CHAPTER 8

Working with Text Data

In Chapter 5, 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 application, which is text.

For example, if we want to classify in email into whether it is 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, their speeches or tweets might provide useful information.

In customer services, we often want to find out if a message is a complaint or an inquiry. And depending on the kind of complaint, we might be able to provide automatic advice or forward it to a specific department. These decisions can all be supported by the content of the message that was sent to customer service.

Text data is usually represented as strings, made up of characters. In any of the examples above, the length of the text of each text will be different.

This feature is clearly very different from the numeric features that we discussed so far, and we need to process the text data before we can apply our machine learning algorithms to the text data.

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 each string feature 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 8 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 confirming 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 half-way 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 user to provide their own favorite color. Many people might respond with a color name like "black" or "blue". Others might have typographic errors, or use aliases, or different spellings like "gray" and "grey", use more evocative names like "midnight blue", and there will certainly be answers that can not reasonably be related to any color, say "calmly checkered hissing" or "asdfasdfasdf".

The responses you can obtain from a text field belong to the second category, *free strings that correspond to a set of categories*. It will probably be best to encode this data as a categorical variable, where you can select the categories either 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 can not 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 kind 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 *free form text that consists of phrases or sentences*. Examples of these include tweets, chat logs, hotel reviews, but also the collected works of Shakespeare, the content of Wikipedia or the project Gutenberg collection of 50.000 e-books. All of these collections contain information mostly as sentences of words[footnote: arguably the content of websites linked to in tweets contain more information than the text of the tweet]. For simplicity's sake, let's assume all our documents are in one language, English [footnote: 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 also to other alphabets with word boundary delimiters. Chinese for example does not delimit word boundaries, and has other challenges that make applying the techniques of this chapter difficult]. In the context of text analysis, the dataset if 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 data set of movie reviews collected from the IMDb (Internet Movie Database) website collected by Standford Researcher Andrew Maas [footnote: The dataset is available at http://ai.stanford.edu/~amaas/data/sentiment/]. This dataset contains the text of the reviews, together with a label that indicates "positive" and "negative" reviews. The IMDb website itself contains ratings from one to ten. 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 data set 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 "positive" and one called "negative":

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

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


6 directories, 0 files
```

The "positive" folder contains all the positive documents, each as a separate text file, and similarly for the "negative" 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:

```
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: ", type(text_train))
print("length of text_train: ", len(text_train))
print("text_train[1]:")
# print review number 1
print(text_train[1])
```

```
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 s
```

We 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 one. You can 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 from this formating before we proceed:

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

The type of the entries of `text_train` depends on your Python version. In Python3, they will be of type "bytes" which represents a binary encoding of the string data. In

Python2, `text_train` contains strings. We won't go into the details of the different string types in Python here, but recommend that you read the documentation regarding strings and unicode in Python [Footnote: https://docs.python.org/3/howto/unicode.html for Python 3 and https://docs.python.org/2/howto/unicode.html for Python 2].

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

```
print(np.bincount(y_train))
```

```
[12500 12500]
```

We load the test dataset in the same manner:

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

```
Number of documents in test data: 25000

[12500 12500]
```

The task we want to solve is given a review, we want to assign the labels "positive" and "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 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 bag-of-words, 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*. Discarding all this structure and counting only occurrence leads to the mental image of representing text as a "bag".

Computing the bag-of-word 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 by 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.

bag_of_words

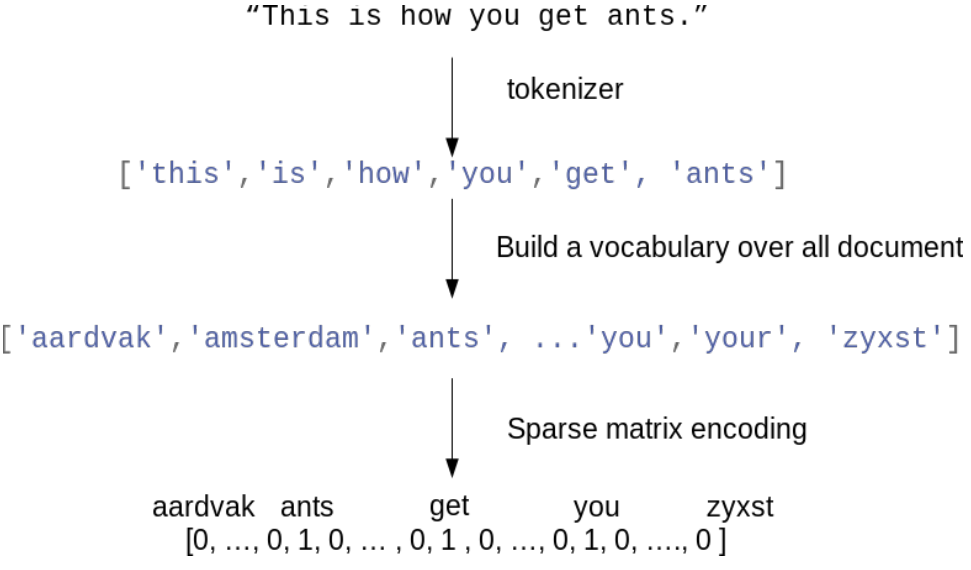


Figure bag_of_words illustrates the process on the string “This is how you get ants”. The output of the process is one vector of word-counts for each document. For each word in the vocabulary, we have a count of how often it appears in each document. That means our numeric representation has one feature 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. There are some subtleties involved in step 1 and step 2 above, which we will discuss in more detail later in this chapter.

For now, let’s look at how we can apply the bag-of-word processing using scikit-learn.

Applying bag-of-words to a toy dataset

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

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

We import and instantiate the `CountVecorizer` and `fit` it to our toy data:

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

```
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
        dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), preprocessor=None, stop_words=None,
        strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
        tokenizer=None, vocabulary=None)
```

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

```
print(len(vect.vocabulary_))
print("vocabulary content:")
vect.vocabulary_
```

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

vocabulary content:
```

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:

```
bag_of_words = vect.transform(bards_words)
bag_of_words
```

```
<2x13 sparse matrix of type '<class 'numpy.int64'>'

    with 16 stored elements in Compressed Sparse Row format>
```

The bag of word representation is stored in a SciPy sparse matrix that only stores the entries that are non-zero (see Chapter 1). The matrix is of shape 2 x 13, 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 zero. 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 these 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 zero entries) using the `toarray` method. 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`.

```
print(bag_of_words.toarray())
```

```
[[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 zero or one, none of the two strings in `bards_words` contain a word twice. You can read these feature vectors as follows: 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 does contain 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-word for movie reviews

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

```
vect = CountVectorizer().fit(text_train)
X_train = vect.transform(text_train)
print(repr(X_train))
```

```
<25000x74849 sparse matrix of type '<class 'numpy.int64'>'

    with 3431196 stored elements in Compressed Sparse Row format>
```

The shape of `X_train`, the bag-of-words representation of the training data, is 25.000 x 74.849, indicating that the vocabulary contains 74.849 entries. Again, the data is stored as a SciPy sparse matrix. Let's look in a bit more detail at the vocabulary. 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:

```
feature_names = vect.get_feature_names()
print(len(feature_names))
# print first fifty features
print(feature_names[:20])
# print feature 20010 to 20030
print(feature_names[20010:20030])
# get every 2000th word to get an overview
print(feature_names[::2000])
```

```
74849

['00', '000', '0000000000001', '00001', '00015', '000s', '001', '003830', '006', '007', '0079', '0

['dratted', 'draub', 'draught', 'draughts', 'draughtswoman', 'draw', 'drawback', 'drawbacks', 'dra

['00', 'aesir', 'aquarian', 'barking', 'blustering', 'bête', 'chicanery', 'condensing', 'cunning',
```

As you can see, possibly a bit surprisingly, is that the first ten 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 particular in the context of movies, is likely to refer to the James Bond character [footnote: A quick analysis of the data confirms that this is indeed the case. Try confirming it yourself.]. Weeding out the meaningful from the non-meaningful "words" is sometimes tricky. Looking at some words 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 form 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.

Before we try to improve our feature extraction, let us obtain a quantitative measure of performance by actually building a classifier. We have the training labels stored in `y_train` and the bag-of-word 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[footnote: The attentive reader might notice that we violate our lesson from Chapter 7 on cross-validation with preprocessing. 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.]

```
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)
np.mean(scores)
```

```
0.88131999999999999
```

We obtain a mean cross-validation score of 88.2%, 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:

```
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: ", grid.best_score_)
print("Best parameters: ", grid.best_params_)
```

```
Best cross-validation score:  0.88816


Best parameters:  {'C': 0.1}
```

We obtain a cross-validation score of 88.8% using `C=0.1`. We can now assess the generalization-performance of this parameter setting on the test set:

```
X_test = vect.transform(text_test)
grid.score(X_test, y_test)
```

```
0.87895999999999996
```

Now, let's see if we can improve the extraction of words. The way the `CountVectorizer` extracts tokens is 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"`), in particular it does not find single-letter words, and it splits up contractions like "doesn't" or "bit.ly", but matches "h8ter" as a single word. The `CountVectorizer` then converts all words to lower-case 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 above, 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 2 documents (or at least 5 documents etc). 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:

```
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print(repr(X_train))
```

```
<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 27.272 (see the output above), only about a third of the original features. Let's look at some tokens again:

```
feature_names = vect.get_feature_names()

# print first fifty features
print(feature_names[:50])
# print feature 20010 to 20020
print(feature_names[20010:20030])
#
print(feature_names[::700])
```

```
['00', '000', '007', '00s', '01', '02', '03', '04', '05', '06', '07', '08', '09', '10', '100', '10

['repentance', 'repercussions', 'repertoire', 'repetition', 'repetitions', 'repetitious', 'repetit

['00', 'affections', 'appropriately', 'barbra', 'blurbs', 'butchered', 'cheese', 'commitment', 'co
```

There are clearly much 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:

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

```
Best cross-validation score:  0.88812
```

The best validation accuracy of the grid-search is still 88.8%, unchanged from before. We didn't improve our model, but having less features to deal with speeds up processing and throwing away useless features might make the model more interpretable.

Stop-words

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 stop words, or discarding words that appear too frequently. Scikit-learn had a built-in list of English stop-words in the `feature_extraction.text` module:

```
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
# print number of stop words
```

```
print(len(ENGLISH_STOP_WORDS))
# print some of the stop words
print(list(ENGLISH_STOP_WORDS)[::10])
```

```
318

['show', 'however', 'something', 'is', 'of', 'are', 'about', 'least', 'eight', 'thereupon', 'alway
```

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

```
# specifying "english" uses the build-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(repr(X_train))
```

```
<25000x26966 sparse matrix of type '<class 'numpy.int64'>'

    with 2149958 stored elements in Compressed Sparse Row format>
```

There are now 305 (=27272 - 26967) less features in the dataset, which means that most, but not all of the stop-words appeared. Let's run the grid-search again:

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

```
Best cross-validation score:  0.88296
```

The grid-search performance decreased slightly using the stop words. The change is very slight, but given that excluding 305 features 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, that might not contain enough information for the model to determine which words are stop words 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 TFIDF

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 a 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, the `TfidfTransformer`, which takes in the sparse matrix output produced by `CountVectorizer` and transforms it, or `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 find on the wikipedia page [footnote: https://en.wikipedia.org/wiki/Tf-idf]. The tf-idf score for word w in document d as implemented in both the `TfidfTransformer` and `TfidfVectorizer` is given by:

\begin{equation*}

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

\end{equation*}

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 d appears in, and tf, the term frequency, is the number of times that the word w appears in the query document (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. We provide this formula here mostly for completeness, and you don't need to remember it to use the tf-idf encoding.

Because tf-idf actually makes use of the statistical properties of the training data, we will use a pipeline, as described in Chapter 7, to ensure the results of our grid-search are valid. This leads to the following code:

```
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: ", grid.best_score_)
```

```
Best cross-validation score:  0.89392
```

As you can see, there is some improvement of using tf-idf instead of using 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 related to the "positive review" and "negative review" labels we are interested in. First we extract the `TfidfVectorizer` from the pipeline:

```
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 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")
print(feature_names[sorted_by_tfidf[:20]])

print("features with highest tfidf")
print(feature_names[sorted_by_tfidf[-20:]])
```

```
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 are either 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 review.

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:

```
sorted_by_idf = np.argsort(vectorizer.idf_)
print("features with lowest idf")
print(feature_names[sorted_by_idf[:100]])
```

```
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 stop words 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, even though we might expect these to be very important for our sentiment analysis task.

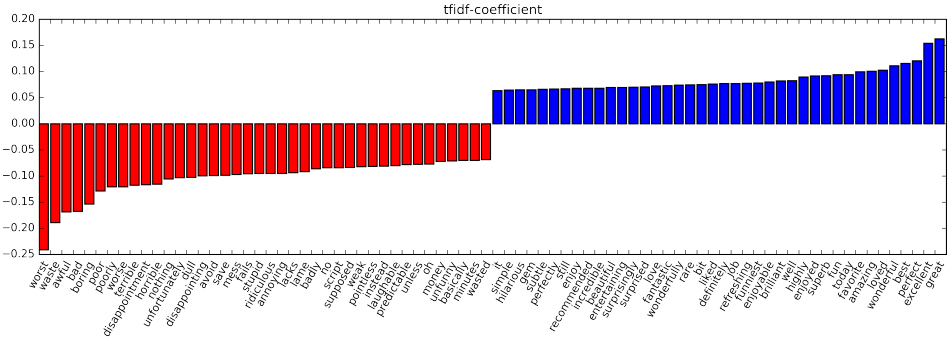
Investigating model coefficients

Finally, let us look into a bit more detail into what our logistic regression model actually learned from the data.

Because there are so many features, 27.272 after removing the infrequent ones, we can clearly not 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.

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



The bar-chart in Figure tfidf-coefficient shows the 25 largest and 25 smallest coefficients of the logistic regression model, with the bar showing the size of each coefficient. 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 word 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-word 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 (if extreme) example of how context matters. There is a way of capturing context when using a bag-of-word representation, by not only considering the counts of single tokens, but also the counts of pairs or triples 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 the `CountVecorizer` or `TfidfVectorizer`. The `ngram_range` parameter is a tuple, consisting of the minimum length and the maximum length of the sequences of tokens that are considered. Here is an example on the toy data from above:

```
print(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 are at least one token long, and at most one token long, in other words exactly one token long (single tokens are also called *unigrams*):

```
cv = CountVectorizer(ngram_range=(1, 1)).fit(bards_words)
print(len(cv.vocabulary_))
print(cv.get_feature_names())
```

```
13

['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)`:

```
cv = CountVectorizer(ngram_range=(2, 2)).fit(bards_words)
print(len(cv.vocabulary_))
print(cv.get_feature_names())
```

```
14

['be fool', 'but the', 'doth think', 'fool doth', 'he is', 'himself to', 'is wise', 'knows himself
```

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`:

```
cv.transform(bards_words).toarray()
```

```
array([[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, and adding longer sequences, up to 5-grams, might help, but will lead to an explosion of the number of features, and might lead to overfitting, as there are many very specific features.

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

```
cv = CountVectorizer(ngram_range=(1, 3)).fit(bards_words)
print(len(cv.vocabulary_))
print(cv.get_feature_names())
```

```
39

['be', 'be fool', 'but', 'but the', 'but the wise', 'doth', 'doth think', 'doth think he', 'fool',
```

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

```
pipe = make_pipeline(TfidfVectorizer(min_df=5), LogisticRegression())
param_grid = {'logisticregression__C': [0.001, 0.01, 0.1, 1, 10, 100, 1000],
              "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: ", grid.best_score_)
grid.best_params_
```

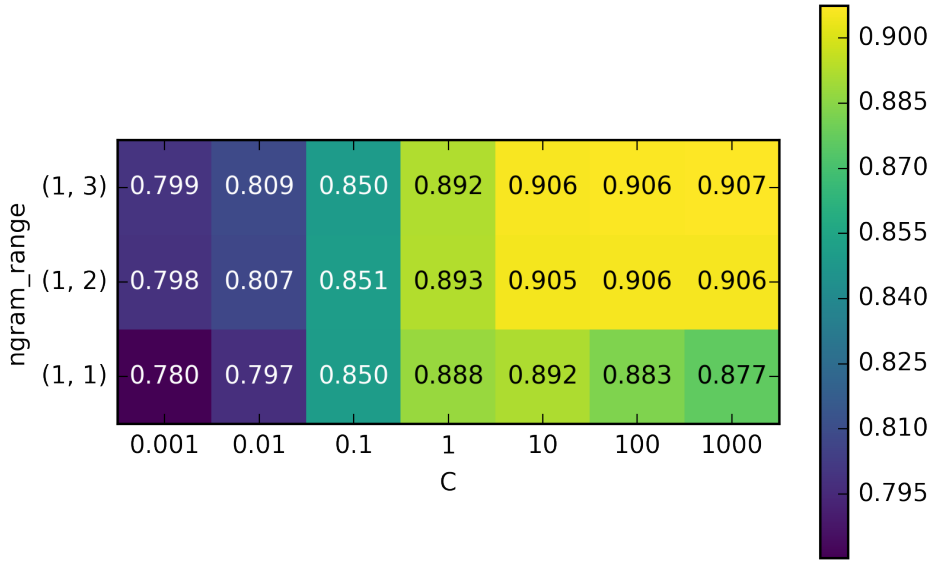
```
{'logisticregression__C': 1000, 'tfidfvectorizer__ngram_range': (1, 3)}
```

```
Best cross-validation score:  0.9074
```

As you can see from the results, we improved performance 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 6:

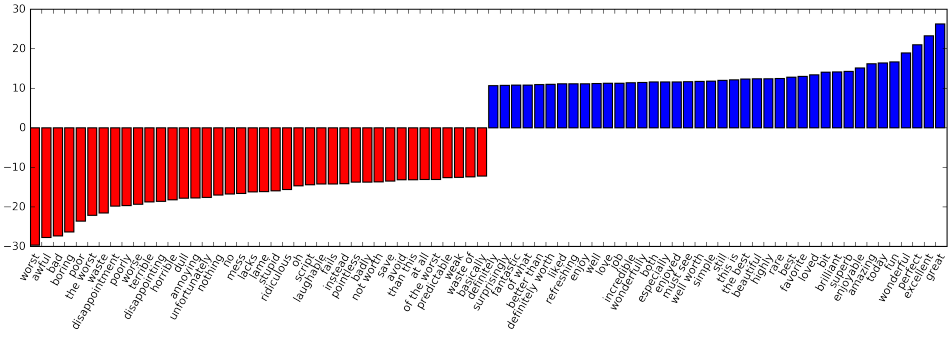
```
# extract scores from grid_search
scores = [s.mean_validation_score for s in grid.grid_scores_]
scores = np.array(scores).reshape(-1, 3).T
# visualize heatmap
heatmap = mglearn.tools.heatmap(scores, xlabel="C", ylabel="ngram_range",
                                xticklabels=param_grid['logisticregression__C'],
                                yticklabels=param_grid['tfidfvectorizer__ngram_range'],
                                cmap="viridis", fmt="%.3f")
plt.colorbar(heatmap);
```



From the heat map we can see that using bigrams increases performance quite a bit, while adding three-grams only provides a very small benefit in terms of accuracy. To understand better how the model improved, we visualize the important coefficient for the best model (which includes unigrams, bigrams and trigrams):

```
# extract feature names and coefficients
feature_names = np.array(grid.best_estimator_.named_steps['tfidfvectorizer'].get_feature_names())
coef = grid.best_estimator_.named_steps['logisticregression'].coef_
```

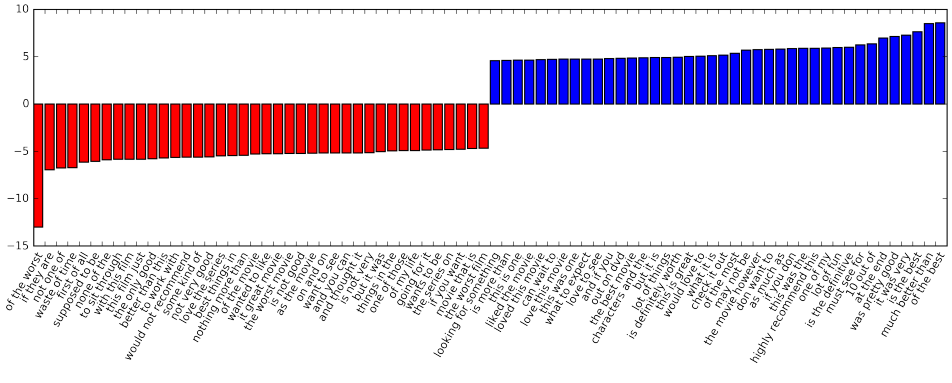
```
mglearn.tools.visualize_coefficients(coef, feature_names, n_top_features=40)
plt.title("ngram-coefficient")
```



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 wroth" and "well worth" are indicative of a positive review. This is a prime example of context influencing the meaning of the word "worth".

Below, we visualize only bigrams and 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 was one of", "of the most" and so on. However, the impact of these features is quite limited compared to the importance of the unigram features.

```
# 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)
```



Advanced tokenization, stemming and lemmatization

We mentioned above that 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-word model, the tokenization, the step defines what constitutes a word for the purpose of feature extraction.

We saw above that the vocabulary often contains singular and plural version of words as in `'drawback', 'drawbacks', 'drawer', 'drawers', 'drawing', 'drawings'`. For the purpose 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', 'replacing'`, which are different verb forms and a nouns relating to the verb "to replace".

Similarly to having singular and plural 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*, 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, this is usually referred to as *stemming*. If instead a dictionary of known word forms is used (that is using an explicit and human-verified system), and the role of the word in the sentence taken into account, the process is referred to as *lemmatization* and the standardized form of the word is referred to as *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 spell correction, which can be helpful in practice, but is outside of the scope of this book.

To get a better feeling for 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. For details of the interface, consult the `nltk` and `SpaCy` documentations. We are more interested in the general principles here.

```
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 NLKT
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:

```
compare_normalization(u"Our meeting today was worse than yesterday, I'm scared of meeting the clie

Lemmatization:

['our', 'meeting', 'today', 'be', 'bad', 'than', 'yesterday', ',', 'i', 'be', 'scared', 'of', 'mee

Stemming:

['our', 'meet', 'today', 'wa', 'wors', 'than', 'yesterday', ',', 'i', "'m", 'scare', 'of', 'meet',
```

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 noun, and left as-is, while the second occurrence is recognized as verb, and reduced to "meet". In general, lemmatization is a much more involved process than stemming, but usually produces better results 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:

```
# Technicallity: 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 t
# 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 regexp above
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:

```
# transform text_train using CountVectorizer with lemmatization
X_train_lemma = lemma_vect.fit_transform(text_train)
print("X_train_lemma.shape: ", 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: ", X_train.shape)
```

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

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

As you can see from the output above, lemmatization reduced the number of features from 27.272 (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:

```
# build a grid-search using only 1% of the data as 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_))
```

```
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 decomposition methods from Chapter 3. Each of the components we learn then corresponds to one topic, and the coefficient of the components in the representation of a document tells us how much each document is about 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 [footnote: There is another machine learning model called LDA, which is Linear Discriminant Analysis, a linear classification model. This leads to quite some confusion. In this book, LDA refers to 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 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 words like "govenor", "vote", "party" etc, while in a sports article we might expect words like "team", "score" and "season". Each of these groups will likely appear together, while it's less likely that "team" and "govenor" 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 he word "demarcate" and one likes the word "polarize". Another "topic" would then be "words often used by reporter A" and "words often used by reporter B", thought 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 remove words that appear in at

least 20 percent of the documents, and we limit the bag-of-words model to the 10.000 that are most common after removing the top 20 percent:

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

We 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. [footnote: In fact, NMF and LDA solve quite related problems, and we could also use NMF to extract "topics".]

We choose the "batch" learning method, which is somewhat slower than the default, but usually provides better results, and increase "max_iter", which can also lead to better models.

```
from sklearn.decomposition import LatentDirichletAllocation
lda = LatentDirichletAllocation(n_topics=10, learning_method="batch", max_iter=25, random_state=0)
# be 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)
```

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

```
lda.components_.shape
```

```
(10, 10000)
```

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

```
# 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())
```

```
# 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)
```

```
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 comedy, topic 3 might be about tv series, topic 4 seems to caputre some very common words, topic 6 seem to capture children's movies, and topic 8 seems to capture award-related reviews. Using only ten 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 intesting subsets of the data.

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

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

```
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)
```

```
topic 7       topic 16      topic 24      topic 25      topic 28      topic 36      topic 37

--------      --------      --------      --------      --------      --------      --------

thriller      worst         german        car           beautiful     performance   excellent

suspense      awful         hitler        gets          young         role          highly

horror        boring        nazi          guy           old           actor         amazing

atmosphere    horrible      midnight      around        romantic      cast          wonderful

mystery       stupid        joe           down          between       play          truly

house         thing         germany       kill          romance       actors        superb

director      terrible      years         goes          wonderful     performances  actors

quite         script        history       killed        heart         played        brilliant

bit           nothing       new           going         feel          supporting    recommend

de            worse         modesty       house         year          director      quite

performances  waste         cowboy        away          each          oscar         performance

dark          pretty        jewish        head          french        roles         performances

twist         minutes       past          take          sweet         actress       perfect

hitchcock     didn          kirk          another       boy           excellent     drama

tension       actors        young         getting       loved         screen        without

interesting   actually      spanish       doesn         girl          plays         beautiful
```

```
mysterious  re          enterprise  now         relationship  award       human
murder      supposed    von         night       saw           work        moving
ending      mean        nazis       right       both          playing     world
creepy      want        spock       woman       simple        gives       recommended
```

```
topic 45    topic 51    topic 53    topic 54    topic 63    topic 89    topic 97
--------    --------    --------    --------    --------    --------    --------
music       earth       scott       money       funny       dead        didn
song        space       gary        budget      comedy      zombie      thought
songs       planet      streisand   actors      laugh       gore        wasn
rock        superman    star        low         jokes       zombies     ending
band        alien       hart        worst       humor       blood       minutes
soundtrack  world       lundgren    waste       hilarious   horror      got
singing     evil        dolph       10          laughs      flesh       felt
voice       humans      career      give        fun         minutes     part
singer      aliens      sabrina     want        re          body        going
sing        human       role        nothing     funniest    living      seemed
musical     creatures   temple      terrible    laughing    eating      bit
roll        miike       phantom     crap        joke        flick       found
fan         monsters    judy        must        few         budget      though
metal       apes        melissa     reviews     moments     head        nothing
concert     clark       zorro       imdb        guy         gory        lot
playing     burton      gets        director    unfunny     evil        saw
hear        tim         barbra      thing       times       shot        long
fans        outer       cast        believe     laughed     low         interesting
```

```
prince        men           short         am            comedies      fulci         few

especially    moon          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 see, to capture bad reviews, while topic 63 mostly seems to be capturing positive reviews of comedies.

If you want to make further inferences using the topics that were discovered, it is good to confirm the intuition we gained from looking 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 kind of reviews are assigned this topic:

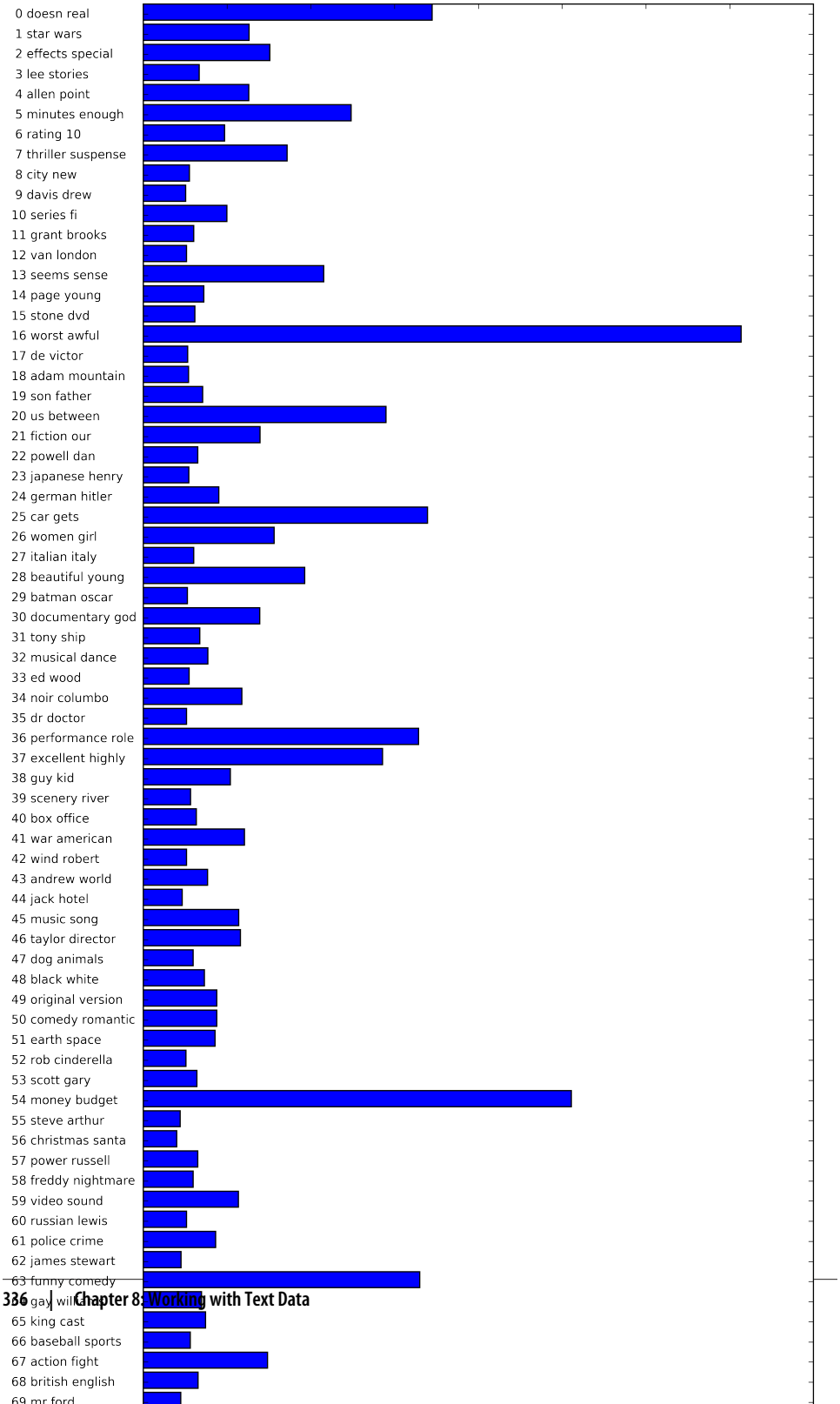
```
# sort by weight of "music" topic 45
music = np.argsort(document_topics100[:, 45])[::-1]
# print the five documents where the topic is most important
for i in music[:10]:
    # pshow first two sentences
    print(b".".join(text_train[i].split(b".")[:2]) + b".\n")
```

```
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 c

b'Hollywood Hotel was the last movie musical that Busby Berkeley directed for Warner Bros. His dir

b"What happens to washed up rock-n-roll stars in the late 1990's? They launch a comeback / reunion

b'As a big-time Prince fan of the last three to four years, I really can\'t believe I\'ve only jus

b"This film is worth seeing alone for Jared Harris' outstanding portrayal of John Lennon. It doesn

b"The funky, yet strictly second-tier British glam-rock band Strange Fruit breaks up at the end of

b"I just finished reading a book on Anita Loos' work and the photo in TCM Magazine of MacDonald in

b'I love this movie!!! Purple Rain came out the year I was born and it has had my heart since I ca

b"This movie is sort of a Carrie meets Heavy Metal. It's about a highschool guy who gets picked on
```

As we can see, this topic covers a wide variety of music-centered reviews, from musicals, to bigraphic 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 comen words:

```
plt.figure(figsize=(10, 30))
plt.barh(np.arange(100), np.sum(document_topics100, axis=0))
topic_names = ["{:>2} ".format(i) + " ".join(words) for i, words in enumerate(feature_names[sortin
```

```
plt.yticks(np.arange(100) + .5, topic_names, ha="left");
ax = plt.gca()
ax.invert_yaxis()
yax = ax.get_yaxis()
yax.set_tick_params(pad=110)
```


0 doesn real
1 star wars
2 effects special
3 lee stories
4 allen point
5 minutes enough
6 rating 10
7 thriller suspense
8 city new
9 davis drew
10 series fi
11 grant brooks
12 van london
13 seems sense
14 page young
15 stone dvd
16 worst awful
17 de victor
18 adam mountain
19 son father
20 us between
21 fiction our
22 powell dan
23 japanese henry
24 german hitler
25 car gets
26 women girl
27 italian italy
28 beautiful young
29 batman oscar
30 documentary god
31 tony ship
32 musical dance
33 ed wood
34 noir columbo
35 dr doctor
36 performance role
37 excellent highly
38 guy kid
39 scenery river
40 box office
41 war american
42 wind robert
43 andrew world
44 jack hotel
45 music song
46 taylor director
47 dog animals
48 black white
49 original version
50 comedy romantic
51 earth space
52 rob cinderella
53 scott gary
54 money budget
55 steve arthur
56 christmas santa
57 power russell
58 freddy nightmare
59 video sound
60 russian lewis
61 police crime
62 james stewart
63 funny comedy
64 gay will
65 king cast
66 baseball sports
67 action fight
68 british english
69 mr ford

The most important topics are 97, which seems to consist mostly of stop-words, possibly with a slight negative direction, topic 16, which is clearly about bad reviews, followed by some genre-specific 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 seems like an interesting discovery, as most reviews are made of some movie-specific comments, and some comments that justify or emphasize the rating.

Topic models like LDA are an interesting methods to understand large text corpora in the absence of labels --- or, as here, even if labels are available. The LDA algorithms 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 intution by looking at the documents in a specific topic.

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 such as spam and fraud detection or sentiment analysis, bag of word representations provide a simple and powerful solution. As so often 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 above, both for supervised and unsupervised tasks. You should take full advantage of this ability when using NLP based methods in practice.

NLP 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 about text processing and natural language processing, we recommend the O'Reilly book Natural Language Processing with Python by Bird, Klein and 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 Manning, Raghavan and 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 above, 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 modelling.

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 Mikolov, Suskever, Chen, Corrado and Dean 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 are *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 Suskever, Vinyals and Le. A more practical tutorial using `tensorflow` framework can found on the tensorflow website [footnote https://www.tensorflow.org/versions/r0.8/tutorials/seq2seq/index.html].