CHAPTER 7

Working with Text Data

In Chapter 4, we talked about two kinds of features that can represent properties of the data: continuous features that describe a quantity, and categorical features that are items from a fixed list. There is a third kind of feature that can be found in many applications, which is text. For example, if we want to classify an email message as either a legitimate email or spam, the content of the email will certainly contain important information for this classification task. Or maybe we want to learn about the opinion of a politician on the topic of immigration. Here, that individual's speeches or tweets might provide useful information. In customer service, we often want to find out if a message is a complaint or an inquiry. We can use the subject line and content of a message to automatically determine the customer's intent, which allows us to send the message to the appropriate department, or even send a fully automatic reply.

Text data is usually represented as strings, made up of characters. In any of the examples just given, the length of the text data will vary. This feature is clearly very different from the numeric features that we've discussed so far, and we will need to process the data before we can apply our machine learning algorithms to it.

Types of Data Represented as Strings

Before we dive into the processing steps that go into representing text data for machine learning, we want to briefly discuss different kinds of text data that you might encounter. Text is usually just a string in your dataset, but not all string features should be treated as text. A string feature can sometimes represent categorical variables, as we discussed in Chapter 5. There is no way to know how to treat a string feature before looking at the data.

There are four kinds of string data you might see:

- Categorical data
- Free strings that can be semantically mapped to categories
- Structured string data
- Text data

Categorical data is data that comes from a fixed list. Say you collect data via a survey where you ask people their favorite color, with a drop-down menu that allows them to select from "red," "green," "blue," "yellow," "black," "white," "purple," and "pink." This will result in a dataset with exactly eight different possible values, which clearly encode a categorical variable. You can check whether this is the case for your data by eyeballing it (if you see very many different strings it is unlikely that this is a categorical variable) and confirm it by computing the unique values over the dataset, and possibly a histogram over how often each appears. You also might want to check whether each variable actually corresponds to a category that makes sense for your application. Maybe halfway through the existence of your survey, someone found that "black" was misspelled as "blak" and subsequently fixed the survey. As a result, your dataset contains both "blak" and "black," which correspond to the same semantic meaning and should be consolidated.

Now imagine instead of providing a drop-down menu, you provide a text field for the users to provide their own favorite colors. Many people might respond with a color name like "black" or "blue." Others might make typographical errors, use different spellings like "gray" and "grey," or use more evocative and specific names like "midnight blue." You will also have some very strange entries. Some good examples come from the xkcd Color Survey, where people had to name colors and came up with names like "velociraptor cloaka" and "my dentist's office orange. I still remember his dandruff slowly wafting into my gaping yaw," which are hard to map to colors automatically (or at all). The responses you can obtain from a text field belong to the second category in the list, *free strings that can be semantically mapped to categories*. It will probably be best to encode this data as a categorical variable, where you can select the categories either by using the most common entries, or by defining categories that will capture responses in a way that makes sense for your application. You might then have some categories for standard colors, maybe a category "multicolored" for people that gave answers like "green and red stripes," and an "other" category for things that cannot be encoded otherwise. This kind of preprocessing of strings can take a lot of manual effort and is not easily automated. If you are in a position where you can influence data collection, we highly recommend avoiding manually entered values for concepts that are better captured using categorical variables.

Often, manually entered values do not correspond to fixed categories, but still have some underlying *structure*, like addresses, names of places or people, dates, telephone

numbers, or other identifiers. These kinds of strings are often very hard to parse, and their treatment is highly dependent on context and domain. A systematic treatment of these cases is beyond the scope of this book.

The final category of string data is freeform *text data* that consists of phrases or sentences. Examples include tweets, chat logs, and hotel reviews, as well as the collected works of Shakespeare, the content of Wikipedia, or the Project Gutenberg collection of 50,000 ebooks. All of these collections contain information mostly as sentences composed of words.[1] For simplicity's sake, let's assume all our documents are in one language, English.[2] In the context of text analysis, the dataset is often called the *corpus*, and each data point, represented as a single text, is called a *document*. These terms come from the *information retrieval* (IR) and *natural language processing* (NLP) community, which both deal mostly in text data.

Example Application: Sentiment Analysis of Movie Reviews

As a running example in this chapter, we will use a dataset of movie reviews from the IMDb (Internet Movie Database) website collected by Stanford researcher Andrew Maas.[3] This dataset contains the text of the reviews, together with a label that indicates whether a review is "positive" or "negative." The IMDb website itself contains ratings from 1 to 10. To simplify the modeling, this annotation is summarized as a two-class classification dataset where reviews with a score of 6 or higher are labeled as positive, and the rest as negative. We will leave the question of whether this is a good representation of the data open, and simply use the data as provided by Andrew Maas.

After unpacking the data, the dataset is provided as text files in two separate folders, one for the training data and one for the test data. Each of these in turn has two subfolders, one called *pos* and one called *neg*:

1 Arguably, the content of websites linked to in tweets contains more information than the text of the tweets themselves.

2 Most of what we will talk about in the rest of the chapter also applies to other languages that use the Roman alphabet, and partially to other languages with word boundary delimiters. Chinese, for example, does not delimit word boundaries, and has other challenges that make applying the techniques in this chapter difficult.

3 The dataset is available at *http://ai.stanford.edu/~amaas/data/sentiment/*.

In[2]:

```
!tree -L 2 data/aclImdb
```

Out[2]:

```
data/aclImdb
├── test
│   ├── neg
│   └── pos
└── train
    ├── neg
    └── pos

6 directories, 0 files
```

The *pos* folder contains all the positive reviews, each as a separate text file, and similarly for the *neg* folder. There is a helper function in `scikit-learn` to load files stored in such a folder structure, where each subfolder corresponds to a label, called `load_files`. We apply the `load_files` function first to the training data:

In[3]:

```
from sklearn.datasets import load_files

reviews_train = load_files("data/aclImdb/train/")
# load_files returns a bunch, containing training texts and training labels
text_train, y_train = reviews_train.data, reviews_train.target
print("type of text_train: {}".format(type(text_train)))
print("length of text_train: {}".format(len(text_train)))
print("text_train[1]:\n{}".format(text_train[1]))
```

Out[3]:

```
type of text_train:  <class 'list'>
length of text_train:  25000
text_train[1]:
b'Words can\'t describe how bad this movie is. I can\'t explain it by writing
  only. You have too see it for yourself to get at grip of how horrible a movie
  really can be. Not that I recommend you to do that. There are so many
  clich\xc3\xa9s, mistakes (and all other negative things you can imagine) here
  that will just make you cry. To start with the technical first, there are a
  LOT of mistakes regarding the airplane. I won\'t list them here, but just
  mention the coloring of the plane. They didn\'t even manage to show an
  airliner in the colors of a fictional airline, but instead used a 747
  painted in the original Boeing livery. Very bad. The plot is stupid and has
  been done many times before, only much, much better. There are so many
  ridiculous moments here that i lost count of it really early. Also, I was on
  the bad guys\' side all the time in the movie, because the good guys were so
  stupid. "Executive Decision" should without a doubt be you\'re choice over
  this one, even the "Turbulence"-movies are better. In fact, every other
  movie in the world is better than this one.'
```

You can see that `text_train` is a list of length 25,000, where each entry is a string containing a review. We printed the review with index 1. You can also see that the review contains some HTML line breaks (`<br />`). While these are unlikely to have a

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.

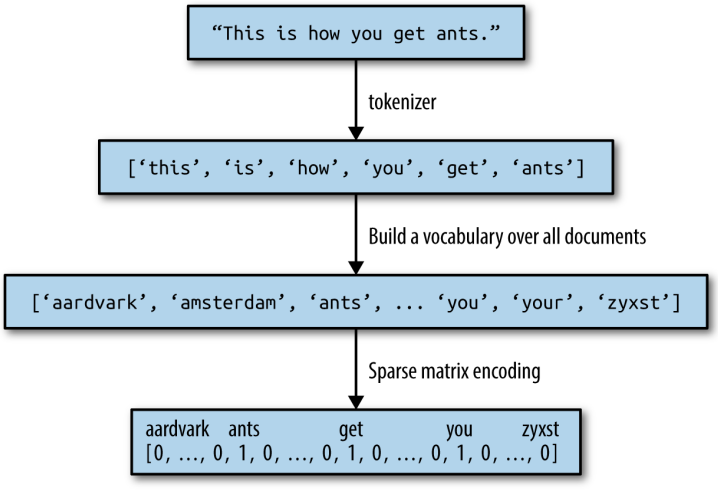

Figure 7-1. Bag-of-words processing

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

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:

In[20]:

```
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
print("Number of stop words: {}".format(len(ENGLISH_STOP_WORDS)))
print("Every 10th stopword:\n{}".format(list(ENGLISH_STOP_WORDS)[::10]))
```

Out[20]:

```
Number of stop words: 318
Every 10th stopword:
['above', 'elsewhere', 'into', 'well', 'rather', 'fifteen', 'had', 'enough',
 'herein', 'should', 'third', 'although', 'more', 'this', 'none', 'seemed',
 'nobody', 'seems', 'he', 'also', 'fill', 'anyone', 'anything', 'me', 'the',
 'yet', 'go', 'seeming', 'front', 'beforehand', 'forty', 'i']
```

Clearly, removing the stopwords in the list can only decrease the number of features by the length of the list—here, 318—but it might lead to an improvement in performance. Let's give it a try:

In[21]:

```
# Specifying stop_words="english" uses the built-in list.
# We could also augment it and pass our own.
vect = CountVectorizer(min_df=5, stop_words="english").fit(text_train)
X_train = vect.transform(text_train)
print("X_train with stop words:\n{}".format(repr(X_train)))
```

Out[21]:

```
X_train with stop words:
<25000x26966 sparse matrix of type '<class 'numpy.int64'>'
    with 2149958 stored elements in Compressed Sparse Row format>
```

There are now 305 (27,271–26,966) fewer features in the dataset, which means that most, but not all, of the stopwords appeared. Let's run the grid search again:

In[22]:

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

Out[22]:

```
Best cross-validation score: 0.88
```

The grid search performance decreased slightly using the stopwords—not enough to worry about, but given that excluding 305 features out of over 27,000 is unlikely to change performance or interpretability a lot, it doesn't seem worth using this list. Fixed lists are mostly helpful for small datasets, which might not contain enough information for the model to determine which words are stopwords from the data itself. As an exercise, you can try out the other approach, discarding frequently

appearing words, by setting the `max_df` option of `CountVectorizer` and see how it influences the number of features and the performance.

Rescaling the Data with tf–idf

Instead of dropping features that are deemed unimportant, another approach is to rescale features by how informative we expect them to be. One of the most common ways to do this is using the *term frequency–inverse document frequency* (tf–idf) method. The intuition of this method is to give high weight to any term that appears often in a particular document, but not in many documents in the corpus. If a word appears often in a particular document, but not in very many documents, it is likely to be very descriptive of the content of that document. `scikit-learn` implements the tf–idf method in two classes: `TfidfTransformer`, which takes in the sparse matrix output produced by `CountVectorizer` and transforms it, and `TfidfVectorizer`, which takes in the text data and does both the bag-of-words feature extraction and the tf–idf transformation. There are several variants of the tf–idf rescaling scheme, which you can read about on Wikipedia. The tf–idf score for word w in document d as implemented in both the `TfidfTransformer` and `TfidfVectorizer` classes is given by:[7]

$$\text{tfidf}(w, d) = \text{tf} \log\left(\frac{N+1}{N_w+1}\right) + 1$$

where N is the number of documents in the training set, N_w is the number of documents in the training set that the word w appears in, and *tf* (the term frequency) is the number of times that the word w appears in the query document d (the document you want to transform or encode). Both classes also apply L2 normalization after computing the tf–idf representation; in other words, they rescale the representation of each document to have Euclidean norm 1. Rescaling in this way means that the length of a document (the number of words) does not change the vectorized representation.

Because tf–idf actually makes use of the statistical properties of the training data, we will use a pipeline, as described in Chapter 6, to ensure the results of our grid search are valid. This leads to the following code:

7 We provide this formula here mostly for completeness; you don't need to remember it to use the tf–idf encoding.

In[23]:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(TfidfVectorizer(min_df=5, norm=None),
                     LogisticRegression())
param_grid = {'logisticregression__C': [0.001, 0.01, 0.1, 1, 10]}

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

Out[23]:

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

As you can see, there is some improvement when using tf–idf instead of just word counts. We can also inspect which words tf–idf found most important. Keep in mind that the tf–idf scaling is meant to find words that distinguish documents, but it is a purely unsupervised technique. So, "important" here does not necessarily relate to the "positive review" and "negative review" labels we are interested in. First, we extract the `TfidfVectorizer` from the pipeline:

In[24]:

```
vectorizer = grid.best_estimator_.named_steps["tfidfvectorizer"]
# transform the training dataset
X_train = vectorizer.transform(text_train)
# find maximum value for each of the features over the dataset
max_value = X_train.max(axis=0).toarray().ravel()
sorted_by_tfidf = max_value.argsort()
# get feature names
feature_names = np.array(vectorizer.get_feature_names())

print("Features with lowest tfidf:\n{}".format(
    feature_names[sorted_by_tfidf[:20]]))

print("Features with highest tfidf: \n{}".format(
    feature_names[sorted_by_tfidf[-20:]]))
```

Out[24]:

```
Features with lowest tfidf:
['poignant' 'disagree' 'instantly' 'importantly' 'lacked' 'occurred'
 'currently' 'altogether' 'nearby' 'undoubtedly' 'directs' 'fond' 'stinker'
 'avoided' 'emphasis' 'commented' 'disappoint' 'realizing' 'downhill'
 'inane']
Features with highest tfidf:
['coop' 'homer' 'dillinger' 'hackenstein' 'gadget' 'taker' 'macarthur'
 'vargas' 'jesse' 'basket' 'dominick' 'the' 'victor' 'bridget' 'victoria'
 'khouri' 'zizek' 'rob' 'timon' 'titanic']
```

Features with low tf–idf are those that either are very commonly used across documents or are only used sparingly, and only in very long documents. Interestingly, many of the high-tf–idf features actually identify certain shows or movies. These terms only appear in reviews for this particular show or franchise, but tend to appear very often in these particular reviews. This is very clear, for example, for `"pokemon"`, `"smallville"`, and `"doodlebops"`, but `"scanners"` here actually also refers to a movie title. These words are unlikely to help us in our sentiment classification task (unless maybe some franchises are universally reviewed positively or negatively) but certainly contain a lot of specific information about the reviews.

We can also find the words that have low inverse document frequency—that is, those that appear frequently and are therefore deemed less important. The inverse document frequency values found on the training set are stored in the `idf_` attribute:

`In[25]:`

```
sorted_by_idf = np.argsort(vectorizer.idf_)
print("Features with lowest idf:\n{}".format(
    feature_names[sorted_by_idf[:100]]))
```

`Out[25]:`

```
Features with lowest idf:
['the' 'and' 'of' 'to' 'this' 'is' 'it' 'in' 'that' 'but' 'for' 'with'
 'was' 'as' 'on' 'movie' 'not' 'have' 'one' 'be' 'film' 'are' 'you' 'all'
 'at' 'an' 'by' 'so' 'from' 'like' 'who' 'they' 'there' 'if' 'his' 'out'
 'just' 'about' 'he' 'or' 'has' 'what' 'some' 'good' 'can' 'more' 'when'
 'time' 'up' 'very' 'even' 'only' 'no' 'would' 'my' 'see' 'really' 'story'
 'which' 'well' 'had' 'me' 'than' 'much' 'their' 'get' 'were' 'other'
 'been' 'do' 'most' 'don' 'her' 'also' 'into' 'first' 'made' 'how' 'great'
 'because' 'will' 'people' 'make' 'way' 'could' 'we' 'bad' 'after' 'any'
 'too' 'then' 'them' 'she' 'watch' 'think' 'acting' 'movies' 'seen' 'its'
 'him']
```

As expected, these are mostly English stopwords like `"the"` and `"no"`. But some are clearly domain-specific to the movie reviews, like `"movie"`, `"film"`, `"time"`, `"story"`, and so on. Interestingly, `"good"`, `"great"`, and `"bad"` are also among the most frequent and therefore "least relevant" words according to the tf–idf measure, even though we might expect these to be very important for our sentiment analysis task.

Investigating Model Coefficients

Finally, let's look in a bit more detail into what our logistic regression model actually learned from the data. Because there are so many features—27,271 after removing the infrequent ones—we clearly cannot look at all of the coefficients at the same time. However, we can look at the largest coefficients, and see which words these correspond to. We will use the last model that we trained, based on the tf–idf features.

The following bar chart (Figure 7-2) shows the 25 largest and 25 smallest coefficients of the logistic regression model, with the bars showing the size of each coefficient:

In[26]:

```
mglearn.tools.visualize_coefficients(
    grid.best_estimator_.named_steps["logisticregression"].coef_,
    feature_names, n_top_features=40)
```

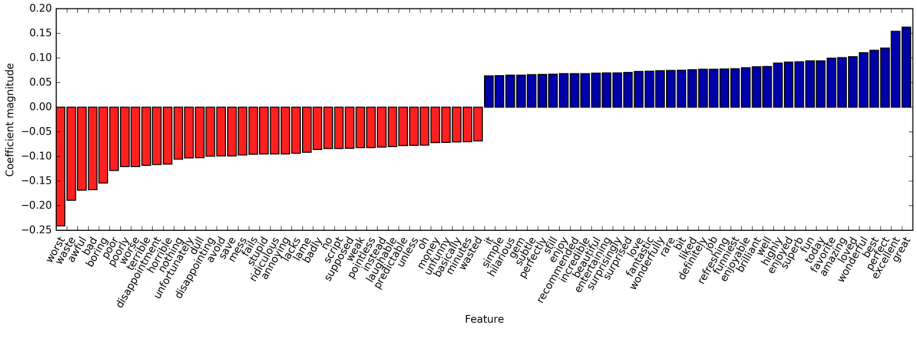

Figure 7-2. Largest and smallest coefficients of logistic regression trained on tf-idf features

The negative coefficients on the left belong to words that according to the model are indicative of negative reviews, while the positive coefficients on the right belong to words that according to the model indicate positive reviews. Most of the terms are quite intuitive, like `"worst"`, `"waste"`, `"disappointment"`, and `"laughable"` indicating bad movie reviews, while `"excellent"`, `"wonderful"`, `"enjoyable"`, and `"refreshing"` indicate positive movie reviews. Some words are slightly less clear, like `"bit"`, `"job"`, and `"today"`, but these might be part of phrases like "good job" or "best today."

Bag-of-Words with More Than One Word (n-Grams)

One of the main disadvantages of using a bag-of-words representation is that word order is completely discarded. Therefore, the two strings "it's bad, not good at all" and "it's good, not bad at all" have exactly the same representation, even though the meanings are inverted. Putting "not" in front of a word is only one example (if an extreme one) of how context matters. Fortunately, there is a way of capturing context when using a bag-of-words representation, by not only considering the counts of single tokens, but also the counts of pairs or triplets of tokens that appear next to each other. Pairs of tokens are known as *bigrams*, triplets of tokens are known as *trigrams*, and more generally sequences of tokens are known as *n-grams*. We can change the range of tokens that are considered as features by changing the `ngram_range` parameter of `CountVectorizer` or `TfidfVectorizer`. The `ngram_range` parameter is a tuple, con-

sisting of the minimum length and the maximum length of the sequences of tokens that are considered. Here is an example on the toy data we used earlier:

In[27]:

```
print("bards_words:\n{}".format(bards_words))
```

Out[27]:

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

The default is to create one feature per sequence of tokens that is at least one token long and at most one token long, or in other words exactly one token long (single tokens are also called *unigrams*):

In[28]:

```
cv = CountVectorizer(ngram_range=(1, 1)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

Out[28]:

```
Vocabulary size: 13
Vocabulary:
['be', 'but', 'doth', 'fool', 'he', 'himself', 'is', 'knows', 'man', 'the',
 'think', 'to', 'wise']
```

To look only at bigrams—that is, only at sequences of two tokens following each other—we can set `ngram_range` to `(2, 2)`:

In[29]:

```
cv = CountVectorizer(ngram_range=(2, 2)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

Out[29]:

```
Vocabulary size: 14
Vocabulary:
['be fool', 'but the', 'doth think', 'fool doth', 'he is', 'himself to',
 'is wise', 'knows himself', 'man knows', 'the fool', 'the wise',
 'think he', 'to be', 'wise man']
```

Using longer sequences of tokens usually results in many more features, and in more specific features. There is no common bigram between the two phrases in `bard_words`:

In[30]:

```
print("Transformed data (dense):\n{}".format(cv.transform(bards_words).toarray()))
```

Out[30]:

```
Transformed data (dense):
[[0 0 1 1 1 0 1 0 0 1 0 1 0 0]
 [1 1 0 0 0 1 0 1 1 0 1 0 1 1]]
```

For most applications, the minimum number of tokens should be one, as single words often capture a lot of meaning. Adding bigrams helps in most cases. Adding longer sequences—up to 5-grams—might help too, but this will lead to an explosion of the number of features and might lead to overfitting, as there will be many very specific features. In principle, the number of bigrams could be the number of unigrams squared and the number of trigrams could be the number of unigrams to the power of three, leading to very large feature spaces. In practice, the number of higher *n*-grams that actually appear in the data is much smaller, because of the structure of the (English) language, though it is still large.

Here is what using unigrams, bigrams, and trigrams on `bards_words` looks like:

In[31]:

```
cv = CountVectorizer(ngram_range=(1, 3)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

Out[31]:

```
Vocabulary size: 39
Vocabulary:
['be', 'be fool', 'but', 'but the', 'but the wise', 'doth', 'doth think',
 'doth think he', 'fool', 'fool doth', 'fool doth think', 'he', 'he is',
 'he is wise', 'himself', 'himself to', 'himself to be', 'is', 'is wise',
 'knows', 'knows himself', 'knows himself to', 'man', 'man knows',
 'man knows himself', 'the', 'the fool', 'the fool doth', 'the wise',
 'the wise man', 'think', 'think he', 'think he is', 'to', 'to be',
 'to be fool', 'wise', 'wise man', 'wise man knows']
```

Let's try out the `TfidfVectorizer` on the IMDb movie review data and find the best setting of *n*-gram range using a grid search:

In[32]:

```
pipe = make_pipeline(TfidfVectorizer(min_df=5), LogisticRegression())
# running the grid search takes a long time because of the
# relatively large grid and the inclusion of trigrams
param_grid = {"logisticregression__C": [0.001, 0.01, 0.1, 1, 10, 100],
              "tfidfvectorizer__ngram_range": [(1, 1), (1, 2), (1, 3)]}

grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
print("Best parameters:\n{}".format(grid.best_params_))
```

Out[32]:

```
Best cross-validation score: 0.91
Best parameters:
{'tfidfvectorizer__ngram_range': (1, 3), 'logisticregression__C': 100}
```

As you can see from the results, we improved performance by a bit more than a percent by adding bigram and trigram features. We can visualize the cross-validation accuracy as a function of the `ngram_range` and `C` parameter as a heat map, as we did in Chapter 5 (see Figure 7-3):

In[33]:

```
# extract scores from grid_search
scores = grid.cv_results_['mean_test_score'].reshape(-1, 3).T
# visualize heat map
heatmap = mglearn.tools.heatmap(
    scores, xlabel="C", ylabel="ngram_range", cmap="viridis", fmt="%.3f",
    xticklabels=param_grid['logisticregression__C'],
    yticklabels=param_grid['tfidfvectorizer__ngram_range'])
plt.colorbar(heatmap)
```

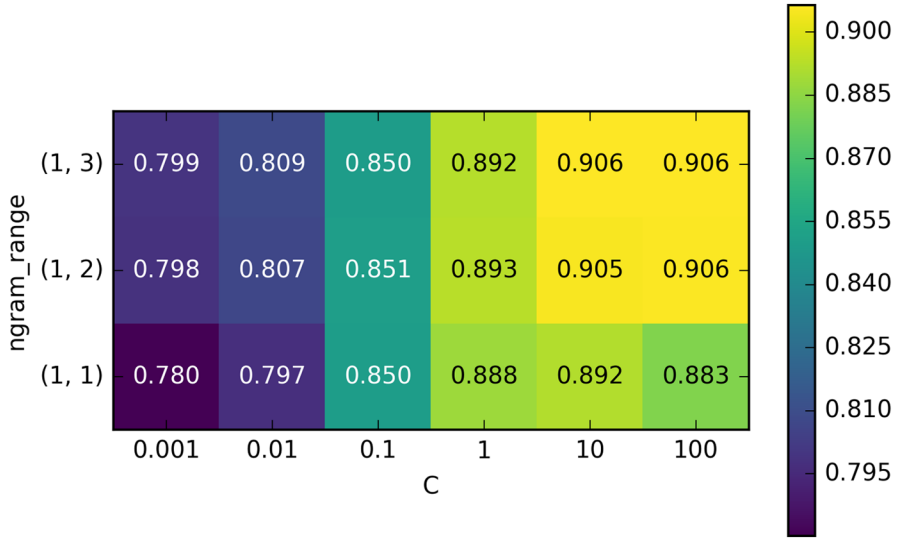

Figure 7-3. Heat map visualization of mean cross-validation accuracy as a function of the parameters ngram_range and C

From the heat map we can see that using bigrams increases performance quite a bit, while adding trigrams only provides a very small benefit in terms of accuracy. To understand better how the model improved, we can visualize the important coeffi-

cient for the best model, which includes unigrams, bigrams, and trigrams (see Figure 7-4):

In[34]:

```
# extract feature names and coefficients
vect = grid.best_estimator_.named_steps['tfidfvectorizer']
feature_names = np.array(vect.get_feature_names())
coef = grid.best_estimator_.named_steps['logisticregression'].coef_
mglearn.tools.visualize_coefficients(coef, feature_names, n_top_features=40)
```

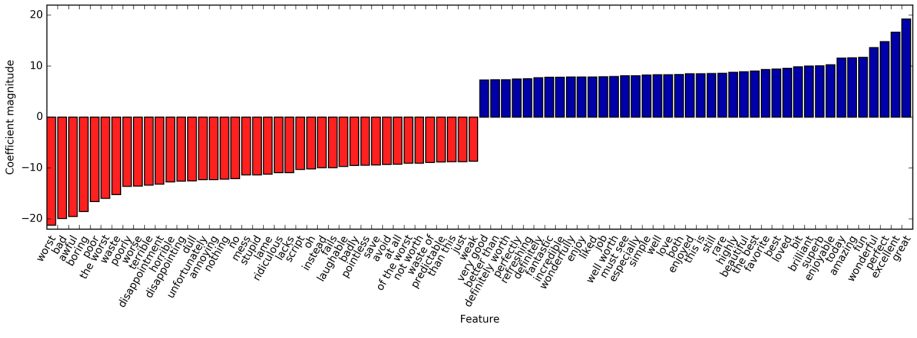

Figure 7-4. Most important features when using unigrams, bigrams, and trigrams with tf-idf rescaling

There are particularly interesting features containing the word "worth" that were not present in the unigram model: `"not worth"` is indicative of a negative review, while `"definitely worth"` and `"well worth"` are indicative of a positive review. This is a prime example of context influencing the meaning of the word "worth."

Next, we'll visualize only trigrams, to provide further insight into why these features are helpful. Many of the useful bigrams and trigrams consist of common words that would not be informative on their own, as in the phrases `"none of the"`, `"the only good"`, `"on and on"`, `"this is one"`, `"of the most"`, and so on. However, the impact of these features is quite limited compared to the importance of the unigram features, as you can see in Figure 7-5:

In[35]:

```
# find 3-gram features
mask = np.array([len(feature.split(" ")) for feature in feature_names]) == 3
# visualize only 3-gram features
mglearn.tools.visualize_coefficients(coef.ravel()[mask],
                                     feature_names[mask], n_top_features=40)
```

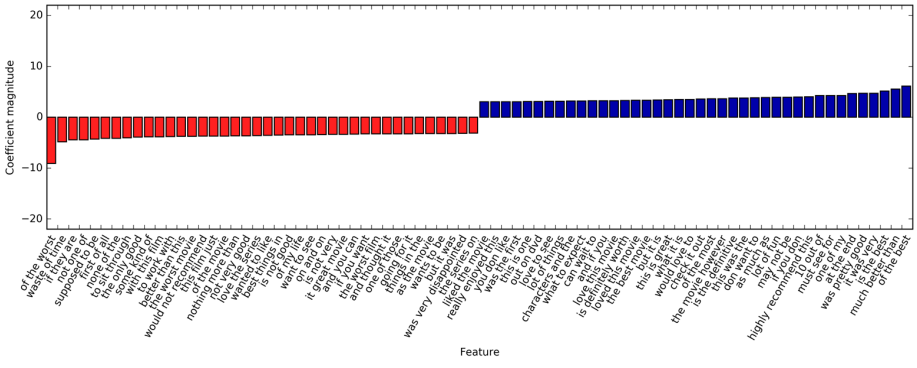

Figure 7-5. Visualization of only the important trigram features of the model

Advanced Tokenization, Stemming, and Lemmatization

As mentioned previously, the feature extraction in the `CountVectorizer` and `TfidfVectorizer` is relatively simple, and much more elaborate methods are possible. One particular step that is often improved in more sophisticated text-processing applications is the first step in the bag-of-words model: tokenization. This step defines what constitutes a word for the purpose of feature extraction.

We saw earlier that the vocabulary often contains singular and plural versions of some words, as in `"drawback"` and `"drawbacks"`, `"drawer"` and `"drawers"`, and `"drawing"` and `"drawings"`. For the purposes of a bag-of-words model, the semantics of `"drawback"` and `"drawbacks"` are so close that distinguishing them will only increase overfitting, and not allow the model to fully exploit the training data. Similarly, we found the vocabulary includes words like `"replace"`, `"replaced"`, `"replacement"`, `"replaces"`, and `"replacing"`, which are different verb forms and a noun relating to the verb "to replace." Similarly to having singular and plural forms of a noun, treating different verb forms and related words as distinct tokens is disadvantageous for building a model that generalizes well.

This problem can be overcome by representing each word using its *word stem*, which involves identifying (or *conflating*) all the words that have the same word stem. If this is done by using a rule-based heuristic, like dropping common suffixes, it is usually referred to as *stemming*. If instead a dictionary of known word forms is used (an explicit and human-verified system), and the role of the word in the sentence is taken into account, the process is referred to as *lemmatization* and the standardized form of the word is referred to as the *lemma*. Both processing methods, lemmatization and stemming, are forms of *normalization* that try to extract some normal form of a word. Another interesting case of normalization is spelling correction, which can be helpful in practice but is outside of the scope of this book.

To get a better understanding of normalization, let's compare a method for stemming —the Porter stemmer, a widely used collection of heuristics (here imported from the `nltk` package)—to lemmatization as implemented in the `spacy` package:[8]

In[36]:

```
import spacy
import nltk

# load spacy's English-language models
en_nlp = spacy.load('en')
# instantiate nltk's Porter stemmer
stemmer = nltk.stem.PorterStemmer()

# define function to compare lemmatization in spacy with stemming in nltk
def compare_normalization(doc):
    # tokenize document in spacy
    doc_spacy = en_nlp(doc)
    # print lemmas found by spacy
    print("Lemmatization:")
    print([token.lemma_ for token in doc_spacy])
    # print tokens found by Porter stemmer
    print("Stemming:")
    print([stemmer.stem(token.norm_.lower()) for token in doc_spacy])
```

We will compare lemmatization and the Porter stemmer on a sentence designed to show some of the differences:

In[37]:

```
compare_normalization(u"Our meeting today was worse than yesterday, "
                      "I'm scared of meeting the clients tomorrow.")
```

Out[37]:

```
Lemmatization:
['our', 'meeting', 'today', 'be', 'bad', 'than', 'yesterday', ',', 'i', 'be',
 'scared', 'of', 'meet', 'the', 'client', 'tomorrow', '.']
Stemming:
['our', 'meet', 'today', 'wa', 'wors', 'than', 'yesterday', ',', 'i', "'m",
 'scare', 'of', 'meet', 'the', 'client', 'tomorrow', '.']
```

Stemming is always restricted to trimming the word to a stem, so `"was"` becomes `"wa"`, while lemmatization can retrieve the correct base verb form, `"be"`. Similarly, lemmatization can normalize `"worse"` to `"bad"`, while stemming produces `"wors"`. Another major difference is that stemming reduces both occurrences of `"meeting"` to `"meet"`. Using lemmatization, the first occurrence of `"meeting"` is recognized as a

8 For details of the interface, consult the `nltk` and `spacy` documentation. We are more interested in the general principles here.

noun and left as is, while the second occurrence is recognized as a verb and reduced to `"meet"`. In general, lemmatization is a much more involved process than stemming, but it usually produces better results than stemming when used for normalizing tokens for machine learning.

While `scikit-learn` implements neither form of normalization, `CountVectorizer` allows specifying your own tokenizer to convert each document into a list of tokens using the `tokenizer` parameter. We can use the lemmatization from `spacy` to create a callable that will take a string and produce a list of lemmas:

In[38]:

```
# Technicality: we want to use the regexp-based tokenizer
# that is used by CountVectorizer and only use the lemmatization
# from spacy. To this end, we replace en_nlp.tokenizer (the spacy tokenizer)
# with the regexp-based tokenization.
import re
# regexp used in CountVectorizer
regexp = re.compile('(?u)\\b\\w\\w+\\b')

# load spacy language model and save old tokenizer
en_nlp = spacy.load('en')
old_tokenizer = en_nlp.tokenizer
# replace the tokenizer with the preceding regexp
en_nlp.tokenizer = lambda string: old_tokenizer.tokens_from_list(
    regexp.findall(string))

# create a custom tokenizer using the spacy document processing pipeline
# (now using our own tokenizer)
def custom_tokenizer(document):
    doc_spacy = en_nlp(document, entity=False, parse=False)
    return [token.lemma_ for token in doc_spacy]

# define a count vectorizer with the custom tokenizer
lemma_vect = CountVectorizer(tokenizer=custom_tokenizer, min_df=5)
```

Let's transform the data and inspect the vocabulary size:

In[39]:

```
# transform text_train using CountVectorizer with lemmatization
X_train_lemma = lemma_vect.fit_transform(text_train)
print("X_train_lemma.shape: {}".format(X_train_lemma.shape))

# standard CountVectorizer for reference
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("X_train.shape: {}".format(X_train.shape))
```

Out[39]:

```
X_train_lemma.shape:  (25000, 21596)
X_train.shape:  (25000, 27271)
```

As you can see from the output, lemmatization reduced the number of features from 27,271 (with the standard `CountVectorizer` processing) to 21,596. Lemmatization can be seen as a kind of regularization, as it conflates certain features. Therefore, we expect lemmatization to improve performance most when the dataset is small. To illustrate how lemmatization can help, we will use `StratifiedShuffleSplit` for cross-validation, using only 1% of the data as training data and the rest as test data:

In[40]:

```
# build a grid search using only 1% of the data as the training set
from sklearn.model_selection import StratifiedShuffleSplit

param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
cv = StratifiedShuffleSplit(n_iter=5, test_size=0.99,
                            train_size=0.01, random_state=0)
grid = GridSearchCV(LogisticRegression(), param_grid, cv=cv)
# perform grid search with standard CountVectorizer
grid.fit(X_train, y_train)
print("Best cross-validation score "
      "(standard CountVectorizer): {:.3f}".format(grid.best_score_))
# perform grid search with lemmatization
grid.fit(X_train_lemma, y_train)
print("Best cross-validation score "
      "(lemmatization): {:.3f}".format(grid.best_score_))
```

Out[40]:

```
Best cross-validation score (standard CountVectorizer): 0.721
Best cross-validation score (lemmatization): 0.731
```

In this case, lemmatization provided a modest improvement in performance. As with many of the different feature extraction techniques, the result varies depending on the dataset. Lemmatization and stemming can sometimes help in building better (or at least more compact) models, so we suggest you give these techniques a try when trying to squeeze out the last bit of performance on a particular task.

Topic Modeling and Document Clustering

One particular technique that is often applied to text data is *topic modeling*, which is an umbrella term describing the task of assigning each document to one or multiple *topics*, usually without supervision. A good example for this is news data, which might be categorized into topics like "politics," "sports," "finance," and so on. If each document is assigned a single topic, this is the task of clustering the documents, as discussed in Chapter 3. If each document can have more than one topic, the task

relates to the decomposition methods from Chapter 3. Each of the components we learn then corresponds to one topic, and the coefficients of the components in the representation of a document tell us how strongly related that document is to a particular topic. Often, when people talk about topic modeling, they refer to one particular decomposition method called *Latent Dirichlet Allocation* (often LDA for short).[9]

Latent Dirichlet Allocation

Intuitively, the LDA model tries to find groups of words (the topics) that appear together frequently. LDA also requires that each document can be understood as a "mixture" of a subset of the topics. It is important to understand that for the machine learning model a "topic" might not be what we would normally call a topic in everyday speech, but that it resembles more the components extracted by PCA or NMF (which we discussed in Chapter 3), which might or might not have a semantic meaning. Even if there is a semantic meaning for an LDA "topic", it might not be something we'd usually call a topic. Going back to the example of news articles, we might have a collection of articles about sports, politics, and finance, written by two specific authors. In a politics article, we might expect to see words like "governor," "vote," "party," etc., while in a sports article we might expect words like "team," "score," and "season." Words in each of these groups will likely appear together, while it's less likely that, for example, "team" and "governor" will appear together. However, these are not the only groups of words we might expect to appear together. The two reporters might prefer different phrases or different choices of words. Maybe one of them likes to use the word "demarcate" and one likes the word "polarize." Other "topics" would then be "words often used by reporter A" and "words often used by reporter B," though these are not topics in the usual sense of the word.

Let's apply LDA to our movie review dataset to see how it works in practice. For unsupervised text document models, it is often good to remove very common words, as they might otherwise dominate the analysis. We'll remove words that appear in at least 20 percent of the documents, and we'll limit the bag-of-words model to the 10,000 words that are most common after removing the top 20 percent:

In[41]:

```
vect = CountVectorizer(max_features=10000, max_df=.15)
X = vect.fit_transform(text_train)
```

9 There is another machine learning model that is also often abbreviated LDA: Linear Discriminant Analysis, a linear classification model. This leads to quite some confusion. In this book, LDA refers to Latent Dirichlet Allocation.

We will learn a topic model with 10 topics, which is few enough that we can look at all of them. Similarly to the components in NMF, topics don't have an inherent ordering, and changing the number of topics will change all of the topics.[10] We'll use the `"batch"` learning method, which is somewhat slower than the default (`"online"`) but usually provides better results, and increase `"max_iter"`, which can also lead to better models:

In[42]:

```
from sklearn.decomposition import LatentDirichletAllocation
lda = LatentDirichletAllocation(n_topics=10, learning_method="batch",
                                max_iter=25, random_state=0)
# We build the model and transform the data in one step
# Computing transform takes some time,
# and we can save time by doing both at once
document_topics = lda.fit_transform(X)
```

Like the decomposition methods we saw in Chapter 3, `LatentDirichletAllocation` has a `components_` attribute that stores how important each word is for each topic. The size of `components_` is `(n_topics, n_words)`:

In[43]:

```
lda.components_.shape
```

Out[43]:

```
(10, 10000)
```

To understand better what the different topics mean, we will look at the most important words for each of the topics. The `print_topics` function provides a nice formatting for these features:

In[44]:

```
# For each topic (a row in the components_), sort the features (ascending)
# Invert rows with [:, ::-1] to make sorting descending
sorting = np.argsort(lda.components_, axis=1)[:, ::-1]
# Get the feature names from the vectorizer
feature_names = np.array(vect.get_feature_names())
```

In[45]:

```
# Print out the 10 topics:
mglearn.tools.print_topics(topics=range(10), feature_names=feature_names,
                           sorting=sorting, topics_per_chunk=5, n_words=10)
```

10 In fact, NMF and LDA solve quite related problems, and we could also use NMF to extract topics.

Out[45]:

```
topic 0       topic 1       topic 2       topic 3       topic 4
--------      --------      --------      --------      --------
between       war           funny         show          didn
young         world         worst         series        saw
family        us            comedy        episode       am
real          our           thing         tv            thought
performance   american      guy           episodes      years
beautiful     documentary   re            shows         book
work          history       stupid        season        watched
each          new           actually      new           now
both          own           nothing       television    dvd
director      point         want          years         got


topic 5       topic 6       topic 7       topic 8       topic 9
--------      --------      --------      --------      --------
horror        kids          cast          performance   house
action        action        role          role          woman
effects       animation     john          john          gets
budget        game          version       actor         killer
nothing       fun           novel         oscar         girl
original      disney        both          cast          wife
director      children      director      plays         horror
minutes       10            played        jack          young
pretty        kid           performance   joe           goes
doesn         old           mr            performances  around
```

Judging from the important words, topic 1 seems to be about historical and war movies, topic 2 might be about bad comedies, topic 3 might be about TV series. Topic 4 seems to capture some very common words, while topic 6 appears to be about children's movies and topic 8 seems to capture award-related reviews. Using only 10 topics, each of the topics needs to be very broad, so that they can together cover all the different kinds of reviews in our dataset.

Next, we will learn another model, this time with 100 topics. Using more topics makes the analysis much harder, but makes it more likely that topics can specialize to interesting subsets of the data:

In[46]:

```
lda100 = LatentDirichletAllocation(n_topics=100, learning_method="batch",
                                   max_iter=25, random_state=0)
document_topics100 = lda100.fit_transform(X)
```

Looking at all 100 topics would be a bit overwhelming, so we selected some interesting and representative topics:

In[47]:

```
topics = np.array([7, 16, 24, 25, 28, 36, 37, 45, 51, 53, 54, 63, 89, 97])

sorting = np.argsort(lda100.components_, axis=1)[:, ::-1]
feature_names = np.array(vect.get_feature_names())
mglearn.tools.print_topics(topics=topics, feature_names=feature_names,
                           sorting=sorting, topics_per_chunk=7, n_words=20)
```

Out[48]:

```
topic 7       topic 16      topic 24      topic 25      topic 28
--------      --------      --------      --------      --------
thriller      worst         german        car           beautiful
suspense      awful         hitler        gets          young
horror        boring        nazi          guy           old
atmosphere    horrible      midnight      around        romantic
mystery       stupid        joe           down          between
house         thing         germany       kill          romance
director      terrible      years         goes          wonderful
quite         script        history       killed        heart
bit           nothing       new           going         feel
de            worse         modesty       house         year
performances  waste         cowboy        away          each
dark          pretty        jewish        head          french
twist         minutes       past          take          sweet
hitchcock     didn          kirk          another       boy
tension       actors        young         getting       loved
interesting   actually      spanish       doesn         girl
mysterious    re            enterprise    now           relationship
murder        supposed      von           night         saw
ending        mean          nazis         right         both
creepy        want          spock         woman         simple


topic 36      topic 37      topic 41      topic 45      topic 51
--------      --------      --------      --------      --------
performance   excellent     war           music         earth
role          highly        american      song          space
actor         amazing       world         songs         planet
cast          wonderful     soldiers      rock          superman
play          truly         military      band          alien
actors        superb        army          soundtrack    world
performances  actors        tarzan        singing       evil
played        brilliant     soldier       voice         humans
supporting    recommend     america       singer        aliens
director      quite         country       sing          human
oscar         performance   americans     musical       creatures
roles         performances  during        roll          miike
actress       perfect       men           fan           monsters
excellent     drama         us            metal         apes
screen        without       government    concert       clark
plays         beautiful     jungle        playing       burton
award         human         vietnam       hear          tim
work          moving        ii            fans          outer
playing       world         political     prince        men
gives         recommended   against       especially    moon
```

```
topic 53      topic 54      topic 63      topic 89      topic 97
--------      --------      --------      --------      --------
scott         money         funny         dead          didn
gary          budget        comedy        zombie        thought
streisand     actors        laugh         gore          wasn
star          low           jokes         zombies       ending
hart          worst         humor         blood         minutes
lundgren      waste         hilarious     horror        got
dolph         10            laughs        flesh         felt
career        give          fun           minutes       part
sabrina       want          re            body          going
role          nothing       funniest      living        seemed
temple        terrible      laughing      eating        bit
phantom       crap          joke          flick         found
judy          must          few           budget        though
melissa       reviews       moments       head          nothing
zorro         imdb          guy           gory          lot
gets          director      unfunny       evil          saw
barbra        thing         times         shot          long
cast          believe       laughed       low           interesting
short         am            comedies      fulci         few
serial        actually      isn           re            half
```

The topics we extracted this time seem to be more specific, though many are hard to interpret. Topic 7 seems to be about horror movies and thrillers; topics 16 and 54 seem to capture bad reviews, while topic 63 mostly seems to be capturing positive reviews of comedies. If we want to make further inferences using the topics that were discovered, we should confirm the intuition we gained from looking at the highest-ranking words for each topic by looking at the documents that are assigned to these topics. For example, topic 45 seems to be about music. Let's check which kinds of reviews are assigned to this topic:

In[49]:

```
# sort by weight of "music" topic 45
music = np.argsort(document_topics100[:, 45])[::-1]
# print the five documents where the topic is most important
for i in music[:10]:
    # pshow first two sentences
    print(b".".join(text_train[i].split(b".")[:2]) + b".\n")
```

Out[49]:

```
b'I love this movie and never get tired of watching. The music in it is great.\n'
b"I enjoyed Still Crazy more than any film I have seen in years. A successful
  band from the 70's decide to give it another try.\n"
b'Hollywood Hotel was the last movie musical that Busby Berkeley directed for
  Warner Bros. His directing style had changed or evolved to the point that
  this film does not contain his signature overhead shots or huge production
  numbers with thousands of extras.\n'
b"What happens to washed up rock-n-roll stars in the late 1990's?
  They launch a comeback / reunion tour. At least, that's what the members of
  Strange Fruit, a (fictional) 70's stadium rock group do.\n"
```

```
b'As a big-time Prince fan of the last three to four years, I really can\'t
  believe I\'ve only just got round to watching "Purple Rain". The brand new
  2-disc anniversary Special Edition led me to buy it.\n'
b"This film is worth seeing alone for Jared Harris' outstanding portrayal
  of John Lennon. It doesn't matter that Harris doesn't exactly resemble
  Lennon; his mannerisms, expressions, posture, accent and attitude are
  pure Lennon.\n"
b"The funky, yet strictly second-tier British glam-rock band Strange Fruit
  breaks up at the end of the wild'n'wacky excess-ridden 70's. The individual
  band members go their separate ways and uncomfortably settle into lackluster
  middle age in the dull and uneventful 90's: morose keyboardist Stephen Rea
  winds up penniless and down on his luck, vain, neurotic, pretentious lead
  singer Bill Nighy tries (and fails) to pursue a floundering solo career,
  paranoid drummer Timothy Spall resides in obscurity on a remote farm so he
  can avoid paying a hefty back taxes debt, and surly bass player Jimmy Nail
  installs roofs for a living.\n"
b"I just finished reading a book on Anita Loos' work and the photo in TCM
  Magazine of MacDonald in her angel costume looked great (impressive wings),
  so I thought I'd watch this movie. I'd never heard of the film before, so I
  had no preconceived notions about it whatsoever.\n"
b'I love this movie!!! Purple Rain came out the year I was born and it has had
  my heart since I can remember. Prince is so tight in this movie.\n'
b"This movie is sort of a Carrie meets Heavy Metal. It's about a highschool
  guy who gets picked on alot and he totally gets revenge with the help of a
  Heavy Metal ghost.\n"
```

As we can see, this topic covers a wide variety of music-centered reviews, from musicals, to biographical movies, to some hard-to-specify genre in the last review. Another interesting way to inspect the topics is to see how much weight each topic gets overall, by summing the `document_topics` over all reviews. We name each topic by the two most common words. Figure 7-6 shows the topic weights learned:

In[50]:

```
fig, ax = plt.subplots(1, 2, figsize=(10, 10))
topic_names = ["{:>2} ".format(i) + " ".join(words)
               for i, words in enumerate(feature_names[sorting[:, :2]])]
# two column bar chart:
for col in [0, 1]:
    start = col * 50
    end = (col + 1) * 50
    ax[col].barh(np.arange(50), np.sum(document_topics100, axis=0)[start:end])
    ax[col].set_yticks(np.arange(50))
    ax[col].set_yticklabels(topic_names[start:end], ha="left", va="top")
    ax[col].invert_yaxis()
    ax[col].set_xlim(0, 2000)
    yax = ax[col].get_yaxis()
    yax.set_tick_params(pad=130)
plt.tight_layout()
```

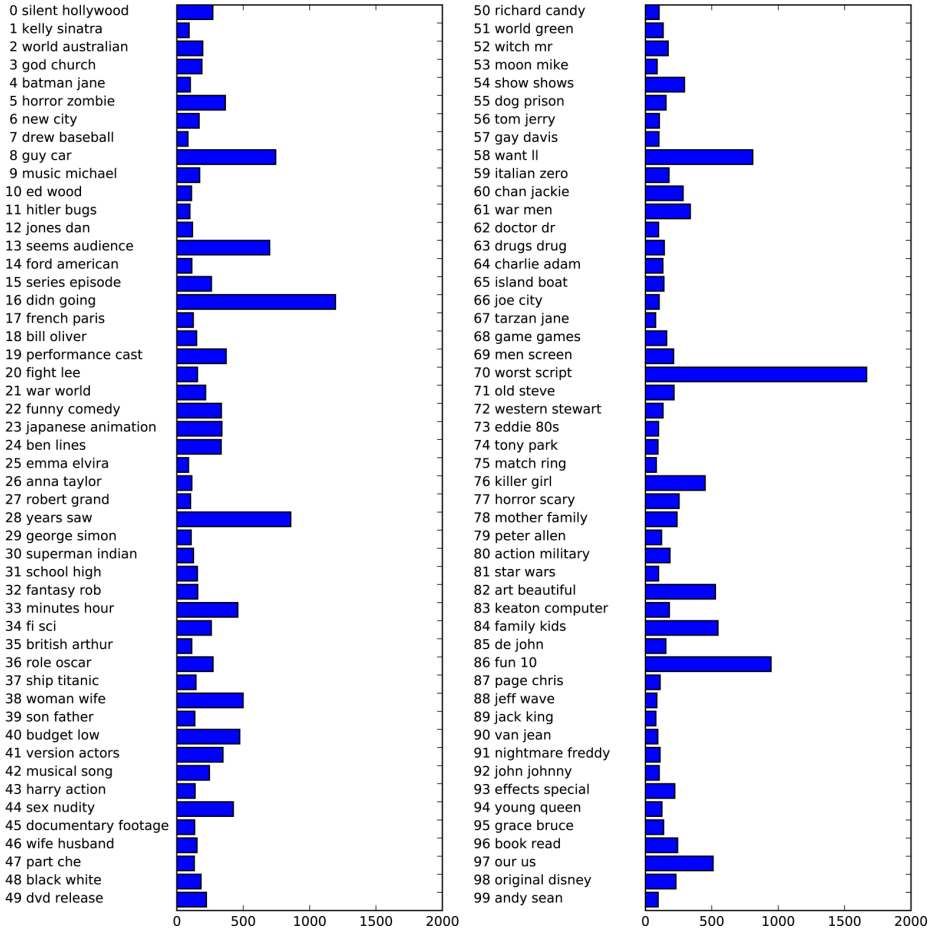

Figure 7-6. Topic weights learned by LDA

The most important topics are 97, which seems to consist mostly of stopwords, possibly with a slight negative direction; topic 16, which is clearly about bad reviews; followed by some genre-specific topics and 36 and 37, both of which seem to contain laudatory words.

It seems like LDA mostly discovered two kind of topics, genre-specific and rating-specific, in addition to several more unspecific topics. This is an interesting discovery, as most reviews are made up of some movie-specific comments and some comments that justify or emphasize the rating.

Topic models like LDA are interesting methods to understand large text corpora in the absence of labels—or, as here, even if labels are available. The LDA algorithm is randomized, though, and changing the `random_state` parameter can lead to quite

different outcomes. While identifying topics can be helpful, any conclusions you draw from an unsupervised model should be taken with a grain of salt, and we recommend verifying your intuition by looking at the documents in a specific topic. The topics produced by the `LDA.transform` method can also sometimes be used as a compact representation for supervised learning. This is particularly helpful when few training examples are available.

Summary and Outlook

In this chapter we talked about the basics of processing text, also known as *natural language processing* (NLP), with an example application classifying movie reviews. The tools discussed here should serve as a great starting point when trying to process text data. In particular for text classification tasks such as spam and fraud detection or sentiment analysis, bag-of-words representations provide a simple and powerful solution. As is often the case in machine learning, the representation of the data is key in NLP applications, and inspecting the tokens and *n*-grams that are extracted can give powerful insights into the modeling process. In text-processing applications, it is often possible to introspect models in a meaningful way, as we saw in this chapter, for both supervised and unsupervised tasks. You should take full advantage of this ability when using NLP-based methods in practice.

Natural language and text processing is a large research field, and discussing the details of advanced methods is far beyond the scope of this book. If you want to learn more, we recommend the O'Reilly book *Natural Language Processing with Python* by Steven Bird, Ewan Klein, and Edward Loper, which provides an overview of NLP together with an introduction to the `nltk` Python package for NLP. Another great and more conceptual book is the standard reference *Introduction to Information Retrieval* by Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze, which describes fundamental algorithms in information retrieval, NLP, and machine learning. Both books have online versions that can be accessed free of charge. As we discussed earlier, the classes `CountVectorizer` and `TfidfVectorizer` only implement relatively simple text-processing methods. For more advanced text-processing methods, we recommend the Python packages `spacy` (a relatively new but very efficient and well-designed package), `nltk` (a very well-established and complete but somewhat dated library), and `gensim` (an NLP package with an emphasis on topic modeling).

There have been several very exciting new developments in text processing in recent years, which are outside of the scope of this book and relate to neural networks. The first is the use of continuous vector representations, also known as word vectors or distributed word representations, as implemented in the `word2vec` library. The original paper "Distributed Representations of Words and Phrases and Their Compositionality" by Thomas Mikolov et al. is a great introduction to the subject. Both `spacy`

and `gensim` provide functionality for the techniques discussed in this paper and its follow-ups.

Another direction in NLP that has picked up momentum in recent years is the use of *recurrent neural networks* (RNNs) for text processing. RNNs are a particularly powerful type of neural network that can produce output that is again text, in contrast to classification models that can only assign class labels. The ability to produce text as output makes RNNs well suited for automatic translation and summarization. An introduction to the topic can be found in the relatively technical paper "Sequence to Sequence Learning with Neural Networks" by Ilya Suskever, Oriol Vinyals, and Quoc Le. A more practical tutorial using the `tensorflow` framework can be found on the TensorFlow website.