about how many different words might appear in a movie review compared to all the words in the English language (which is what the vocabulary models). Storing all those zeros would be prohibitive, and a waste of memory. To look at the actual content of the sparse matrix, we can convert it to a "dense" NumPy array (that also stores all the 0 entries) using the `toarray` method:[4]

**In[11]:**

```
print("Dense representation of bag_of_words:\n{}".format(
    bag_of_words.toarray()))
```

**Out[11]:**

```
Dense representation of bag_of_words:
[[0 0 1 1 1 0 1 0 0 1 1 0 1]
 [1 1 0 1 0 1 0 1 1 1 0 1 1]]
```

We can see that the word counts for each word are either 0 or 1; neither of the two strings in `bards_words` contains a word twice. Let's take a look at how to read these feature vectors. The first string (`"The fool doth think he is wise,"`) is represented as the first row in, and it contains the first word in the vocabulary, `"be"`, zero times. It also contains the second word in the vocabulary, `"but"`, zero times. It contains the third word, `"doth"`, once, and so on. Looking at both rows, we can see that the fourth word, `"fool"`, the tenth word, `"the"`, and the thirteenth word, `"wise"`, appear in both strings.

## Bag-of-Words for Movie Reviews

Now that we've gone through the bag-of-words process in detail, let's apply it to our task of sentiment analysis for movie reviews. Earlier, we loaded our training and test data from the IMDb reviews into lists of strings (`text_train` and `text_test`), which we will now process:

**In[12]:**

```
vect = CountVectorizer().fit(text_train)
X_train = vect.transform(text_train)
print("X_train:\n{}".format(repr(X_train)))
```

**Out[12]:**

```
X_train:
<25000x74849 sparse matrix of type '<class 'numpy.int64'>'
    with 3431196 stored elements in Compressed Sparse Row format>
```

4 This is possible because we are using a small toy dataset that contains only 13 words. For any real dataset, this would result in a `MemoryError`.

The shape of `X_train`, the bag-of-words representation of the training data, is 25,000×74,849, indicating that the vocabulary contains 74,849 entries. Again, the data is stored as a SciPy sparse matrix. Let's look at the vocabulary in a bit more detail. Another way to access the vocabulary is using the `get_feature_name` method of the vectorizer, which returns a convenient list where each entry corresponds to one feature:

**In[13]:**

```
feature_names = vect.get_feature_names()
print("Number of features: {}".format(len(feature_names)))
print("First 20 features:\n{}".format(feature_names[:20]))
print("Features 20010 to 20030:\n{}".format(feature_names[20010:20030]))
print("Every 2000th feature:\n{}".format(feature_names[::2000]))
```

**Out[13]:**

```
Number of features: 74849
First 20 features:
['00', '000', '0000000000001', '00001', '00015', '000s', '001', '003830',
 '006', '007', '0079', '0080', '0083', '0093638', '00am', '00pm', '00s',
 '01', '01pm', '02']
Features 20010 to 20030:
['dratted', 'draub', 'draught', 'draughts', 'draughtswoman', 'draw', 'drawback',
 'drawbacks', 'drawer', 'drawers', 'drawing', 'drawings', 'drawl',
 'drawled', 'drawling', 'drawn', 'draws', 'draza', 'dre', 'drea']
Every 2000th feature:
['00', 'aesir', 'aquarian', 'barking', 'blustering', 'bête', 'chicanery',
 'condensing', 'cunning', 'detox', 'draper', 'enshrined', 'favorit', 'freezer',
 'goldman', 'hasan', 'huitieme', 'intelligible', 'kantrowitz', 'lawful',
 'maars', 'megalunged', 'mostey', 'norrland', 'padilla', 'pincher',
 'promisingly', 'receptionist', 'rivals', 'schnaas', 'shunning', 'sparse',
 'subset', 'temptations', 'treatises', 'unproven', 'walkman', 'xylophonist']
```

As you can see, possibly a bit surprisingly, the first 10 entries in the vocabulary are all numbers. All these numbers appear somewhere in the reviews, and are therefore extracted as words. Most of these numbers don't have any immediate semantic meaning—apart from `"007"`, which in the particular context of movies is likely to refer to the James Bond character.[5] Weeding out the meaningful from the nonmeaningful "words" is sometimes tricky. Looking further along in the vocabulary, we find a collection of English words starting with "dra". You might notice that for `"draught"`, `"drawback"`, and `"drawer"` both the singular and plural forms are contained in the vocabulary as distinct words. These words have very closely related semantic meanings, and counting them as different words, corresponding to different features, might not be ideal.

5 A quick analysis of the data confirms that this is indeed the case. Try confirming it yourself.

Before we try to improve our feature extraction, let's obtain a quantitative measure of performance by actually building a classifier. We have the training labels stored in `y_train` and the bag-of-words representation of the training data in `X_train`, so we can train a classifier on this data. For high-dimensional, sparse data like this, linear models like `LogisticRegression` often work best.

Let's start by evaluating `LogisticRegresssion` using cross-validation:[6]

**In[14]:**

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
scores = cross_val_score(LogisticRegression(), X_train, y_train, cv=5)
print("Mean cross-validation accuracy: {:.2f}".format(np.mean(scores)))
```

**Out[14]:**

```
Mean cross-validation accuracy: 0.88
```

We obtain a mean cross-validation score of 88%, which indicates reasonable performance for a balanced binary classification task. We know that `LogisticRegression` has a regularization parameter, `C`, which we can tune via cross-validation:

**In[15]:**

```
from sklearn.model_selection import GridSearchCV
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
print("Best parameters: ", grid.best_params_)
```

**Out[15]:**

```
Best cross-validation score: 0.89
Best parameters:  {'C': 0.1}
```

We obtain a cross-validation score of 89% using `C=0.1`. We can now assess the generalization performance of this parameter setting on the test set:

**In[16]:**

```
X_test = vect.transform(text_test)
print("{:.2f}".format(grid.score(X_test, y_test)))
```

**Out[16]:**

```
0.88
```

6 The attentive reader might notice that we violate our lesson from Chapter 6 on cross-validation with preprocessing here. Using the default settings of `CountVectorizer`, it actually does not collect any statistics, so our results are valid. Using `Pipeline` from the start would be a better choice for applications, but we defer it for ease of exposure.

Now, let's see if we can improve the extraction of words. The `CountVectorizer` extracts tokens using a regular expression. By default, the regular expression that is used is `"\b\w\w+\b"`. If you are not familiar with regular expressions, this means it finds all sequences of characters that consist of at least two letters or numbers (`\w`) and that are separated by word boundaries (`\b`). It does not find single-letter words, and it splits up contractions like "doesn't" or "bit.ly", but it matches "h8ter" as a single word. The `CountVectorizer` then converts all words to lowercase characters, so that "soon", "Soon", and "sOon" all correspond to the same token (and therefore feature). This simple mechanism works quite well in practice, but as we saw earlier, we get many uninformative features (like the numbers). One way to cut back on these is to only use tokens that appear in at least two documents (or at least five documents, and so on). A token that appears only in a single document is unlikely to appear in the test set and is therefore not helpful. We can set the minimum number of documents a token needs to appear in with the `min_df` parameter:

**In[17]:**

```
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("X_train with min_df: {}".format(repr(X_train)))
```

**Out[17]:**

```
X_train with min_df: <25000x27271 sparse matrix of type '<class 'numpy.int64'>'
    with 3354014 stored elements in Compressed Sparse Row format>
```

By requiring at least five appearances of each token, we can bring down the number of features to 27,271, as seen in the preceding output—only about a third of the original features. Let's look at some tokens again:

**In[18]:**

```
feature_names = vect.get_feature_names()

print("First 50 features:\n{}".format(feature_names[:50]))
print("Features 20010 to 20030:\n{}".format(feature_names[20010:20030]))
print("Every 700th feature:\n{}".format(feature_names[::700]))
```

**Out[18]:**

```
First 50 features:
['00', '000', '007', '00s', '01', '02', '03', '04', '05', '06', '07', '08',
 '09', '10', '100', '1000', '100th', '101', '102', '103', '104', '105', '107',
 '108', '10s', '10th', '11', '110', '112', '116', '117', '11th', '12', '120',
 '12th', '13', '135', '13th', '14', '140', '14th', '15', '150', '15th', '16',
 '160', '1600', '16mm', '16s', '16th']
Features 20010 to 20030:
['repentance', 'repercussions', 'repertoire', 'repetition', 'repetitions',
 'repetitious', 'repetitive', 'rephrase', 'replace', 'replaced', 'replacement',
 'replaces', 'replacing', 'replay', 'replayable', 'replayed', 'replaying',
 'replays', 'replete', 'replica']
```

```
Every 700th feature:
['00', 'affections', 'appropriately', 'barbra', 'blurbs', 'butchered',
 'cheese', 'commitment', 'courts', 'deconstructed', 'disgraceful', 'dvds',
 'eschews', 'fell', 'freezer', 'goriest', 'hauser', 'hungary', 'insinuate',
 'juggle', 'leering', 'maelstrom', 'messiah', 'music', 'occasional', 'parking',
 'pleasantville', 'pronunciation', 'recipient', 'reviews', 'sas', 'shea',
 'sneers', 'steiger', 'swastika', 'thrusting', 'tvs', 'vampyre', 'westerns']
```

There are clearly many fewer numbers, and some of the more obscure words or misspellings seem to have vanished. Let's see how well our model performs by doing a grid search again:

**In[19]:**

```
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
```

**Out[19]:**

```
Best cross-validation score: 0.89
```

The best validation accuracy of the grid search is still 89%, unchanged from before. We didn't improve our model, but having fewer features to deal with speeds up processing and throwing away useless features might make the model more interpretable.

If the `transform` method of `CountVectorizer` is called on a document that contains words that were not contained in the training data, these words will be ignored as they are not part of the dictionary. This is not really an issue for classification, as it's not possible to learn anything about words that are not in the training data. For some applications, like spam detection, it might be helpful to add a feature that encodes how many so-called "out of vocabulary" words there are in a particular document, though. For this to work, you need to set `min_df`; otherwise, this feature will never be active during training.

## Stopwords

Another way that we can get rid of uninformative words is by discarding words that are too frequent to be informative. There are two main approaches: using a language-specific list of stopwords, or discarding words that appear too frequently. `scikit-learn` has a built-in list of English stopwords in the `feature_extraction.text` module: