In this recipe, we will use a different type of vectorizer that can apply the TF-IDF algorithm to the input text. Like the **`CountVectorizer`** class, it has an analyzer that we will use to show the representations of new sentences.

Getting ready

We will be using the **`TfidfVectorizer`** class from the **`sklearn`** package. We will also be using the stopwords list from ***Chapter 1***, *Learning NLP Basics*.

How to do it...

The **`TfidfVectorizer`** class allows for all the functionality of **`CountVectorizer`**, except that it uses the TF-IDF algorithm to count the words instead of direct counts. The other features of the class should be familiar. We will again be using the **`sherlock_holmes_1.txt`** file.

Here are the steps you should follow to build and use the TF-IDF vectorizer:

1. Import the **`TfidfVectorizer`** class, **`nltk`**, and the relevant helper functions:
```
import nltk
import string
from sklearn.feature_extraction.text import
TfidfVectorizer
from nltk.stem.snowball import SnowballStemmer
from Chapter01.removing_stopwords import
read_in_csv
from Chapter03.bag_of_words import get_sentences
```
2. Define the stemmer and the stopwords file path:
```
stemmer = SnowballStemmer('english')
```

```
stopwords_file_path = "Chapter01/stopwords.csv"
```

3. Get the sentences from the **sherlock_holmes_1.txt** file:

```
sentences = get_sentences("sherlock_holmes_1.txt")
```

4. We will use a function to tokenize and stem every word, including stopwords. See the *Dividing sentences into words* and *Word stemming* recipes in ***Chapter 1***, *Learning NLP Basics*, for more information:

```
def tokenize_and_stem(sentence):
    tokens = nltk.word_tokenize(sentence)
    filtered_tokens = [t for t in tokens if t not
in \
                       string.punctuation]
    stems = [stemmer.stem(t) for t in
filtered_tokens]
    return stems
```

5. Read in, tokenize, and stem the stopwords:

```
stopword_list = read_in_csv(stopwords_file_path)
stemmed_stopwords = [tokenize_and_stem(stopword)[0]
for \
                     stopword in stopword_list]
stopword_list = stopword_list + stemmed_stopwords
```

6. Create a new vectorizer class and fit the incoming sentences. You might see some warnings here, which is fine:

```
tfidf_vectorizer = \
TfidfVectorizer(max_df=0.90, max_features=200000,
                min_df=0.05,
stop_words=stopword_list,
                use_idf=True,tokenizer=tokenize_and
_stem,
                ngram_range=(1,3))
tfidf_vectorizer = tfidf_vectorizer.fit(sentences)
```

7. Use the **vectorizer** object to get the matrix:

```
tfidf_matrix =
tfidf_vectorizer.transform(sentences)
```

8. Print the result:

```
print(tfidf_matrix)
```

9. The resulting matrix is a **scipy.sparse.csr.csr_matrix** object, and the beginning of its printout looks like this:

```
  (0, 195)      0.2892833606818738
  (0, 167)      0.33843668854613723
  (0, 166)      0.33843668854613723
  (0, 165)      0.33843668854613723
```

```
  (0, 84)       0.33843668854613723
  (0, 83)       0.33843668854613723
  (0, 82)       0.33843668854613723
  (0, 1)        0.33843668854613723
  (0, 0)        0.33843668854613723
   …
```

10. To get a **`numpy.matrixlib.defmatrix.matrix`** object, where each sentence is a vector, use the **`todense()`** function:

    ```
    dense_matrix = tfidf_matrix.todense()
    ```

    Its printout looks like this:

    ```
    [[0.33843669 0.33843669 0.         ...
    0.         0.         0.        ]
    [0.         0.         0.         ...
    0.         0.         0.        ]
    [0.         0.         0.         ...
    0.         0.         0.        ]
    ...
    [0.         0.         0.         ...
    0.         0.         0.        ]
    [0.         0.         0.         ...
    0.         0.         0.        ]
    [0.         0.         0.         ...
    0.         0.         0.        ]] [0 0 0 ... 1 0
    0]
    [0 0 0 ... 0 1 1]]
    ```

11. Let's look at the vocabulary that the model uses:

    ```
    print(tfidf_vectorizer.get_feature_names())
    ```

    The resulting vocabulary includes each stemmed word, each bigram, and each trigram:

    ```
    [['_the_', '_the_ woman', 'abhorr', 'abhorr cold',
    'abhorr cold precis', 'action', 'adjust', 'adjust
    tempera', 'adjust tempera introduc', 'adler',
    'adler dubious', 'adler dubious question', 'admir',
    'admir balanc', 'admir balanc mind', 'admir
    observer—excel', 'admir observer—excel draw',
    'admit', 'admit intrus', 'admit intrus own',
    'akin', 'akin love', 'akin love iren', 'balanc',
    'balanc mind', 'cold', 'cold precis', 'cold precis
    admir', 'crack', 'crack own', 'crack own high-pow',
    'delic', 'delic fine', 'delic fine adjust',
    ```

```
'distract', 'distract factor', 'distract factor
throw', 'disturb', 'disturb strong', 'disturb
strong emot', 'doubt', 'doubt mental', 'doubt
mental result', 'draw', 'draw veil', 'draw veil
men', 'dubious', 'dubious question', 'dubious
question memori', 'eclips', 'eclips predomin',
'eclips predomin whole', 'emot', 'emot abhorr',
'emot abhorr cold', 'emot akin', 'emot akin love',
'emot natur', 'eye', 'eye eclips', 'eye eclips
predomin', 'factor', 'factor throw', 'factor throw
doubt', 'fals', 'fals posit', 'felt', 'felt emot',
'felt emot akin', 'fine', 'fine adjust', 'fine
adjust tempera', 'gibe', 'gibe sneer', 'grit',
'grit sensit', 'grit sensit instrument', 'heard',
'heard mention', 'high-pow', 'high-pow lens',
'high-pow lens disturb', 'holm', 'holm _the_',
'holm _the_ woman', 'instrument', 'instrument
crack', 'instrument crack own', 'introduc',
'introduc distract', 'introduc distract factor',
'intrus', 'intrus own', 'intrus own delic', 'iren',
'iren adler', 'iren adler dubious', 'lens', 'lens
disturb', 'lens disturb strong', …]
```

12. Let's build an analyzer function and analyze the sentence *To Sherlock Holmes she is always _the_ woman*:

```
analyze = tfidf_vectorizer.build_analyzer()
print(analyze("To Sherlock Holmes she is always
_the_ woman."))
```

This is the result:

```
['sherlock', 'holm', '_the_', 'woman', 'sherlock
holm', 'holm _the_', '_the_ woman', 'sherlock holm
_the_', 'holm _the_ woman']
```

How it works…

The **`TfidfVectorizer`** class works almost exactly like the **`CountVectorizer`** class, differing only in the way the word frequencies are calculated, so most of the steps should be familiar here. Word frequencies are calculated as follows. For each word, the overall frequency is a product of the term frequency and the inverse document frequency. Term frequency is the number of times the word occurs in the document. Inverse document frequency is the total number of documents divided by

the number of documents where the word occurs. Usually, these frequencies are logarithmically scaled.

In *step 1*, we import the **`TfidfVectorizer`** and **`SnowballStemmer`** classes and helper functions. In *step 2*, we define the stemmer object and the path to the stopwords file. In *step 3*, we create the list of sentences from the **`sherlock_holmes_1.txt`** file.

In *step 4*, we define the **`tokenize_and_stem`** function, which we will use to tokenize and stem the words in a sentence.

In *step 5*, we read in the stopwords list and apply the **`tokenize_and_stem`** function to it. Since we will be stemming the words in the text, we also need to stem the stopwords. The reason we need to do this is because if we leave the stopwords unstemmed, the stemmed words in the dataset will not match them. The functions also exclude all punctuation from text since we do not want to include n-grams with punctuation in them. To do that, we check if each token is included in the **`string.punctuation`** set, which lists all punctuation symbols. In *step 6*, we create a new vectorizer class and then fit the sentences. The **`min_df`** and **`max_df`** arguments limit the minimum and maximum document frequency, respectively. If the corpus is large enough, the maximum document frequency can take care of the stopwords by excluding words that are very frequent across the documents. For our small corpus, I had to provide the stopword list. The **`min_df`** and **`max_df`** arguments are either floats between 0 and 1, representing a proportion of documents, or an integer, representing an absolute count. The **`max_features`** argument caps the number of words and n-grams in the vocabulary at the number provided. For more information about **`TfidfVectorizer`**, see **https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html**.

In *step 7*, we use the vectorizer to get the matrix encoding the sentences and in *step 8*, we print the result. We can see that the resulting matrix is very similar to the matrices in the *Putting documents into a bag of words* and *Constructing the N-gram model* recipes of this chapter. The difference is that the frequency count is a fraction, since it is a product of two ratios.

In *step 9*, we create a dense matrix that represents each sentence as a vector.

In *step 10*, we print out the vocabulary of the vectorizer, which includes

In *step 10*, we print out the vocabulary of the vectorizer, which includes unigrams as well as bigrams.

In *step 11*, we create the analyzer object and analyze a sentence from the **sherlock_holmes_1.txt** file. The result shows that now, the sentence is represented with uni-, bi- and trigrams, along with words that have been stemmed and whose stopwords have been removed.

There's more...

We can build **TfidfVectorizer** and use character n-grams instead of word n-grams. Character n-grams use the character, and not the word, as their basic unit. For example, if we were to build character n-grams for the phrase *the woman* with the n-gram range (1, 3), its set would be **[t, h, e, w, o, m, a, n, th, he, wo, om, ma, an, the, wom, oma, man]**. In many experimental settings, models based on character n-grams perform better than word-based n-gram models.

We will use the same Sherlock Holmes text file, **sherlock_holmes_1.txt**, and the same class, **TfidfVectorizer**. We will not need a tokenizer function or a stopwords list, since the unit of analysis is the character and not the word. The steps to create the vectorizer and analyze a sentence are as follows:

1. Get the sentences from the **sherlock_holmes_1.txt** file:
```
sentences = get_sentences("sherlock_holmes_1.txt")
```
2. Create a new **vectorizer** class and fit the incoming sentences:
```
tfidf_char_vectorizer = \
TfidfVectorizer(analyzer='char_wb',
                max_df=0.90,
                max_features=200000,
                min_df=0.05,
                use_idf=True,
                ngram_range=(1,3))
tfidf_char_vectorizer =
tfidf_char_vectorizer.fit(sentences)
```
3. Use the **vectorizer** object to get the matrix:
```
tfidf_matrix =
tfidf_char_vectorizer.transform(sentences)
```
4. Print the result:
```
print(tfidf_matrix)
```
Predictably, the resulting matrix is much larger than the ones based

on words. The beginning of its printout looks like this:

```
  (0, 763)      0.12662434631923655
  (0, 762)      0.12662434631923655
  (0, 753)      0.05840470946313
  (0, 745)      0.10823388151187574
  (0, 744)      0.0850646359499111
  (0, 733)      0.12662434631923655
  (0, 731)      0.07679517427049085
  (0, 684)      0.07679517427049085
  (0, 683)      0.07679517427049085
  (0, 675)      0.05840470946313
  (0, 639)      0.21646776302375148
  (0, 638)      0.21646776302375148
   …
```

5. To get a **`numpy.matrixlib.defmatrix.matrix`** object, where each sentence is a vector, use the **`todense()`** function:

   ```
   dense_matrix = tfidf_matrix.todense()
   ```

   Its printout looks like this:

   ```
   [[0.12662435 0.12662435 0.         ...
   0.         0.         0.        ]
   [0.         0.         0.         ...
   0.         0.         0.        ]
   [0.         0.         0.         ...
   0.         0.         0.        ]
   ...
   [0.         0.         0.07119069 ...
   0.         0.         0.        ]
   [0.         0.         0.17252729 ...
   0.         0.         0.        ]
   [0.         0.         0.         ...
   0.         0.         0.        ]]
   ```

6. Let's look at the vocabulary that the model uses:

   ```
   print(tfidf_char_vectorizer.get_feature_names())
   ```

   The resulting vocabulary includes each character, as well as the character bigrams and trigrams. The beginning of the printout looks like this:

   ```
   [' _', ' _t', ' a ', ' ab', ' ac', ' ad', ' ak', '
   al', ' an', ' as', ' b', ' ba', ' be', ' bu', ' c',
   ' co', ' cr', ' d', ' de', ' di', ' do', ' dr', '
   du', ' e', ' ec', ' em', ' ey', ' f', ' fa', ' fe',
   ' fi', ' fo', ' fr', ' g', ' gi', ' gr', ' ha', '
   ```

```
 'fi', 'fo', 'fr', 'g', 'gi', 'gr', 'ha',
he', ' hi', ' ho', ' i', ' i ', ' in', ' ir', '
is', ' it', ' l', ' la', ' le', ' lo', ' m', ' ma',
' me', ' mi', ' mo', ' n', ' na', ' ne', ' no', '
o', ' ob', ' of', ' on', ' or', ' ot', ' ow', ' p',
' pa', ' pe', ' pl', ' po', ' pr', ' q', ' qu', '
r', ' re', ' s', ' sa', ' se', ' sh', ' sn', ' so',
' sp', ' st', ' su', ' ta', ' te', ' th', ' to', '
tr', ' u', ' un', ' up', ' v', ' ve', ' wa', ' we',
' wh', ' wi', ' wo', ' y', ' ye', ',', ', ', '-',
'-p', '-po', '_', '_ ', '_t', '_th', 'a ', 'ab',
'abh', 'abl', 'ac', 'ace', 'ach', 'ack', 'act',
'ad', 'adj', 'adl', 'adm', 'ai', 'ain', 'ak',
'ake', 'aki', 'al', 'al ', 'ala', 'all', 'als',
'alw', 'am', 'ame', 'an ', 'an.', 'anc', 'and',
'any', 'ar', 'ard', 'arl', 'art', 'as', 'as ',
'as,', 'aso', 'ass', 'at', 'at ', 'ate', 'atu',
'av', 'ave', 'aw', 'awi', 'ay', 'ays', 'b', 'ba',
'bal', 'be', 'be ', 'bh', 'bho', 'bi', 'bin',
'bio', 'bl', 'ble', 'bly', 'bs', 'bse', 'bt', 'bt
', 'bu', 'but', 'c', 'ca', 'cat', 'ce', 'ce ',
'ced', 'cel', 'ch', 'ch ', 'chi', 'ci', 'cis',
'ck', 'ck ', 'cl', 'cli', 'co', 'col', 'cr', 'cra',
'ct', 'ct ', 'cti', 'cto', 'cu', 'cul', …]
```

7. Let's build an analyzer function and analyze the sentence *To Sherlock Holmes she is always _the_ woman*:

```
analyze = tfidf_char_vectorizer.build_analyzer()
print(analyze("To Sherlock Holmes she is always
_the_ woman."))
```

This is the result:

```
[' ', 't', 'o', ' ', ' t', 'to', 'o ', ' to', 'to
', ' ', 's', 'h', 'e', 'r', 'l', 'o', 'c', 'k', '
', ' s', 'sh', 'he', 'er', 'rl', 'lo', 'oc', 'ck',
'k ', ' sh', 'she', 'her', 'erl', 'rlo', 'loc',
'ock', 'ck ', ' ', 'h', 'o', 'l', 'm', 'e', 's', '
', ' h', 'ho', 'ol', 'lm', 'me', 'es', 's ', ' ho',
'hol', 'olm', 'lme', 'mes', 'es ', ' ', 's', 'h',
'e', ' ', ' s', 'sh', 'he', 'e ', ' sh', 'she', 'he
', ' ', 'i', 's', ' ', ' i', 'is', 's ', ' is', 'is
', ' ', 'a', 'l', 'w', 'a', 'y', 's', ' ', ' a',
'al', 'lw', 'wa', 'ay', 'ys', 's ', ' al', 'alw',
```

```
 'lwa', 'way', 'ays', 'ys ', ' ', '_', 't', 'h',
 'e', '_', ' ', ' _', '_t', 'th', 'he', 'e_', '_ ',
 ' _t', '_th', 'the', 'he_', 'e_ ', ' ', 'w', 'o',
 'm', 'a', 'n', '.', ' ', ' w', 'wo', 'om', 'ma',
 'an', 'n.', '. ', ' wo', 'wom', 'oma', 'man',
 'an.', 'n. ']
```

Using word embeddings

In this recipe we switch gears and learn how to represent *words* using word embeddings, which are powerful because they are a result of training a neural network that predicts a word from all other words in the sentence. The resulting vector embeddings are similar for words that occur in similar contexts. We will use the embeddings to show these similarities.

Getting ready

In this recipe, we will use a pretrained word2vec model, which can be found at **http://vectors.nlpl.eu/repository/20/40.zip**. Download the model and unzip it in the **`Chapter03`** directory. You should now have a file whose path is **`…/Chapter03/40/model.bin`**.

We will also be using the **`gensim`** package to load and use the model. Install it using **`pip`**:

```
pip install gensim
```

How to do it…

We will load the model, demonstrate some features of the **`gensim`** package, and then compute a sentence vector using the word embeddings. Let's get started:

1. Import the **`KeyedVectors`** object from **`gensim.models`** and **`numpy`**:
```
from gensim.models import KeyedVectors
import numpy as np
```
2. Assign the model path to a variable:
```
w2vec_model_path = "Chapter03/40/model.bin"
```
3. Load the pretrained model:
```
model =
KeyedVectors.load_word2vec_format(w2vec_model_path,
```

```
                                                    binary=Tr
ue)
```

4. Using the pretrained model, we can now load individual word vectors:

```
print(model['holmes'])
```

5. The result will be as follows:

```
[-0.309647 -0.127936 -0.136244 -0.252969  0.410695
  0.206325  0.119236
-0.244745 -0.436801  0.058889  0.237439  0.247656
  0.072103  0.044183
-0.424878  0.367344  0.153287  0.343856  0.232269
-0.181432 -0.050021
  0.225756  0.71465  -0.564166 -0.168468 -0.153668
  0.300445 -0.220122
-0.021261  0.25779  …]
```

6. We can also get words that are most similar to a given word. For example, let's print out the words most similar to *Holmes* (lowercase, since all the words are lowercased in the training process):

```
print(model.most_similar(['holmes'], topn=15))
```

The result is as follows:

```
[('sherlock', 0.8416914939880371), ('parker',
0.8099909424781799), ('moriarty',
0.8039606809616089), ('sawyer',
0.8002701997756958), ('moore', 0.7932805418968201),
('wolfe', 0.7923581600189209), ('hale',
0.791009247303009), ('doyle', 0.7906038761138916),
('holmes.the', 0.7895270586013794), ('watson',
0.7887690663337708), ('yates', 0.7882785797119141),
('stevenson', 0.7879440188407898), ('spencer',
0.7877693772315979), ('goodwin',
0.7866846919059753), ('baxter',
0.7864187955856323)]
```

7. We can now also compute a sentence vector by averaging all the word vectors in the sentence. We will use the sentence I*t was not that he felt any emotion akin to love for Irene Adler*:

```
sentence = "It was not that he felt any emotion
akin to love for Irene Adler."
```

8. Let's define a function that will take a sentence and a model and will return a list of the sentence word vectors:

```
def get_word_vectors(sentence, model):
    word_vectors = []
    for word in sentence:
```

```
    for word in sentence:
        try:
            word_vector =
model.get_vector(word.lower())
            word_vectors.append(word_vector)
        except KeyError:
            continue
    return word_vectors
```

9. Now, let's define a function that will take the word vector list and compute the sentence vector:

```
def get_sentence_vector(word_vectors):
    matrix = np.array(word_vectors)
    centroid = np.mean(matrix[:,:], axis=0)
    return centroid
```

IMPORTANT NOTE

Averaging the word vectors to get the sentence vector is only one way of approaching this task, and is not without its problems. One alternative is to train a doc2vec model, where sentences, paragraphs, and whole documents can all be units instead of words.

10. We can now compute the sentence vector:

```
word_vectors = get_word_vectors(sentence, model)
sentence_vector = get_sentence_vector(word_vectors)
print(sentence_vector)
```

11. The result is as follows:

```
[ 0.09226871  0.14478634  0.23788658
-0.31754282  0.42911175 -0.05198449
  0.12572111  0.01170996
-0.01138579  0.05200932  0.15247145  0.34026343
  0.12961692  0.05010585
-0.09165132  0.3782767   0.08390289  0.30078036
-0.24396846  0.42507184
-0.13556597  0.157348    0.19739327 -0.13114193
-0.16301586  0.19061208 …]
```

How it works…

In *step 1*, we import the necessary objects. In *step 2*, we assign the path of the model we downloaded in the *Getting ready* section to the **`w2vec_model_path`** variable. In *step 3*, we load the model.

In *step 4*, we load the word vector for the word *Holmes*. We have to lowercase it since all the words in the model are in lowercase. The result is a

case it since all the words in the model are in lowercase. The result is a long vector that represents this word in the word2vec model.

In *step 5*, we get 15 words that are most similar to the input word. The output prints out the words that are the most similar (occur in similar contexts), as well as their similarity scores. The score is the cosine distance between a pair of vectors, in this case representing a pair of words. The larger the score, the more similar the two words. In this case, the result is pretty good, as it contains the words *Sherlock*, *Moriarty*, *Watson*, and *Doyle*.

In the next few steps, we compute a sentence vector by averaging the word vectors. This is one approach to representing sentences using word2vec, and it has its disadvantages. One of the challenges of this approach is representing words that are not present in the model.

In *step 6*, we initialize the **`sentence`** variable with a sentence from the text. In *step 7*, we create the **`get_word_vectors`** function, which returns a list of all word vectors in the sentence. The function reads the word vector from the model and appends it to the **`word_vectors`** list. It also catches the **`KeyError`** error that is raised if the word is not present in the model.

In *step 8*, we create the **`get_sentence_vector`** function, which takes in a list of words vectors and returns their average. In order to compute the average, we represent the matrix as a NumPy array and use NumPy's **`mean`** function to get the average vector.

In *step 9*, we compute the word vectors for the sentence we defined in *step 6*, and then the sentence vector using the functions we just defined in *steps 7* and *8*. We then print the resulting sentence vector.

There's more...

There are some other fun things **`gensim`** can do with a pretrained model. For example, it can find a word that doesn't match from a list of words and find a word that is most similar to the given word from a list. Let's look at these:

1. Import the **`KeyedVectors`** object from **`gensim.models`**:
```
from gensim.models import KeyedVectors
```
2. Assign the model path to a variable:

```
w2vec_model_path = "Chapter03/40/model.bin"
```

3. Load the pretrained model:

```
model = \
KeyedVectors.load_word2vec_format(w2vec_model_path,
                                  binary=True)
```

4. Compile a list of words with one that doesn't match:

```
words = ['banana', 'apple', 'computer',
'strawberry']
```

5. Apply the **`doesnt_match`** function to the list and print the result:

```
print(model.doesnt_match(words))
```

The result will be as follows:

```
computer
```

6. Now, let's find a word that's most similar to another word:

```
word = "cup"
words = ['glass', 'computer', 'pencil', 'watch']
print(model.most_similar_to_given(word, words))
```

The result will be as follows:

```
glass
```

See also

There are many other pretrained models available, including in other languages; see **http://vectors.nlpl.eu/repository/** for more details.

Some of the pretrained models include part of speech information, which can be helpful when you're disambiguating words. These models concatenate words with their **part-of-speech** (**POS**), such as cat_NOUN, so keep that in mind when using them.

To learn more about the theory behind word2vec, you could start here: **https://jalammar.github.io/illustrated-word2vec/**.

Training your own embeddings model

We can now train our own **`word2vec`** model on a corpus. For this task, we will use the top 20 Project Guttenberg books, which includes *The Adventures of Sherlock Holmes*. The reason for this is that training a model on just one book will result in suboptimal results. Once we get more text, the results will be better.

Getting ready

You can download the dataset for this recipe from Kaggle: **https://www.kaggle.com/currie32/project-gutenbergs-top-20-books**. The dataset includes files in RTF format, so you will have to save them as text. We will use the same package, **gensim**, to train our custom model.

We will use the **pickle** package to save the model on disk. If you do not have it installed, install it by using pip:

```
pip install pickle
```

How to do it...

We will read in all 20 books and use the text to create a **word2vec** model. Make sure all the books are located in one directory. Let's get started:

1. Import the necessary packages and functions:
```
import gensim
import pickle
from os import listdir
from os.path import isfile, join
from Chapter03.bag_of_words import get_sentences
from Chapter01.tokenization import tokenize_nltk
```
2. Assign the path of the **books** directory and the model path (where the model will be saved) to variables:
```
word2vec_model_path = "word2vec.model"
books_dir = "1025_1853_bundle_archive"
```
3. The **get_all_book_sentences** function will read all the text files from a directory and return a list containing all the sentences from them:
```
def get_all_book_sentences(directory):
    text_files = \
    [join(directory, f) for f in listdir(directory)
if \
     isfile(join(directory, f)) and ".txt" in f]
    all_sentences = []
    for text_file in text_files:
        sentences = get_sentences(text_file)
        all_sentences = all_sentences + sentences
    return all_sentences
```

4. The **`train_word2vec`** function will train the model and save it to a file using **`pickle`**:

```
def train_word2vec(words, word2vec_model_path):
    model = gensim.models.Word2Vec(words, window=5,
                                   size=200)
    model.train(words, total_examples=len(words),
                epochs=200)
    pickle.dump(model, open(word2vec_model_path,
'wb'))
    return model
```

5. Get the **`books`** directory's sentences:

```
sentences = get_all_book_sentences(books_dir)
```

6. Tokenize and lowercase all the sentences:

```
sentences = [tokenize_nltk(s.lower()) for s in
sentences]
```

7. Train the model. This step will take several minutes to run:

```
model = train_word2vec(sentences,
word2vec_model_path)
```

8. We can now see which most similar words the model returns for different input words, such as *river*:

```
w1 = "river"
words = model.wv.most_similar(w1, topn=10)
print(words)
```

   Every time a model is trained, the results will be different. My results look like this:

```
[('shore', 0.5025173425674438), ('woods',
0.46839720010757446), ('raft',
0.44671306014060974), ('illinois',
0.44637370109558105), ('hill', 0.4400100111961365),
('island', 0.43077412247657776), ('rock',
0.4293714761734009), ('stream',
0.42731013894081116), ('strand',
0.42297834157943726), ('road',
0.41813182830810547)]
```

How it works...

The code trains a neural network that predicts a word when given a sentence with words blanked out. The byproduct of the neural network being trained is a vector representation for each word in the training vocabulary.

In *step 1*, we import the necessary functions and classes. In *step 2*, we initialize the directory and model variables. The directory should contain the books we are going to train the model on, while the model path should be where the model will be saved.

In *step 3*, we create the **`get_all_book_sentences`** function, which will return all the sentences in all the books in the dataset. The first line in this function creates a list of all text files in the given directory. Next, we have a loop where we get the sentences for each of the text files and add them to the **`all_sentences`** array, which we return at the end.

In *step 4*, we create the function that will train the word2vec model. The only required argument is the list of words, though some of the other important ones are **`min_count`**, **`size`**, **`window`**, and **`workers`**. **`min_count`** is the minimum number of times a word has to occur in the training corpus, with the default being 5. The **`size`** parameter sets the size of the word vector. **`window`** restricts the maximum number of words between the predicted and current word in a sentence. **`workers`** is the number of working threads; the more there are, the quicker the training will proceed. When training the model, the **`epoch`** parameter will determine the number of training iterations the model will go through.

At the end of the function, we save the model to the provided path.

In *step 6*, we get all the sentences from the books using the previously defined function. In *step 7*, we lowercase and tokenize them into words. Then, we train the word2vec model using the **`train_word2vec`** function; this will take several minutes. In *step 8*, we print out the words that are the most similar to the word *river* using the newly trained model. Since the model is different every time you train it, your results will be different from mine, but the resulting words should still be similar to *river* in the sense that they will be about nature.

There's more...

There are tools we can use to evaluate a **`word2vec`** model, although its creation is unsupervised. **`gensim`** comes with a file that lists word analogies, such as *Athens* to *Greece* being the same as *Moscow* to *Russia*. The **`evaluate_word_analogies`** function runs the analogies through the model and calculates how many were correct.

Here is how we can do this:

1. Import the necessary packages and functions:
```
from gensim.test.utils import datapath
from gensim.models import KeyedVectors
import pickle
```
2. Load the previously pickled model:
```
model = pickle.load(open(word2vec_model_path,
'rb'))
```
3. Evaluate the model against the provided file. This file is available in this book's GitHub repository at **`Chapter03/questions-words.txt`**:
```
(analogy_score, word_list) = \
model.wv.evaluate_word_analogies(datapath('question
s-words.txt'))
```
4. The score is the ratio of correct analogies, so a score of 1 would mean that all the analogies were correctly answered using the model, while a score of 0 means that none were. The word list is a detailed breakdown by word analogy. We can print the analogy score like so:
```
print(analogy_score)
```
The result will be as follows:
```
0.20059045432179762
```
5. We can now load the pretrained model and compare its performance to the 20-book model. These commands might take a few minutes to run:
```
pretrained_model_path = "Chapter03/40/model.bin"
pretrained_model = \
KeyedVectors.load_word2vec_format(pretrained_model_
path,
                                  binary=True)
(analogy_score, word_list) = \
pretrained_model.evaluate_word_analogies(datapath('
questions-words.txt'))
print(analogy_score)
```
The result will be as follows:
```
0.5867802524889665
```

The pretrained model was trained on a much larger corpus, and, predictably, performs better. However, it still doesn't get a very high score. Your evaluation should be based on the type of text you are going to be working with, since the file that's provided with the **`gensim`** package is a generic evaluation

generic evaluation.

IMPORTANT NOTE

Make sure your evaluation is based on the type of data that you are going to be using in your application; otherwise, you risk having misleading evaluation results.

See also

There is an additional way of evaluating the model's performance; that is, by comparing the similarity between word pairs that have been assigned by the model to the human-assigned judgments. You can do this by using the **`evaluate_word_pairs`** function and the provided **`wordsim353.tsv`** data file. You can find out more at **https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedvectors.FastTextKeyedVectors.evaluate_word_pairs**.

Representing phrases – phrase2vec

Encoding words is useful, but usually, we deal with more complex units, such as phrases and sentences. Phrases are important because they specify more detail than just words. For example, the phrase *delicious fried rice* is very different than just the word *rice*.

In this recipe, we will train a **`word2vec`** model that uses phrases as well as words.

Getting ready

We will be using the Yelp restaurant review dataset in this recipe, which is available here: **https://www.yelp.com/dataset** (the file is about 4 GB.) Download the file and unzip it in the **`Chapter03`** folder. I downloaded the dataset in September 2020, and the results in the recipe are from that download. Your results might differ, since the dataset is updated by Yelp periodically.

The dataset is multilingual, and we will be working with the English reviews. In order to filter them, we will need the **`langdetect`** package. Install it using **`pip`**:

```
pip install langdetect
```

How to do it...

Our recipe will consist of two parts. The first part will discover phrases and tag them in the corpus, while the second part will train a **word2vec** model, following the same steps from the previous recipe. This recipe was inspired by reading Kavita Ganesan's work (**https://kavita-ganesan.com/how-to-incorporate-phrases-into-word2vec-a-text-mining-approach/**), and the idea of using stopwords as boundaries from phrases has been taken from there. Let's get started:

1. Import the necessary packages and functions:

```
import nltk
import string
import csv
import json
import pandas as pd
import gensim
from langdetect import detect
import pickle
from nltk import FreqDist
from Chapter01.dividing_into_sentences import \
divide_into_sentences_nltk
from Chapter01.tokenization import tokenize_nltk
from Chapter01.removing_stopwords import
read_in_csv
```

2. Assign the path of the Yelp! reviews JSON file, the stopwords path, and read in the stopwords:

```
stopwords_file = "Chapter01/stopwords.csv"
stopwords = read_in_csv(stopwords_file)
yelp_reviews_file = "Chapter03/yelp-dataset/
review.json"
```

3. The **get_yelp_reviews** function will read the first 10,000 lines from the file and only filter out English text:

```
def get_yelp_reviews(filename):
    reader = pd.read_json(filename,
orient="records",
                          lines=True,
                          chunksize=10000)
    chunk = next(reader)
```

```
    text = ''
    for index, row in chunk.iterrows():
        row_text =row['text']
        lang = detect(row_text)
        if (lang == "en"):
            text = text + row_text.lower()
    return text
```

4. The **`get_phrases`** function records all the phrases that have been found in the text, and then creates a dictionary with the original phrase as the key and the phrases with underscores instead of spaces as the entry. The boundaries are stopwords or punctuation. Using this function, we will get all the phrases in the text and then annotate them in the dataset:

```
def get_phrases(text):
    words = nltk.tokenize.word_tokenize(text)
    phrases = {}
    current_phrase = []
    for word in words:
        if (word in stopwords or word in \
            string.punctuation):
            if (len(current_phrase) > 1):
                phrases[" ".join(current_phrase)] =
\
                "_".join(current_phrase)
                current_phrase = []
        else:
            current_phrase.append(word)
    if (len(current_phrase) > 1):
        phrases[" ".join(current_phrase)] = \
        "_".join(current_phrase)
    return phrases
```

5. The **`replace_phrases`** function takes a corpus of text and replace phrases with their underscored versions:

```
def replace_phrases(phrases_dict, text):
    for phrase in phrases_dict.keys():
        text = text.replace(phrase,
phrases_dict[phrase])
    return text
```

6. The **`write_text_to_file`** function takes a string and a filename as input and will write the text to the specified file:

```
def write_text_to_file(text, filename):
```

```
def write_text_to_file(text, filename):
    text_file = open(filename, "w",
encoding="utf-8")
    text_file.write(text)
    text_file.close()
```

7. The **create_and_save_frequency_dist** function takes a word list and a filename as input, creates a frequency distribution, and pickles it to the provided file:

```
def create_and_save_frequency_dist(word_list,
filename):
    fdist = FreqDist(word_list)
    pickle.dump(fdist, open(filename, 'wb'))
    return fdist
```

8. Now, we can use the preceding functions to discover and tag the phrases, and then train the model. First, read in the Yelp! reviews, find the phrases, and substitute the spaces in them with underscores in the original text. We can then save the transformed text to a file:

```
text = get_yelp_reviews(yelp_reviews_file)
phrases = get_phrases(text)
text = replace_phrases(phrases, text)
write_text_to_file(text, "Chapter03/all_text.txt")
```

9. Now, we will create a **FreqDist** object to look at the most common words and phrases. First, divide the text into sentences, then divide each sentence into words, and then create a flat word list instead of a list of lists (we will use the list of lists later to train the model):

```
sentences = divide_into_sentences_nltk(text)
all_sentence_words=[tokenize_nltk(sentence.lower())
for \
                       sentence in sentences]
flat_word_list = [word.lower() for sentence in \
                  all_sentence_words for word in
                  sentence]
fdist = \
create_and_save_frequency_dist(flat_word_list,
                               "Chapter03/
fdist.bin")
```

10. We can print the most frequent words from the **FreqDist** object:

```
print(fdist.most_common()[:1000])
```

The result will be as follows:

```
[('.', 70799), ('the', 64657), (',', 49045),
('and', 40782), ('i', 38192), ('a', 35892), ('to',
```

```
26827), ('was', 23567), ('it', 21520), ('of',
19241), ('is', 16366), ('for', 15530), ('!',
14724), ('in', 14670), ('that', 12265), ('you',
11186), ('with', 10869), ('my', 10508), ('they',
10311), ('but', 9919), ('this', 9578), ('we',
9387), ("n't", 9016), ('on', 8951), ("'s", 8732),
('have', 8378), ('not', 7887), ('were', 6743),
('are', 6591), ('had', 6586), ('so', 6528), (')',
6251), ('at', 6243), ('as', 5898), ('(', 5563),
('there', 5341), ('me', 4819), ('be', 4567), ('if',
4465), ('here', 4459), ('just', 4401), ('all',
4357), ('out', 4241), ('like', 4216), ('very',
4138), ('do', 4064), ('or', 3759), …]
```

11. We will now train the **`word2vec`** model:

```
model = \
create_and_save_word2vec_model(all_sentence_words,
                               "Chapter03/
phrases.model")
```

12. We can now test the model by looking at words that are the most similar to *highly recommend* and *happy hour*:

```
words = model.wv.most_similar("highly_recommend",
topn=10)
print(words)
words = model.wv.most_similar("happy_hour",
topn=10)
print(words)
```

The result will be as follows:

```
[('recommend', 0.7311313152313232),
('would_definitely_recommend', 0.7066166400909424),
('absolutely_recommend', 0.6534838676452637),
('definitely_recommend', 0.6242724657058716),
('absolutely_love', 0.5880271196365356),
('reccomend', 0.5669443011283875),
('highly_recommend_going', 0.5308369994163513),
('strongly_recommend', 0.5057551860809326),
('recommend_kingsway_chiropractic',
0.5053386688232422), ('recommending',
0.5042617321014404)]
[('lunch', 0.5662612915039062), ('sushi',
0.5589481592178345), ('dinner',
0.5486307740211487), ('brunch',
```

```
0.5425440669059753), ('breakfast',
0.5249745845794678), ('restaurant_week',
0.4805092215538025), ('osmosis',
0.44396835565567017), ('happy_hour.wow',
0.4393075406551361), ('actual_massage',
0.43787407875061035), ('friday_night',
0.4282568395137787)]
```

13. We can also test out less frequent phrases, such as *fried rice* and *dim sum*:

```
words = model.wv.most_similar("fried_rice",
topn=10)
print(words)
words = model.wv.most_similar("dim_sum", topn=10)
print(words)
```

The result will be as follows:

```
[('pulled_pork', 0.5275152325630188),
('pork_belly', 0.5048087239265442), ('beef',
0.5020794868469238), ('hollandaise',
0.48470234870910645), ('chicken',
0.47735923528671265), ('rice', 0.4758814871311188),
('pork', 0.457661509513855), ('crab_rangoon',
0.4489888846874237), ('lasagna',
0.43910956382751465), ('lettuce_wraps',
0.4385792315006256)]
[('late_night', 0.4120914041996002), ('lunch',
0.4054332971572876), ('meal.we',
0.3739640414714813), ('tacos', 0.3505086302757263),
('breakfast', 0.34057727456092834),
('high_end_restaurant', 0.33562248945236206),
('cocktails', 0.3332172632217407),
('lunch_specials', 0.33315491676330566),
('longer_period', 0.33072057366371155),
('bubble_tea', 0.32894694805145264)]
```

How it works…

Many of the processing steps in this recipe take a long time, and for that reason, we save the intermediate results to files.

In *step 1*, we import the necessary packages and functions. In *step 2*, we assign the variable definitions for the stopwords file and the Yelp! re-

views file and then read in the stopwords.

In *step 3*, we define the **`get_yelp_reviews`** function, which reads in the reviews. The reviews file is 3.9 GB in size and might not fit into memory (or it will fit and slow down your computer immensely). In order to solve this problem, we can use the **`pandas read_json`** method, which lets us read a specified number of lines at once. In the code, I just used the first 10,000 lines, though there are many more. This method creates a **`pandas Dataframe`** object. We iterate through the rows of the object one by one and use the **`langdetect`** package to determine if the review text is in English. Then, we only include English reviews in the text.

In *step 4*, we define the **`get_phrases`** function. This function takes in the review corpus as input and detects that they are semantically one unit, such as *fried rice*. The function finds them by considering punctuation and stopwords as border tokens. Anything between them is considered a phrase. Once we've found all the phrases, we substitute spaces with underscores so that the **`word2vec`** module considers them as one token. The function returns a dictionary where the key is the input phrase with space(s) and the value is the phrase with an underscore.

In *step 5*, we define the **`replace_phrases`** function, which takes in the review corpus and replaces all the phrases from the **`phrases_dict`** object. Now, the corpus contains all the phrases with underscores in them.

In *step 6*, we define the **`write_text_to_file`** function, which will save the provided text to the file with the provided filename.

In *step 7*, we create the **`create_and_save_frequency_dist`** function, which creates a **`FreqDist`** object from the provided corpus and saves it to a file. The reason we create this function is to show that high frequency phrases, such as **highly recommend** and *happy hour*, have good results with **`word2vec`**, while with lower frequency phrases, such as *fried rice* and *dim sum*, the quality of similar words starts to decline. The solution would be to process larger quantities of data, which, of course, would slow down the tagging and training process.

In *step 8*, we create the text review corpus by using the **`get_yelp_reviews`** function. We then create the phrase dictionary by using the **`get_phrases`** function. Finally, we write the resulting corpus to a file.

In *step 9*, we create a **`FreqDist`** object, which will show us the most com-

In *step 9*, we create a **`FreqDist`** object, which will show us the most common words and phrases in the corpus. First, we divide the text into sentences. Then, we use a list comprehension to get all the words in the sentences. After that, we flatten the resulting double list and also lowercase all the words. Finally, we use the **`create_and_save_frequency_dist`** function to create the **`FreqDist`** object.

In *step 10*, we print the most common words and phrases from the frequency distribution. The result shows the most common words in the corpus. Your numbers and the order of the words might be different, as the Yelp dataset is regularly updated. In *steps 11* and *12*, you can use other phrases that are more or less frequent in your results.

In *step 11*, we train the **`word2vec`** model. In *step 12*, we print out the words that are the most similar to the phrases *highly recommend* and *happy hour*. These phrases are frequent and the similar words are indeed similar, where *recommend* and *would definitely recommend* are the most similar phrases to *highly recommend*.

In *step 13*, we print out the most similar words to the phrases *fried rice* and *dim sum*. Since these a lower frequency phrases, we can see that the words that are returned by the model as the most similar are very similar to the input.

One of the intermediate steps is creating a frequency distribution.

See also

Kavita Ganesan made a Python package for extracting phrases from large corpora using PySpark, and her code is available at **https://github.com/kavgan/phrase-at-scale/.** You can read about her approach at her blog at **https://kavita-ganesan.com/how-to-incorporate-phrases-into-word2vec-a-text-mining-approach/**.

Using BERT instead of word embeddings

A recent development in the embeddings world is **BERT**, also known as **Bidirectional Encoder Representations from Transformers**, which, like word embeddings, gives a vector representation, but it takes context into account and can represent a whole sentence. We can use the Hugging

into account and can represent a whole sentence. We can use the Hugging Face **sentence_transformers** package to represent sentences as vectors.

Getting ready

For this recipe, we need to install PyTorch with Torchvision, and then the transformers and sentence transformers from Hugging Face. Follow these installation steps in an Anaconda prompt. For Windows, use the following code:

```
conda install pytorch torchvision cudatoolkit=10.2 -c
pytorch
pip install transformers
pip install -U sentence-transformers
```

For macOS, use the following code:

```
conda install pytorch torchvision torchaudio -c
pytorch
pip install transformers
pip install -U sentence-transformers
```

How to do it...

The Hugging Face code makes using BERT very easy. The first time the code runs, it will download the necessary model, which might take some time. Once you've downloaded it, it's just a matter of encoding the sentences using the model. We will use the **sherlock_holmes_1.txt** file we've used previously for this. Let's get started:

1. Import the **SentenceTransformer** class and helper methods:
```
from sentence_transformers import
SentenceTransformer
from Chapter01.dividing_into_sentences import
read_text_file, \
divide_into_sentences_nltk
```
2. Read the text file and divide the text into sentences:
```
text = read_text_file("sherlock_holmes.txt")
sentences = divide_into_sentences_nltk(text)
```
3. Load the sentence transformer model:
```
model = SentenceTransformer('bert-base-nli-mean-
tokens')
```
4. Get the sentence embeddings:

```
sentence_embeddings = model.encode(sentences)
```

The result will be as follows:

```
[[-0.41089028  1.1092614   0.653306   ...
-0.9232089   0.4728682
   0.36298898]
[-0.16485551  0.6998439   0.7076392  ...
-0.40428287 -0.30385852
  -0.3291511 ]
[-0.37814915  0.34771013 -0.09765357
...  0.13831234  0.3604408
   0.12382   ]
...
[-0.25148678  0.5758055   1.4596572  ...  0.5689018
4 -0.6003894
  -0.02739916]
[-0.64917654  0.3609671   1.1350367  ...
-0.04054655  0.07568665
   0.1809447 ]
[-0.4241825   0.48146117  0.93001956
...  0.73677135 -0.09357803
  -0.0036802 ]]
```

5. We can also encode a part of a sentence, such as a noun chunk:

```
sentence_embeddings = model.encode(["the beautiful
lake"])
print(sentence_embeddings)
```

The result will be as follows:

```
[[-7.61981383e-02
-5.74670374e-01  1.08264232e+00  7.36554384e-01
   5.51345229e-01 -9.39117730e-01 -2.80430615e-01
-5.41626096e-01
   7.50949085e-01 -4.40971524e-01  5.31526923e-01
-5.41883349e-01
   1.92792594e-01  3.44117582e-01  1.50266397e+00
-6.26989722e-01
  -2.42828876e-01 -3.66734862e-01  5.57459474e-01
-2.21802562e-01 …]]
```

How it works…

The sentence transformer's BERT model is a pre-trained model, just like a **word2vec** model, that encodes a sentence into a vector. The difference

word2vec model, that encodes a sentence into a vector. The difference between a **word2vec** model and a sentence transformer model is that we encode sentences in the latter, and not words.

In *step 1*, we import the **SentenceTransformer** object and the helper functions. In *step 2*, we read in the text of the **sherlock_holmes_1.txt** file and divide it into sentences. In *step 3*, we load the pretrained model and in *step 4*, we load the vectors for each of the sentences in the text. In *step 5*, we encode the phrase *the beautiful lake*. Since the encode function expects a list, we create a one-element list.

Once we've encoded the sentences using the model, we can use them in a downstream task, such as classification or sentiment analysis.

Getting started with semantic search

In this recipe, we will get a glimpse of how to get started on expanding search with the help of a **word2vec** model. When we search for a term, we expect the search engine to show us a result with a synonym when we didn't use the exact term contained in the document. Search engines are far more complicated than what we'll show in the recipe, but this should give you a taste of what it's like to build a customizable search engine.

Getting ready

We will be using an IMDb dataset from Kaggle, which can be downloaded from **https://www.kaggle.com/PromptCloudHQ/imdb-data**. Download the dataset and unzip the CSV file.

We will also use a small-scale Python search engine called Whoosh. Install it using pip:

```
pip install whoosh
```

We will also be using the pretrained **word2vec** model from the *Using word embeddings* recipe.

How to do it...

We will create a class for the Whoosh search engine that will create a document index based on the IMDb file. Then, we will load the pretrained

word2vec model and use it to augment the queries we pass to the engine. Let's get started:

1. Import the helper methods and classes:

```
from whoosh.fields import Schema, TEXT, KEYWORD,
ID, STORED, DATETIME
from whoosh.index import create_in
from whoosh.analysis import StemmingAnalyzer
from whoosh.qparser import MultifieldParser
import csv
from Chapter03.word_embeddings import
w2vec_model_path
from Chapter03.word_embeddings import load_model
```

2. Create a directory called **whoosh_index** in the **Chapter03** folder. Then, set the paths for the search engine index and the IMDb dataset path:

```
imdb_dataset_path = "Chapter03/IMDB-Movie-Data.csv"
search_engine_index_path = "Chapter03/whoosh_index"
```

3. Create the **IMDBSearchEngine** class. The complete code for this class can be found in this book's GitHub repository. The most important part of it is the **query_engine** function:

```
class IMDBSearchEngine:
…
    def query_engine(self, keywords):
        with self.index.searcher() as searcher:
            query=\
            MultifieldParser(["title",
"description"],
                                self.index.schema).\
                                parse(keywords)
            results = searcher.search(query)
            print(results)
            print(results[0])
            return results
```

4. The **get_similar_words** function takes a word and the pretrained model and returns the top three words that are similar to the given word:

```
def get_similar_words(model, search_term):
    similarity_list =
model.most_similar(search_term, topn=3)
    similar words = [sim tuple[0] for sim tuple in
```

```
\
                    similarity_list]
    return similar_words
```

5. Now, we can initialize the search engine. Use the first line to initialize the search engine when the index doesn't exist yet, and the second line when you've already created it once:

```
search_engine = \
IMDBSearchEngine(search_engine_index_path,
                 imdb_dataset_path,
                 load_existing=False)
#search_engine = \
IMDBSearchEngine(search_engine_index_path,
                 load_existing=True)
```

6. Load the **word2vec** model:

```
model = load_model(w2vec_model_path)
```

7. Let's say a user wants to find the movie *Colossal*, but forgot its real name, so they search for *gigantic*. We will use *gigantic* as the search term:

```
search_term = "gigantic"
```

8. We will get three words similar to the input word:

```
other_words = get_similar_words(model, search_term)
```

9. We will then query the engine to return all the movies containing those words:

```
results = \
search_engine.query_engine(" OR
".join([search_term] +
                            other_words))
print(results[0])
```

10. The result will be the movie with an ID of 15, which is the movie *Colossal*:

```
<Hit {'movie_id': '15'}>
```

How it works...

In *step 1*, we import the necessary packages and functions. In *step 2*, we initialize the dataset and search engine paths. Make sure to create a directory called **whoosh_index** in the **Chapter03** folder.

In *step 3*, we create the **IMDBSearchEngine** class. The constructor takes the following arguments: the path to the search engine index, the path to the CSV dataset, which is an empty string by default, and the **load ex-**

the IMDB dataset, which is an empty string by default, and the **load_existing** boolean argument, which is **False** by default. If **load_existing** is **False**, **imdb_path** needs to point to the dataset. In this case, a new index will be created at the path provided by the **index_path** variable. If **load_existing** is **True**, the **imdb_path** variable is ignored, and the existing index is loaded from **index_path**.

All the indices are created using a schema, and this search engine's schema is created in the **create_schema** function. The schema specifies which fields the information about the document will contain. In this case, it is the movie's title, its description, genre, director, actors, and year of release. The document index is then created using the **populate_index** function. Once the index has been populated, we do not need to reindex it, and can open the index from disk.

The **query_engine** function searches the index with the keywords that are sent to it. When creating a query parser, we use the **MultifieldParser** class so that we can search through multiple fields; in this case, title and description.

In *step 4*, we create the **get_similar_words** function, which returns the top three words that are similar to the word being passed in using a **word2vec** model. This should be familiar from the *Using word embeddings* recipe. The function gets the list of word similarity score tuples and returns a list of words.

In *step 5*, we create the **search_engine** object. There are two lines, the first one creates a new search engine from scratch, while the second one loads an existing index. You should run the first line the first time and the second line every other time after that.

In *step 6*, we load the **word2vec** model.

In *step 7*, we set the search term to *gigantic*. In *step 8*, we get the top three similar words for the initial search term using the **get_similar_words** function. In *step 9*, we use the **search_engine** object to perform the search using the original search term and the similar words as the query. The result is movie number 15, *Colossal*, which is the correct answer.

See also

Please refer to the official Whoosh documentation to explore how it's

used. It can be found at **https://whoosh.readthedocs.io/en/latest/**.