large impact on our machine learning models, it is better to clean the data and remove this formatting before we proceed:

In[4]:

```
text_train = [doc.replace(b"<br />", b" ") for doc in text_train]
```

The type of the entries of `text_train` will depend on your Python version. In Python 3, they will be of type `bytes` which represents a binary encoding of the string data. In Python 2, `text_train` contains strings. We won't go into the details of the different string types in Python here, but we recommend that you read the Python 2 and/or Python 3 documentation regarding strings and Unicode.

The dataset was collected such that the positive class and the negative class balanced, so that there are as many positive as negative strings:

In[5]:

```
print("Samples per class (training): {}".format(np.bincount(y_train)))
```

Out[5]:

```
Samples per class (training): [12500 12500]
```

We load the test dataset in the same manner:

In[6]:

```
reviews_test = load_files("data/aclImdb/test/")
text_test, y_test = reviews_test.data, reviews_test.target
print("Number of documents in test data: {}".format(len(text_test)))
print("Samples per class (test): {}".format(np.bincount(y_test)))
text_test = [doc.replace(b"<br />", b" ") for doc in text_test]
```

Out[6]:

```
Number of documents in test data: 25000
Samples per class (test): [12500 12500]
```

The task we want to solve is as follows: given a review, we want to assign the label "positive" or "negative" based on the text content of the review. This is a standard binary classification task. However, the text data is not in a format that a machine learning model can handle. We need to convert the string representation of the text into a numeric representation that we can apply our machine learning algorithms to.

Representing Text Data as a Bag of Words

One of the most simple but effective and commonly used ways to represent text for machine learning is using the *bag-of-words* representation. When using this representation, we discard most of the structure of the input text, like chapters, paragraphs, sentences, and formatting, and only count *how often each word appears in each text* in

the corpus. Discarding the structure and counting only word occurrences leads to the mental image of representing text as a "bag."

Computing the bag-of-words representation for a corpus of documents consists of the following three steps:

1. *Tokenization*. Split each document into the words that appear in it (called *tokens*), for example by splitting them on whitespace and punctuation.
2. *Vocabulary building*. Collect a vocabulary of all words that appear in any of the documents, and number them (say, in alphabetical order).
3. *Encoding*. For each document, count how often each of the words in the vocabulary appear in this document.

There are some subtleties involved in step 1 and step 2, which we will discuss in more detail later in this chapter. For now, let's look at how we can apply the bag-of-words processing using `scikit-learn`. Figure 7-1 illustrates the process on the string `"This is how you get ants."`. The output is one vector of word counts for each document. For each word in the vocabulary, we have a count of how often it appears in each document. That means our numeric representation has one feature for each unique word in the whole dataset. Note how the order of the words in the original string is completely irrelevant to the bag-of-words feature representation.

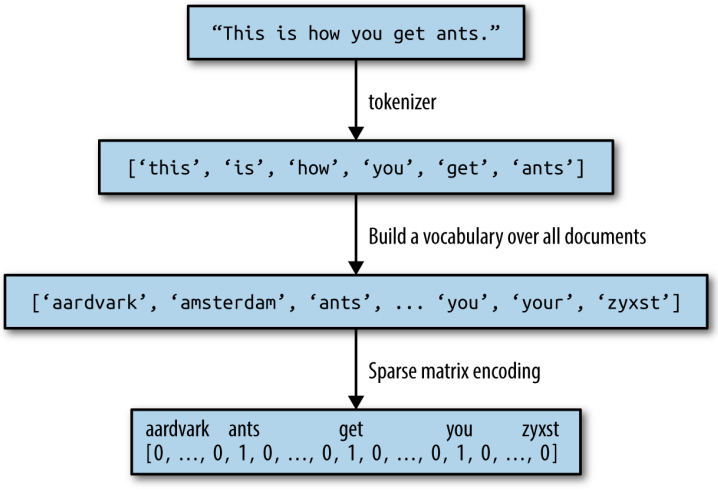

Figure 7-1. Bag-of-words processing

Applying Bag-of-Words to a Toy Dataset

The bag-of-words representation is implemented in `CountVectorizer`, which is a transformer. Let's first apply it to a toy dataset, consisting of two samples, to see it working:

In[7]:

```
bards_words =["The fool doth think he is wise,",
              "but the wise man knows himself to be a fool"]
```

We import and instantiate the `CountVectorizer` and `fit` it to our toy data as follows:

In[8]:

```
from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer()
vect.fit(bards_words)
```

Fitting the `CountVectorizer` consists of the tokenization of the training data and building of the vocabulary, which we can access as the `vocabulary_` attribute:

In[9]:

```
print("Vocabulary size: {}".format(len(vect.vocabulary_)))
print("Vocabulary content:\n {}".format(vect.vocabulary_))
```

Out[9]:

```
Vocabulary size: 13
Vocabulary content:
 {'the': 9, 'himself': 5, 'wise': 12, 'he': 4, 'doth': 2, 'to': 11, 'knows': 7,
  'man': 8, 'fool': 3, 'is': 6, 'be': 0, 'think': 10, 'but': 1}
```

The vocabulary consists of 13 words, from `"be"` to `"wise"`.

To create the bag-of-words representation for the training data, we call the `transform` method:

In[10]:

```
bag_of_words = vect.transform(bards_words)
print("bag_of_words: {}".format(repr(bag_of_words)))
```

Out[10]:

```
bag_of_words: <2x13 sparse matrix of type '<class 'numpy.int64'>'
    with 16 stored elements in Compressed Sparse Row format>
```

The bag-of-words representation is stored in a SciPy sparse matrix that only stores the entries that are nonzero (see Chapter 1). The matrix is of shape 2×13, with one row for each of the two data points and one feature for each of the words in the vocabulary. A sparse matrix is used as most documents only contain a small subset of the words in the vocabulary, meaning most entries in the feature array are 0. Think