

Chapter 6. Analyzing Text Data

In this chapter, we will cover the following recipes:

- Preprocessing data using tokenization
- Stemming text data
- Converting text to its base form using lemmatization
- Dividing text using chunking
- Building a bag-of-words model
- Building a text classifier
- Identifying the gender
- Analyzing the sentiment of a sentence
- Identifying patterns in text using topic modeling

Introduction

Text analysis and **natural language processing (NLP)** is an integral part of modern artificial intelligence systems. Computers are good at understanding rigidly-structured data with limited variety. However, when we deal with unstructured free-form text, things begin to get difficult. Developing NLP applications is challenging because computers have a hard time understanding underlying concepts. There are also many subtle variations to the way in which we communicate things. These can be in the form of dialects, context, slang, and so on.

In order to solve this problem, NLP applications are developed based on machine learning. These algorithms detect patterns in text data so that we can extract insights from it. Artificial intelligence companies make heavy use of NLP and text analysis to deliver relevant results. Some of the most common applications of NLP include search engines, sentiment analysis, topic modeling, part-of-speech tagging, entity recognition, and so on. The goal of NLP is to develop a set of algorithms so that we can interact with computers in plain English. If we can achieve this, then we wouldn't need programming languages to instruct computers about what they should do. In this chapter, we will look at a few recipes that focus on text analysis and how we can extract meaningful information from text data. We will use a Python package called **Natural Language Toolkit (NLTK)** heavily in this chapter. Make sure that you install this before you proceed. You can find the installation steps at <http://www.nltk.org/install.html>. You also need to install NLTK Data, which contains many corpora and trained models. This is an integral part of text analysis! You can find the installation steps at <http://www.nltk.org/data.html>.

Preprocessing data using tokenization

Tokenization is the process of dividing text into a set of meaningful pieces. These pieces are called **tokens**. For example, we can divide a chunk of text into words, or we can divide it into sentences. Depending on the task at hand, we can define our own conditions to divide the input text into meaningful tokens. Let's take a look at how to do this.

How to do it...

1. Create a new Python file and add the following lines. Let's define some sample text for analysis:

```
text = "Are you curious about tokenization? Let's see how it  
works! We need to analyze a couple of sentences with  
punctuations to see it in action."
```

2. Let's start with sentence tokenization. NLTK provides a sentence tokenizer, so let's import this:

```
# Sentence tokenization  
from nltk.tokenize import sent_tokenize
```

3. Run the sentence tokenizer on the input text and extract the tokens:

```
sent_tokenize_list = sent_tokenize(text)
```

4. Print the list of sentences to see whether it works correctly:

```
print "\nSentence tokenizer:"  
print sent_tokenize_list
```

5. Word tokenization is very commonly used in NLP. NLTK comes with a couple of different word tokenizers. Let's start with the basic word tokenizer:

```
# Create a new word tokenizer  
from nltk.tokenize import word_tokenize
```

```
print "\nWord tokenizer:"  
print word_tokenize(text)
```

6. There is another word tokenizer that is available called PunktWord Tokenizer. This splits the text on punctuation, but this keeps it within the words:

```
# Create a new punkt word tokenizer  
from nltk.tokenize import PunktWordTokenizer
```

```
punkt_word_tokenizer = PunktWordTokenizer()  
print "\nPunkt word tokenizer:"  
print punkt_word_tokenizer.tokenize(text)
```

7. If you want to split these punctuations into separate tokens, then we need to use WordPunct Tokenizer:

```
# Create a new WordPunct tokenizer
from nltk.tokenize import WordPunctTokenizer

word_punct_tokenizer = WordPunctTokenizer()
print "\nWord punct tokenizer:"
print word_punct_tokenizer.tokenize(text)
```

8. The full code is in the `tokenizer.py` file. If you run this code, you will see the following output on your Terminal:

```
Sentence tokenizer:
['Are you curious about tokenization?', "Let's see how it works!", 'We need to analyze a couple of sentences with punctuations to see it in action.']

Word tokenizer:
['Are', 'you', 'curious', 'about', 'tokenization', '?', 'Let', "'s", 'see', 'how', 'it', 'works', '!', 'We', 'need', 'to', 'analyze', 'a', 'couple', 'of', 'sentences', 'with', 'punctuations', 'to', 'see', 'it', 'in', 'action', '.']

Punkt word tokenizer:
['Are', 'you', 'curious', 'about', 'tokenization', '?', 'Let', "'s", 'see', 'how', 'it', 'works', '!', 'We', 'need', 'to', 'analyze', 'a', 'couple', 'of', 'sentences', 'with', 'punctuations', 'to', 'see', 'it', 'in', 'action.']

Word punct tokenizer:
['Are', 'you', 'curious', 'about', 'tokenization', '?', 'Let', "'", 's', 'see', 'how', 'it', 'works', '!', 'We', 'need', 'to', 'analyze', 'a', 'couple', 'of', 'sentences', 'with', 'punctuations', 'to', 'see', 'it', 'in', 'action', '.']
```

Stemming text data

When we deal with a text document, we encounter different forms of a word. Consider the word "play". This word can appear in various forms, such as "play", "plays", "player", "playing", and so on. These are basically families of words with similar meanings. During text analysis, it's useful to extract the base form of these words. This will help us in extracting some statistics to analyze the overall text. The goal of stemming is to reduce these different forms into a common base form. This uses a heuristic process to cut off the ends of words to extract the base form. Let's see how to do this in Python.

How to do it...

1. Create a new Python file, and import the following packages:

```
from nltk.stem.porter import PorterStemmer
from nltk.stem.lancaster import LancasterStemmer
from nltk.stem.snowball import SnowballStemmer
```

2. Let's define a few words to play with, as follows:

```
words = ['table', 'probably', 'wolves', 'playing', 'is',
         'dog', 'the', 'beaches', 'grounded', 'dreamt',
         'envision']
```

3. We'll define a list of stemmers that we want to use:

```
# Compare different stemmers
stemmers = ['PORTER', 'LANCASTER', 'SNOWBALL']
```

4. Initialize the required objects for all three stemmers:

```
stemmer_porter = PorterStemmer()
stemmer_lancaster = LancasterStemmer()
stemmer_snowball = SnowballStemmer('english')
```

5. In order to print the output data in a neat tabular form, we need to format it in the right way:

```
formatted_row = '{:>16}' * (len(stemmers) + 1)
print '\n', formatted_row.format('WORD', *stemmers), '\n'
```

6. Let's iterate through the list of words and stem them using the three stemmers:

```
for word in words:
    stemmed_words = [stemmer_porter.stem(word),
                     stemmer_lancaster.stem(word),
                     stemmer_snowball.stem(word)]
    print formatted_row.format(word, *stemmed_words)
```

7. The full code is in the `stemmer.py` file. If you run this code, you will see the following output on your Terminal. Observe how the Lancaster stemmer behaves differently for a couple of words:

WORD	PORTER	LANCASTER	SNOWBALL
table	tabl	tabl	tabl
probably	probabl	prob	probabl
wolves	wolv	wolv	wolv
playing	play	play	play
is	is	is	is
dog	dog	dog	dog
the	the	the	the
beaches	beach	beach	beach
grounded	ground	ground	ground
dreamt	dreamt	dreamt	dreamt
envision	envis	envid	envis

How it works...

All three stemming algorithms basically aim to achieve the same thing. The difference between the three stemming algorithms is basically the level of strictness with which they operate. If you observe the outputs, you will see that the Lancaster stemmer is stricter than the other two stemmers. The Porter stemmer is the least in terms of strictness and Lancaster is the strictest. The stemmed words that we get from Lancaster stemmer tend to get confusing and obfuscated. The algorithm is really fast but it will reduce the words a lot. So, a good rule of thumb is to use the Snowball stemmer.

Converting text to its base form using lemmatization

The goal of lemmatization is also to reduce words to their base forms, but this is a more structured approach. In the previous recipe, we saw that the base words that we obtained using stemmers don't really make sense. For example, the word "wolves" was reduced to "wolv", which is not a real word. Lemmatization solves this problem by doing things using a vocabulary and morphological analysis of words. It removes inflectional word endings, such as "ing" or "ed", and returns the base form of a word. This base form is known as the lemma. If you lemmatize the word "wolves", you will get "wolf" as the output. The output depends on whether the token is a verb or a noun. Let's take a look at how to do this in this recipe.

How to do it...

1. Create a new Python file, and import the following package:

```
from nltk.stem import WordNetLemmatizer
```

2. Let's define the same set of words that we used during stemming:

```
words = ['table', 'probably', 'wolves', 'playing', 'is',  
         'dog', 'the', 'beaches', 'grounded', 'dreamt',  
         'envision']
```

3. We will compare two lemmatizers, the NOUN and VERB lemmatizers. Let's list them as follows:

```
# Compare different lemmatizers  
lemmatizers = ['NOUN LEMMATIZER', 'VERB LEMMATIZER']
```

4. Create the object based on WordNet lemmatizer:

```
lemmatizer_wordnet = WordNetLemmatizer()
```

5. In order to print the output in a tabular form, we need to format it in the right way:

```
formatted_row = '{:>24}' * (len(lemmatizers) + 1)  
print '\n', formatted_row.format('WORD', *lemmatizers), '\n'
```

6. Iterate through the words and lemmatize them:

```
for word in words:  
    lemmatized_words = [lemmatizer_wordnet.lemmatize(word,  
pos='n'),  
                        lemmatizer_wordnet.lemmatize(word, pos='v')]  
    print formatted_row.format(word, *lemmatized_words)
```

7. The full code is in the `lemmatizer.py` file. If you run this code, you will see the following output. Observe how NOUN and VERB lemmatizers differ when they lemmatize the word "is" in the following image:

WORD	NOUN LEMMATIZER	VERB LEMMATIZER
table	table	table
probably	probably	probably
wolves	wolf	wolves
playing	playing	play
is	is	be
dog	dog	dog
the	the	the
beaches	beach	beach
grounded	grounded	ground
dreamt	dreamt	dream
envision	envision	envision

Dividing text using chunking

Chunking refers to dividing the input text into pieces, which are based on any random condition. This is different from tokenization in the sense that there are no constraints and the chunks do not need to be meaningful at all. This is used very frequently during text analysis. When you deal with really large text documents, you need to divide it into chunks for further analysis. In this recipe, we will divide the input text into a number of pieces, where each piece has a fixed number of words.

How to do it...

1. Create a new Python file, and import the following packages:

```
import numpy as np
from nltk.corpus import brown
```

2. Let's define a function to split text into chunks. The first step is to divide the text based on spaces:

```
# Split a text into chunks
def splitter(data, num_words):
    words = data.split(' ')
    output = []
```

3. Initialize a couple of required variables:

```
cur_count = 0
cur_words = []
```

4. Let's iterate through the words:

```
for word in words:
    cur_words.append(word)
    cur_count += 1
```

5. Once you hit the required number of words, reset the variables:

```
if cur_count == num_words:
    output.append(' '.join(cur_words))
    cur_words = []
    cur_count = 0
```

6. Append the chunks to the output variable, and return it:

```
output.append(' '.join(cur_words) )

return output
```

7. We can now define the main function. Load the data from Brown corpus. We will use the first 10,000 words:


```
if __name__=='__main__':  
    # Read the data from the Brown corpus  
    data = ' '.join(brown.words()[:10000])
```

8. Define the number of words in each chunk:

```
# Number of words in each chunk  
num_words = 1700
```

9. Initialize a couple of relevant variables:

```
chunks = []  
counter = 0
```

10. Call the `splitter` function on this text data and print the output:

```
text_chunks = splitter(data, num_words)  
  
print "Number of text chunks =", len(text_chunks)
```

11. The full code is in the `chunking.py` file. If you run this code, you will see the number of chunks generated printed on the Terminal. It should be 6!

Building a bag-of-words model

When we deal with text documents that contain millions of words, we need to convert them into some kind of numeric representation. The reason for this is to make them usable for machine learning algorithms. These algorithms need numerical data so that they can analyze them and output meaningful information. This is where the **bag-of-words** approach comes into picture. This is basically a model that learns a vocabulary from all the words in all the documents. After this, it models each document by building a histogram of all the words in the document.

How to do it...

1. Create a new Python file, and import the following packages:

```
import numpy as np
from nltk.corpus import brown
from chunking import splitter
```

2. Let's define the main function. Load the input data from the Brown corpus:

```
if __name__ == '__main__':
    # Read the data from the Brown corpus
    data = ' '.join(brown.words()[:10000])
```

3. Divide the text data into five chunks:

```
# Number of words in each chunk
num_words = 2000

chunks = []
counter = 0

text_chunks = splitter(data, num_words)
```

4. Create a dictionary that is based on these text chunks:

```
for text in text_chunks:
    chunk = {'index': counter, 'text': text}
    chunks.append(chunk)
    counter += 1
```

5. The next step is to extract a document term matrix. This is basically a matrix that counts the number of occurrences of each word in the document. We will use scikit-learn to do this because it has better provisions as compared to NLTK for this particular task. Import the following package:

```
# Extract document term matrix
from sklearn.feature_extraction.text import CountVectorizer
```

6. Define the object, and extract the document term matrix:

```

vectorizer = CountVectorizer(min_df=5, max_df=.95)
doc_term_matrix = vectorizer.fit_transform([chunk['text']
for chunk in chunks])

```

7. Extract the vocabulary from the vectorizer object and print it:

```

vocab = np.array(vectorizer.get_feature_names())
print "\nVocabulary:"
print vocab

```

8. Print the document term matrix:

```

print "\nDocument term matrix:"
chunk_names = ['Chunk-0', 'Chunk-1', 'Chunk-2', 'Chunk-3',
'Chunk-4']

```

9. To print in tabular form, we need to format this, as follows:

```

formatted_row = '{:>12}' * (len(chunk_names) + 1)
print '\n', formatted_row.format('Word', *chunk_names), '\n'

```

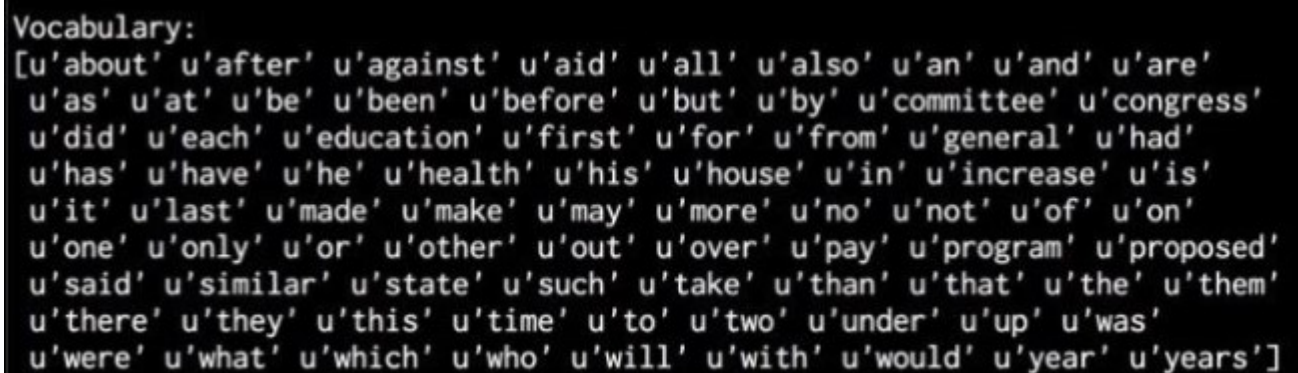
10. Iterate through the words, and print the number of times each word has occurred in different chunks:

```

for word, item in zip(vocab, doc_term_matrix.T):
    # 'item' is a 'csr_matrix' data structure
    output = [str(x) for x in item.data]
    print formatted_row.format(word, *output)

```

11. The full code is in the `bag_of_words.py` file. If you run this code, you will see two main things printed on the Terminal. The first output is the vocabulary as shown in the following image:



```

Vocabulary:
[u'about' u'after' u'against' u'aid' u'all' u'also' u'an' u'and' u'are'
u'as' u'at' u'be' u'been' u'before' u'but' u'by' u'committee' u'congress'
u'did' u'each' u'education' u'first' u'for' u'from' u'general' u'had'
u'has' u'have' u'he' u'health' u'his' u'house' u'in' u'increase' u'is'
u'it' u'last' u'made' u'make' u'may' u'more' u'no' u'not' u'of' u'on'
u'one' u'only' u'or' u'other' u'out' u'over' u'pay' u'program' u'proposed'
u'said' u'similar' u'state' u'such' u'take' u'than' u'that' u'the' u'them'
u'there' u'they' u'this' u'time' u'to' u'two' u'under' u'up' u'was'
u'were' u'what' u'which' u'who' u'will' u'with' u'would' u'year' u'years']

```

12. The second thing is the document term matrix, which is a pretty long. The first few lines will look like the following:

Document term matrix:

Word	Chunk-0	Chunk-1	Chunk-2	Chunk-3	Chunk-4
about	1	1	1	1	3
after	2	3	2	1	3
against	1	2	2	1	1
aid	1	1	1	3	5
all	2	2	5	2	1
also	3	3	3	4	3
an	5	7	5	7	10
and	34	27	36	36	41
are	5	3	6	3	2
as	13	4	14	18	4
at	5	7	9	3	6
be	20	14	7	10	18
been	7	1	6	15	5
before	2	2	1	1	2
but	3	3	2	9	5
by	8	22	15	14	12
committee	2	10	3	1	7

How it works...

Consider the following sentences:

- **Sentence 1:** The brown dog is running.
- **Sentence 2:** The black dog is in the black room.
- **Sentence 3:** Running in the room is forbidden.

If you consider all the three sentences, we have the following nine unique words:

- the
- brown
- dog
- is
- running
- black
- in
- room
- forbidden

Now, let's convert each sentence into a histogram using the count of words in each sentence. Each feature vector will be 9-dimensional because we have nine unique words:

- **Sentence 1:** [1, 1, 1, 1, 1, 0, 0, 0, 0]
- **Sentence 2:** [2, 0, 1, 1, 0, 2, 1, 1, 0]
- **Sentence 3:** [0, 0, 0, 1, 1, 0, 1, 1, 1]

Once we extract these feature vectors, we can use machine learning algorithms to analyze them.

Building a text classifier

The goal of text classification is to categorize text documents into different classes. This is an extremely important analysis technique in NLP. We will use a technique, which is based on a statistic called **tf-idf**, which stands for **term frequency—inverse document frequency**. This is an analysis tool that helps us understand how important a word is to a document in a set of documents. This serves as a feature vector that's used to categorize documents. You can learn more about it at <http://www.tfidf.com>.

How to do it...

1. Create a new Python file, and import the following package:

```
from sklearn.datasets import fetch_20newsgroups
```

2. Let's select a list of categories and name them using a dictionary mapping. These categories are available as part of the news groups dataset that we just imported:

```
category_map = {'misc.forsale': 'Sales', 'rec.motorcycles':  
               'Motorcycles',  
               'rec.sport.baseball': 'Baseball', 'sci.crypt':  
               'Cryptography',  
               'sci.space': 'Space'}
```

3. Load the training data based on the categories that we just defined:

```
training_data = fetch_20newsgroups(subset='train',  
                                   categories=category_map.keys(), shuffle=True,  
                                   random_state=7)
```

4. Import the feature extractor:

```
# Feature extraction  
from sklearn.feature_extraction.text import CountVectorizer
```

5. Extract the features using the training data:

```
vectorizer = CountVectorizer()  
X_train_termcounts =  
vectorizer.fit_transform(training_data.data)  
print "\nDimensions of training data:", X_train_termcounts.shape
```

6. We are now ready to train the classifier. We will use the Multinomial Naive Bayes classifier:

```
# Training a classifier  
from sklearn.naive_bayes import MultinomialNB  
from sklearn.feature_extraction.text import TfidfTransformer
```

7. Define a couple of random input sentences:

```
input_data = [
    "The curveballs of right handed pitchers tend to curve to
the left",
    "Caesar cipher is an ancient form of encryption",
    "This two-wheeler is really good on slippery roads"
]
```

8. Define the tf-idf transformer object and train it:

```
# tf-idf transformer
tfidf_transformer = TfidfTransformer()
X_train_tfidf =
tfidf_transformer.fit_transform(X_train_termcounts)
```

9. Once we have the feature vectors, train the Multinomial Naive Bayes classifier using this data:

```
# Multinomial Naive Bayes classifier
classifier = MultinomialNB().fit(X_train_tfidf,
training_data.target)
```

10. Transform the input data using the word counts:

```
X_input_termcounts = vectorizer.transform(input_data)
```

11. Transform the input data using the tf-idf transformer:

```
X_input_tfidf = tfidf_transformer.transform(X_input_termcounts)
```

12. Predict the output categories of these input sentences using the trained classifier:

```
# Predict the output categories
predicted_categories = classifier.predict(X_input_tfidf)
```

13. Print the outputs, as follows:

```
# Print the outputs
for sentence, category in zip(input_data, predicted_categories):
    print '\nInput:', sentence, '\nPredicted category:', \
        category_map[training_data.target_names[category]]
```

14. The full code is in the `tfidf.py` file. If you run this code, you will see the following output printed on your Terminal:

```
Dimensions of training data: (2968, 40605)
```

```
Input: The curveballs of right handed pitchers tend to curve to the left  
Predicted category: Baseball
```

```
Input: Caesar cipher is an ancient form of encryption  
Predicted category: Cryptography
```

```
Input: This two-wheeler is really good on slippery roads  
Predicted category: Motorcycles
```

How it works...

The tf-idf technique is used frequently in information retrieval. The goal is to understand the importance of each word within a document. We want to identify words that occur many times in a document. At the same time, common words like “is” and “be” don't really reflect the nature of the content. So we need to extract the words that are true indicators. The importance of each word increases as the count increases. At the same time, as it appears a lot, the frequency of this word increases too. These two things tend to balance each other out. We extract the term counts from each sentence. Once we convert this to a feature vector, we train the classifier to categorize these sentences.

The **term frequency (TF)** measures how frequently a word occurs in a given document. As multiple documents differ in length, the numbers in the histogram tend to vary a lot. So, we need to normalize this so that it becomes a level playing field. To achieve normalization, we divide term-frequency by the total number of words in a given document. The **inverse document frequency (IDF)** measures the importance of a given word. When we compute TF, all words are considered to be equally important. To counter-balance the frequencies of commonly-occurring words, we need to weigh them down and scale up the rare ones. We need to calculate the ratio of the number of documents with the given word and divide it by the total number of documents. IDF is calculated by taking the negative algorithm of this ratio.

For example, simple words, such as "is" or "the" tend to appear a lot in various documents. However, this doesn't mean that we can characterize the document based on these words. At the same time, if a word appears a single time, this is not useful either. So, we look for words that appear a number of times, but not so much that they become noisy. This is formulated in the tf-idf technique and used to classify documents. Search engines frequently use this tool to order the search results by relevance.

Identifying the gender

Identifying the gender of a name is an interesting task in NLP. We will use the heuristic that the last few characters in a name is its defining characteristic. For example, if the name ends with "la", it's most likely a female name, such as "Angela" or "Layla". On the other hand, if the name ends with "im", it's most likely a male name, such as "Tim" or "Jim". As we are sure of the exact number of characters to use, we will experiment with this. Let's see how to do it.

How to do it...

1. Create a new Python file, and import the following packages:

```
import random
from nltk.corpus import names
from nltk import NaiveBayesClassifier
from nltk.classify import accuracy as nltk_accuracy
```

2. We need to define a function to extract features from input words:

```
# Extract features from the input word
def gender_features(word, num_letters=2):
    return {'feature': word[-num_letters:].lower() }
```

3. Let's define the main function. We need some labeled training data:

```
if __name__ == '__main__':
    # Extract labeled names
    labeled_names = [(name, 'male') for name in
names.words('male.txt')] +
        [(name, 'female') for name in
names.words('female.txt')]
```

4. Seed the random number generator, and shuffle the training data:

```
random.seed(7)
random.shuffle(labeled_names)
```

5. Define some input names to play with:

```
input_names = ['Leonardo', 'Amy', 'Sam']
```

6. As we don't know how many ending characters we need to consider, we will sweep the parameter space from 1 to 5. Each time, we will extract the features, as follows:

```
# Sweeping the parameter space
for i in range(1, 5):
    print '\nNumber of letters:', i
    featuresets = [(gender_features(n, i), gender) for (n,
gender) in labeled_names]
```

7. Divide this into train and test datasets:

```
train_set, test_set = featuresets[500:],  
featuresets[:500]
```

8. We will use the Naive Bayes classifier to do this:

```
classifier = NaiveBayesClassifier.train(train_set)
```

9. Evaluate the classifier for each value in the parameter space:

```
# Print classifier accuracy  
print 'Accuracy ==>', str(100 *  
nltk_accuracy(classifier, test_set)) + str('%')  
  
# Predict outputs for new inputs  
for name in input_names:  
    print name, '==>',  
    classifier.classify(gender_features(name, i))
```

10. The full code is in the `gender_identification.py` file. If you run this code, you will see the following output printed on your Terminal:

Number of letters: 1
Accuracy ==> 76.6%
Leonardo ==> male
Amy ==> female
Sam ==> male

Number of letters: 2
Accuracy ==> 80.2%
Leonardo ==> male
Amy ==> female
Sam ==> male

Number of letters: 3
Accuracy ==> 78.4%
Leonardo ==> male
Amy ==> female
Sam ==> female

Number of letters: 4
Accuracy ==> 71.6%
Leonardo ==> male
Amy ==> female
Sam ==> female

Analyzing the sentiment of a sentence

Sentiment analysis is one of the most popular applications of NLP. Sentiment analysis refers to the process of determining whether a given piece of text is positive or negative. In some variations, we consider "neutral" as a third option. This technique is commonly used to discover how people feel about a particular topic. This is used to analyze sentiments of users in various forms, such as marketing campaigns, social media, e-commerce customers, and so on.

How to do it...

1. Create a new Python file, and import the following packages:

```
import nltk.classify.util
from nltk.classify import NaiveBayesClassifier
from nltk.corpus import movie_reviews
```

2. Define a function to extract features:

```
def extract_features(word_list):
    return dict([(word, True) for word in word_list])
```

3. We need training data for this, so we will use movie reviews in NLTK:

```
if __name__ == '__main__':
    # Load positive and negative reviews
    positive_fileids = movie_reviews.fileids('pos')
    negative_fileids = movie_reviews.fileids('neg')
```

4. Let's separate these into positive and negative reviews:

```
features_positive =
[(extract_features(movie_reviews.words(fileids=[f])),
  'Positive') for f in positive_fileids]
features_negative =
[(extract_features(movie_reviews.words(fileids=[f])),
  'Negative') for f in negative_fileids]
```

5. Divide the data into training and testing datasets:

```
# Split the data into train and test (80/20)
threshold_factor = 0.8
threshold_positive = int(threshold_factor *
len(features_positive))
threshold_negative = int(threshold_factor *
len(features_negative))
```

6. Extract the features:

```

features_train = features_positive[:threshold_positive] +
features_negative[:threshold_negative]
features_test = features_positive[threshold_positive:] +
features_negative[threshold_negative:]
print "\nNumber of training datapoints:",
len(features_train)
print "Number of test datapoints:", len(features_test)

```

7. We will use a Naive Bayes classifier. Define the object and train it:

```

# Train a Naive Bayes classifier
classifier = NaiveBayesClassifier.train(features_train)
print "\nAccuracy of the classifier:",
nltn.classify.util.accuracy(classifier, features_test)

```

8. The classifier object contains the most informative words that it obtained during analysis. These words basically have a strong say in what's classified as a positive or a negative review. Let's print them out:

```

print "\nTop 10 most informative words:"
for item in classifier.most_informative_features()[0:10]:
    print item[0]

```

9. Create a couple of random input sentences:

```

# Sample input reviews
input_reviews = [
    "It is an amazing movie",
    "This is a dull movie. I would never recommend it to
anyone.",
    "The cinematography is pretty great in this movie",
    "The direction was terrible and the story was all over
the place"
]

```

10. Run the classifier on those input sentences and obtain the predictions:

```

print "\nPredictions:"
for review in input_reviews:
    print "\nReview:", review
    probdist =
classifier.prob_classify(extract_features(review.split()))
    pred_sentiment = probdist.max()

```

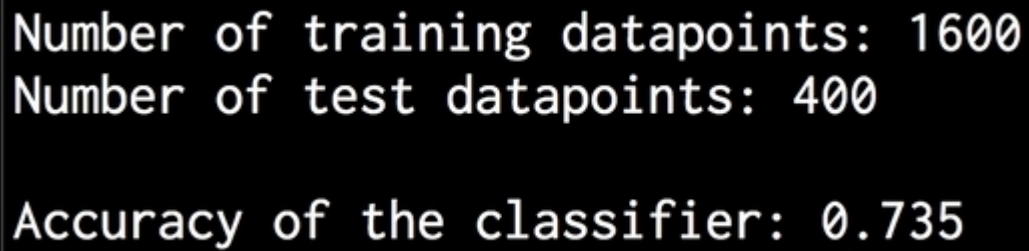
11. Print the output:

```

print "Predicted sentiment:", pred_sentiment
print "Probability:",
round(probdist.prob(pred_sentiment), 2)

```

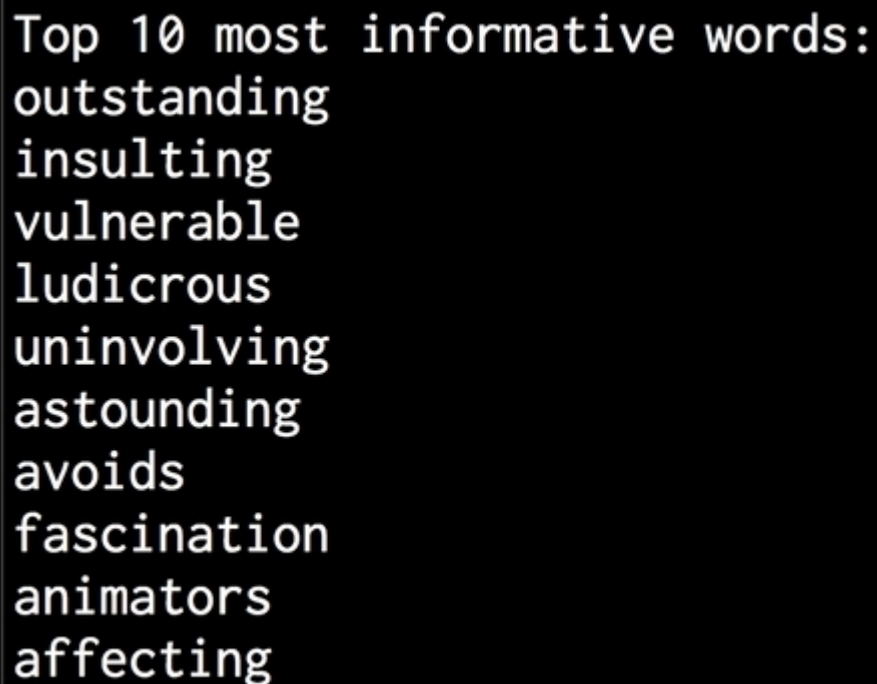
12. The full code is in the `sentiment_analysis.py` file. If you run this code, you will see three main things printed on the Terminal. The first is the accuracy, as shown in the following image:



```
Number of training datapoints: 1600
Number of test datapoints: 400

Accuracy of the classifier: 0.735
```

13. The next is a list of most informative words:



```
Top 10 most informative words:
outstanding
insulting
vulnerable
ludicrous
uninvolving
astounding
avoids
fascination
animators
affecting
```

14. The last is the list of predictions, which are based on the input sentences:

Predictions:

Review: It is an amazing movie

Predicted sentiment: Positive

Probability: 0.61

Review: This is a dull movie. I would never recommend it to anyone.

Predicted sentiment: Negative

Probability: 0.77

Review: The cinematography is pretty great in this movie

Predicted sentiment: Positive

Probability: 0.67

Review: The direction was terrible and the story was all over the place

Predicted sentiment: Negative

Probability: 0.63

How it works...

We use NLTK's Naive Bayes classifier for our task here. In the feature extractor function, we basically extract all the unique words. However, the NLTK classifier needs the data to be arranged in the form of a dictionary. Hence, we arranged it in such a way that the NLTK classifier object can ingest it.

Once we divide the data into training and testing datasets, we train the classifier to categorize the sentences into positive and negative. If you look at the top informative words, you can see that we have words such as "outstanding" to indicate positive reviews and words such as "insulting" to indicate negative reviews. This is interesting information because it tells us what words are being used to indicate strong reactions.

Identifying patterns in text using topic modeling

The **topic modeling** refers to the process of identifying hidden patterns in text data. The goal is to uncover some hidden thematic structure in a collection of documents. This will help us in organizing our documents in a better way so that we can use them for analysis. This is an active area of research in NLP. You can learn more about it at <http://www.cs.columbia.edu/~blei/topicmodeling.html>. We will use a library called `gensim` during this recipe. Make sure that you install this before you proceed. The installation steps are given at <https://radimrehurek.com/gensim/install.html>.

How to do it...

1. Create a new Python file and import the following packages:

```
from nltk.tokenize import RegexpTokenizer
from nltk.stem.snowball import SnowballStemmer
from gensim import models, corpora
from nltk.corpus import stopwords
```

2. Define a function to load the input data. We will use the `data_topic_modeling.txt` text file that is already provided to you:

```
# Load input data
def load_data(input_file):
    data = []
    with open(input_file, 'r') as f:
        for line in f.readlines():
            data.append(line[:-1])

    return data
```

3. Let's define a class to preprocess text. This preprocessor takes care of creating the required objects and extracting the relevant features from input text:

```
# Class to preprocess text
class Preprocessor(object):
    # Initialize various operators
    def __init__(self):
        # Create a regular expression tokenizer
        self.tokenizer = RegexpTokenizer(r'\w+')

    # get the list of stop words
    self.stop_words_english = stopwords.words('english')
```

4. We need a list of stop words so that we can exclude them from analysis. These are common words, such as "in", "the", "is", and so on:

```
# get the list of stop words
self.stop_words_english = stopwords.words('english')
```

5. Define a snowball stemmer:


```
# Create a Snowball stemmer
self.stemmer = SnowballStemmer('english')
```

6. Define a processor function that takes care of tokenization, stop word removal, and stemming:

```
# Tokenizing, stop word removal, and stemming
def process(self, input_text):
    # Tokenize the string
    tokens = self.tokenizer.tokenize(input_text.lower())
```

7. Remove the stop words from the text:

```
# Remove the stop words
tokens_stopwords = [x for x in tokens if not x in
self.stop_words_english]
```

8. Perform stemming on tokens:

```
# Perform stemming on the tokens
tokens_stemmed = [self.stemmer.stem(x) for x in
tokens_stopwords]
```

9. Return the processed tokens:

```
return tokens_stemmed
```

10. We are now ready to define the main function. Load the input data from the text file:

```
if __name__ == '__main__':
    # File containing linewise input data
    input_file = 'data_topic_modeling.txt'

    # Load data
    data = load_data(input_file)
```

11. Define an object that is based on the class that we defined:

```
# Create a preprocessor object
preprocessor = Preprocessor()
```

12. We need to process the text in the file, and extract the processed tokens:

```
# Create a list for processed documents
processed_tokens = [preprocessor.process(x) for x in data]
```

13. Create a dictionary, which is based on tokenized documents so that this can be used for topic modeling:

```
# Create a dictionary based on the tokenized documents
dict_tokens = corpora.Dictionary(processed_tokens)
```

14. We need to create a document-term matrix using the processed tokens, as follows:

```
# Create a document-term matrix
corpus = [dict_tokens.doc2bow(text) for text in
processed_tokens]
```

15. Let's say we know that the text can be divided into two topics. We will use a technique called **Latent Dirichlet Allocation (LDA)** for topic modeling. Define the required parameters and initialize the LDA model object:

```
# Generate the LDA model based on the corpus we just created
num_topics = 2
num_words = 4

ldamodel = models.ldamodel.LdaModel(corpus,
    num_topics=num_topics, id2word=dict_tokens,
passes=25)
```

16. Once this identifies the two topics, we can see how it's separating these two topics by looking at the most-contributed words:

```
print "Most contributing words to the topics:"
for item in ldamodel.print_topics(num_topics=num_topics,
num_words=num_words):
    print "\nTopic", item[0], "==>", item[1]
```

17. The full code is in the `topic_modeling.py` file. If you run this code, you will see the following printed on your Terminal:

```
Most contributing words to the topics:

Topic 0 ==> 0.049*need + 0.030*younger + 0.030*talent + 0.030*train
Topic 1 ==> 0.064*need + 0.063*order + 0.038*encrypt + 0.038*understand
```

How it works...

Topic modeling works by identifying the important words of themes in a document. These words tend to determine what the topic is about. We use a regular expression tokenizer because we just want the words without any punctuation or other kinds of tokens. Hence, we use this to extract the tokens. Stop word removal is another important step because this helps us eliminate the noise caused due to words, such as "is" or "the". After this, we need to stem the words to get to their base forms. This entire thing is packaged as a preprocessing block in text analysis tools. This is what we are doing here as well!

We use a technique called Latent Dirichlet Allocation (LDA) to model the topics. LDA basically represents the documents as a mixture of different topics that tend to spit out words. These words are spat out with certain probabilities. The goal is to find these topics! This is a generative model that tries

to find the set of topics that are responsible for the generation of the given set of documents. You can learn more about it at <http://blog.echen.me/2011/08/22/introduction-to-latent-dirichlet-allocation>.

As you can see from the output, we have words such as "talent" and "train" to characterize the sports topic, whereas we have "encrypt" to characterize the cryptography topic. We are working with a really small text file, which is the reason why some words might seem less relevant. Obviously, the accuracy will improve if you work with a larger dataset.