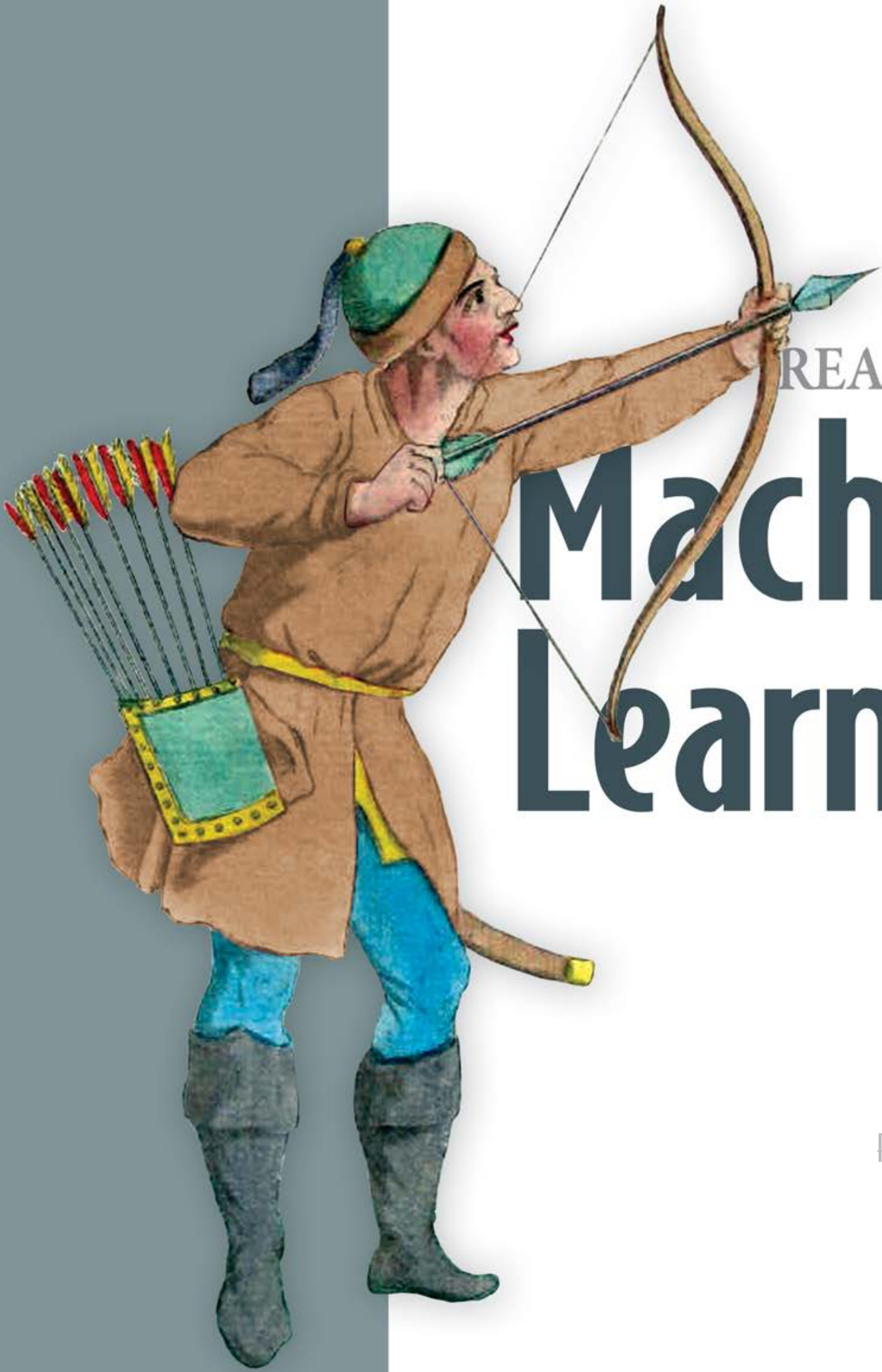
Sample Chapter



Real-World Machine Learning

Henrik Brink
Joseph W. Richards
Mark Fetherolf

Foreword by Beau Cronin

MANNING

Real-World Machine Learning

by Henrik Brink
Joseph W. Richards
Mark Fetherolf

Chapter 8

brief contents

8 Advanced NLP example: movie review sentiment

This chapter covers

- Using a real-world dataset for predicting sentiment from movie reviews
- Exploring possible use cases for this data and the appropriate modeling strategy
- Building an initial model using basic NLP features and optimizing the parameters
- Improving the accuracy of the model by extracting more-advanced NLP features
- Scaling and other deployment aspects of using this model in production

In this chapter, you'll use some of the advanced feature-engineering knowledge acquired in the previous chapter to solve a real-world problem. Specifically, you'll use advanced text and NLP feature-engineering processes to build and optimize a model based on user-submitted reviews of movies.

As always, you'll start by investigating and analyzing the dataset at hand to understand the feature and target columns so you can make the best decisions about which feature-extraction and ML algorithms to use. You'll then build the initial model from the simplest feature-extraction algorithms to see how you can quickly get a useful model with only a few lines of code. Next, you'll dig a little deeper into the library of feature-extraction and ML modeling algorithms to improve the accuracy of the model even further. You'll conclude by exploring various deployment and scalability aspects of putting the model into production.

8.1 Exploring the data and use case

In this chapter, you'll use data from a competition on *Kaggle*—a data-science challenge site where data scientists from around the world work on solving well-defined problems posed by companies to win prizes. You'll work with this data as you learn to use the tools developed in the previous chapters to solve a real-world problem via machine learning.

The data used in this chapter is from the Bag of Words Meets Bags of Popcorn competition (www.kaggle.com/c/word2vec-nlp-tutorial). You need to create an account on the Kaggle platform to download the data, but that's probably a good thing because you might want to try your newly acquired ML skills on a big-prize competition anyway!

In the following sections, we begin by describing the dataset, what the individual columns mean, and how the data was generated. Next, we dive a level deeper, present the data attributes, and make some initial observations about the data that we have. From here, we brainstorm possible use cases that we could solve with the dataset at hand and review the data requirements and real-world implications of each potential use case. Finally, we use this discussion to select a single use case that we'll solve in the remainder of the chapter.

Note that although we structure this section to first describe and explore the data and then to figure out a use case to solve, typically the steps are taken in reverse order. Usually an ML practitioner will start with a use case, hypothesis, or set of questions to answer and then search for and explore data to appropriately solve the problem at hand. This is the preferred methodology, because it forces the practitioner to think hard about the use case and the data required before going "in the weeds" of the dataset. That said, it's not uncommon to be handed a dataset and be asked to build something cool!

8.1.1 A first glance at the dataset

Our dataset consists of written movie reviews from the Internet Movie Database, IMDb (www.imdb.com). The training data consists of 50,000 reviews, selected so that each movie has no more than 30 reviews in the dataset. For each review, the outcome variable is encoded as a binary feature, with the value 1 if the manual IMDb rating for that review is greater than 6, and the value 0 if the rating is less than 5. No reviews in the intermediate ratings of 5–6 are included in the dataset.

The challenge with this dataset is to devise an ML system to learn the patterns and structure of language that constitute positive reviews versus those that constitute negative reviews. Critically, you'll train your model to learn only from the text of the reviews and not from other contextual data such as the movie actors, director, genre, or year of release. Presumably, that data would help the accuracy of your model predictions, but it isn't available in this dataset.

In addition to a training dataset, a separate testing dataset of 25,000 reviews of movies that don't appear in the training dataset is provided. In principle, this set of data could be used to validate the performance of your model and to estimate how well the model will perform when deployed to a real-world production setting. But Kaggle doesn't supply the labels for the testing set. Therefore, you'll construct your own testing set by splitting Kaggle's training set into 70% training and 30% testing.

Note the importance of ensuring that no movies in the training set appear in the testing set.[1] If, for instance, reviews from the same movies were included in both the training and testing sets, then your model could learn which movie titles were good and bad, instead of focusing on the language constituting positivity and negativity. But in production you'll be applying this ML model to new movies, with titles you've never seen. This leakage of movies from the training to the testing set could lead you to believe that your model is better than it is when predicting the sentiment of reviews of new movies. For this reason, we recommend that holdout testing sets always be constructed with temporal cutoffs, so that the testing set consists of instances that are newer than the training instances.

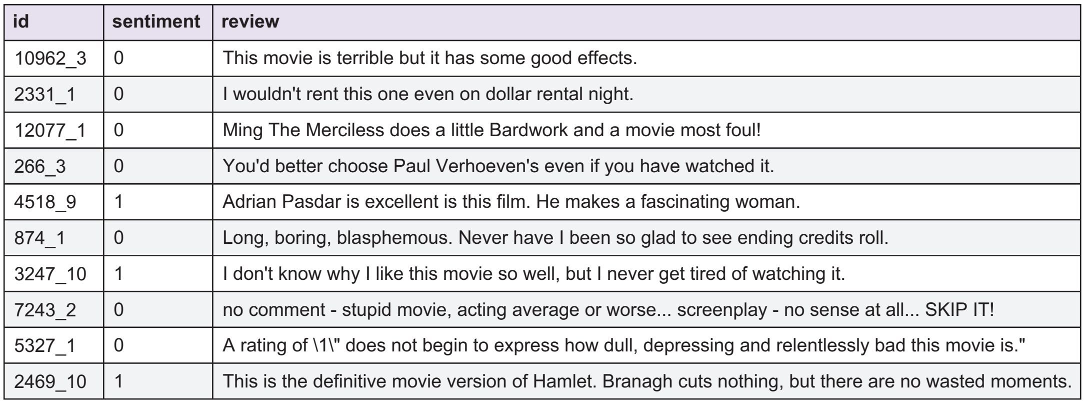
8.1.2 Inspecting the dataset

The individual reviews in this dataset vary in length, from a single sentence up to several pages of text. Because the reviews are pulled from dozens of film critics, the vocabulary can vary dramatically from review to review. The key is to build a machine-learning model that can detect and exploit the differences between the positive and negative reviews so that it can accurately predict the sentiment of new reviews.

The first step of the ML process is to look at the data to see what's there and to begin thinking about the other steps of the ML process, such as model type and featurization. To start the data review process, take a look at the 10 shortest reviews in figure 8.1. Look at the first row (`id = 10962_3`). This particular review demonstrates how nuanced this problem can be: although the review clearly states that the "movie is terrible," it also says that there are "good effects." Despite the use of the word *good*, any person would clearly agree that this is a negative movie review. The challenge now is to teach the ML model that even if positive words such as *good* are used, the use of the phrase "movie is terrible" trumps all!

Similarly, these 10 sample reviews include several examples of negative statements. Phrases such as "never get tired" and "no wasted moments" clearly indicate positive

[1] In our training set, we don't have an indicator of which movie each review describes. Therefore, we make the assumption that the training set is provided presorted by date, and we divide the set so that multiple reviews of the same movie fall together in the training or testing set.



id	sentiment	review
10962_3	0	This movie is terrible but it has some good effects.
2331_1	0	I wouldn't rent this one even on dollar rental night.
12077_1	0	Ming The Merciless does a little Bardwork and a movie most foul!
266_3	0	You'd better choose Paul Verhoeven's even if you have watched it.
4518_9	1	Adrian Pasdar is excellent is this film. He makes a fascinating woman.
874_1	0	Long, boring, blasphemous. Never have I been so glad to see ending credits roll.
3247_10	1	I don't know why I like this movie so well, but I never get tired of watching it.
7243_2	0	no comment - stupid movie, acting average or worse... screenplay - no sense at all... SKIP IT!
5327_1	0	A rating of \1\" does not begin to express how dull, depressing and relentlessly bad this movie is."
2469_10	1	This is the definitive movie version of Hamlet. Branagh cuts nothing, but there are no wasted moments.

Figure 8.1 Ten example reviews in the training set, chosen from the shortest reviews. For each review, you're provided only an ID, the binary sentiment, and the text of the review.

qualities of movies, even if the component words are all negative in nature. This demonstrates that to do well in predicting sentiment, you must combine information across multiple (neighboring) words.

Looking through a few of the other (longer) reviews, it's apparent that these reviews typically consist of verbose, descriptive, flowery language. The language is often sarcastic, ironic, and witty. This makes it a great dataset to demonstrate the power of ML to learn nuanced patterns from real data and to make accurate predictions under uncertainty.

8.1.3 So what's the use case?

Often practitioners of (non-real-world) machine learning dive into a problem without thinking hard about the practical use of their ML model. This is a mistake, because the choice of use case can help determine how you structure the problem and solution, including the following:

- How to encode the target variable (for example, binary versus multiclass versus real value)
- Which evaluation criterion to optimize
- What kinds of learning algorithms to consider
- Which data inputs you should and should not use

So before you get started with ML modeling, you first need to determine what real-world use case you want to solve with this dataset.

For each of three possible use cases, you'll consider the following:

- Why would the use case be valuable?
- What kind of training data would you need?
- What would an appropriate ML modeling strategy be?

- What evaluation metric should you use for your predictions?
- Is the data you have sufficient to solve this use case?

Based on the answers to those questions, you'll choose a single use case, which you'll spend the remainder of the chapter solving.

USE CASE 1: RANKING NEW MOVIES

The first and most obvious use case for a movie review dataset is to automatically rank all new movies based on the text of all their reviews:

- *Why would the use case be valuable?*

  This could be a powerful way to decide which movie to watch this weekend. Scoring individual reviews is one thing, but obviously the more valuable use case is to score each movie on the overall positivity of its reviews. Sites such as Rotten Tomatoes get heavy traffic because of their ability to reliably rate each movie.
- *What kind of training data would you need?*

  The basic necessities would be the review text, an indication of the sentiment of each review, and knowledge about which movie each review refers to. With these three components, building a movie-ranking system would be feasible.
- *What would an appropriate ML modeling strategy be?*

  There are a couple of options: (a) You could treat each movie as an ML instance, aggregate the individual reviews for each movie, and roll up the review sentiment into either an average score or a multiclass model. (b) You could continue to treat each review as an ML instance, score every new review on its positivity, and then assign each new movie its average positivity score. We prefer option (b), because aggregating all reviews for a single movie together could result in some confusing patterns for ML—particularly if the individual reviews are highly polarized! Scoring the individual results and then averaging them into a "metascore" is a more straightforward approach.
- *What evaluation metric should you use for your predictions?*

  Assume here that you have a binary outcome variable for each review and that your ML algorithm assigns a score to each review on its likelihood of being a positive review, which you aggregate into a single score per movie. What you care about here is how closely your score matches the true average rating for that movie (for example, percentage of positive reviews), which could lead you to use a metric such as R^2.

  But you could imagine using a different evaluation metric that focuses more weight at the top of the ranking list. In reality, you're probably interested in a movie ranking in order to pick a flick from the top of the list to see this Saturday. Therefore, you'd instead select a metric that focuses on your ability to get the top of the ranking list right. In this case, you'd select a metric such as the true-positive rate at a small false-positive rate (for example, 5% or 10%).

- *Is the data you have sufficient to solve this use case?*

  Unfortunately, no. You have everything you need except knowledge of which movie each review is describing!

USE CASE 2: RATING EACH REVIEW FROM 1 TO 10

A second possible use case is to auto-rate each review on a scale of 1 to 10 (the IMDb scale) based on the set of user reviews about each movie:

- *Why would the use case be valuable?*

  Any new review could be automatically assigned a rating without any manual reading or scoring. This would cut down on a lot of manual labor that's required to curate the IMDb website and movie ratings; or, if users are providing a score along with their rating, it could provide a more objective score based on the text of the user's review.
- *What kind of training data would you need?*

  Just the text of each review and a score, from 1 to 10, for each review.
- *What would an appropriate ML modeling strategy be?*

  Again, there are two options: (a) Treat the outcome variable as a real-valued number and fit a regression model. (b) Treat the outcome variable as categorical and fit a multiclass classification model. In this case, we much prefer option (a) because, unlike classification, it considers the scores on a numerical scale.
- *What evaluation metric should you use for your predictions?*

  If you choose to run a regression model, the typical regression evaluation metrics such as R^2 or mean squared error are natural choices.
- *Is the data you have sufficient to solve this use case?*

  Again, it's not. You have only a Boolean version of each review (positive versus negative) and not the finely grained numerical score.

USE CASE 3: SEPARATING THE POSITIVE FROM THE NEGATIVE REVIEWS

The final use case to consider is separating all the positive reviews from the rest:

- *Why would the use case be valuable?*

  This use case would represent a less granular version of use case 2, whereby each new review could be automatically classified as positive or negative (instead of scored from 1 to 10). This classification could be useful for IMDb to detect the positive reviews, which it could then promote to its front page or (better yet) sell to movie producers to use as quotes on their movie posters.
- *What kind of training data would you need?*

  Only the review text and the binary positive versus negative indicator.
- *What would an appropriate ML modeling strategy be?*

  You'd fit a binary classification model. From there, you could assign a prediction score for each new review on the likelihood that it's a positive review.

- *What evaluation metric should you use for your predictions?*

  It depends on how you want to use your predictions. If the use case is to automatically pull out the 10 most positive reviews of the week (for example, to use on the IMDb front page), then a good evaluation metric would be the true-positive rate at a very small false-positive rate (for example, 1%). But if the goal is to try to find *all* positive reviews while ignoring the negative reviews (for example, for complete automated sentiment tagging of every review), then a metric such as accuracy or area under the curve (AUC) would be appropriate.
- *Is the data you have sufficient to solve this use case?*

  Finally, yes! You have a training set of the movie review text and the binary sentiment variable. In the remainder of the chapter, you'll build out a machine-learning solution for this use case.

To recap, you first learned the basic details about the dataset: hand-written movie ratings from IMDb. Then, you dove a little deeper to explore some of the patterns and trends in the data. Finally, you considered possible ML use cases. For each use case, you explored the value of a machine-learning solution to the problem, the basic data requirements to build out an ML solution, and how to go about putting together a solution.

Next, you'll build out an ML solution to separate positive from negative movie reviews.

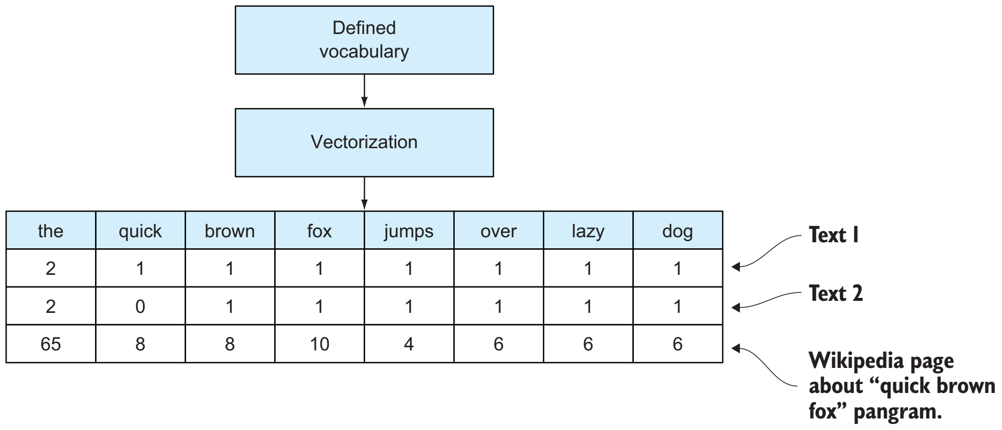
8.2 Extracting basic NLP features and building the initial model

Because the movie review dataset consists of only the review text, you need to use text and natural-language features to build a meaningful dataset for your sentiment model. In the previous chapter, we introduced various methods for extracting features from text, and we use this chapter to discuss various practical aspects of working with ML and free-form text. The steps you'll go through in this section are as follows:

1. Extracting features from movie reviews with the bag-of-words method
2. Building an initial model using the naïve Bayes ML algorithm
3. Improving your bag-of-words features with the tf-idf algorithm
4. Optimizing model parameters

8.2.1 Bag-of-words features

As you may recall from our discussion of NLP features in the previous chapter, we started out with a simple technique to featurize natural-language data: bag of words. This method analyzes the entire corpus of text, builds a dictionary of all words, and translates every instance in the dataset into a list of numbers, counting how many times each word appears in the document. To refresh your memory, let's revisit bag of words in figure 8.2.



the	quick	brown	fox	jumps	over	lazy	dog
2	1	1	1	1	1	1	1
2	0	1	1	1	1	1	1
65	8	8	10	4	6	6	6

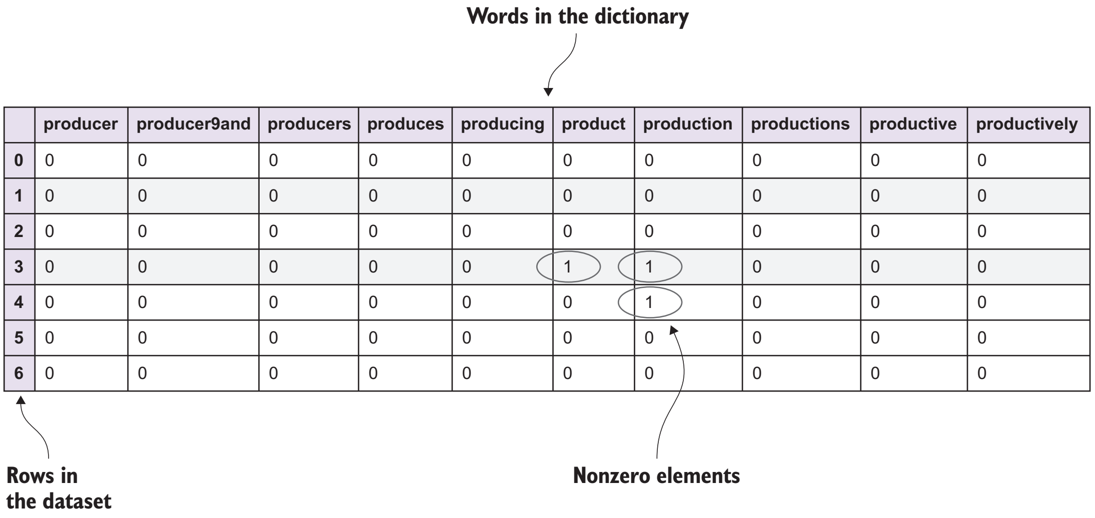
Figure 8.2 The bag-of-words vectorization algorithm. From a dictionary of words, you can transform any new document (for example, Text 1, Text 2 in the figure) into a list of numbers that counts how many times each word appears in the document.

In listing 8.1, you load the dataset, create a 70%–30% train-test split, and use a simple word-count method for extracting features. An important point to realize in this process is that you can't contaminate the bag-of-words dictionary with words from the test set. This is why you split the dataset into training and testing subsets *before* you build the vectorizer dictionary—to get a realistic estimate of the accuracy of the model on previously *unseen* data.

Listing 8.1 Building word-count features from the movie review dataset

```
import pandas
d = pandas.read_csv("movie_reviews/labeledTrainData.tsv", delimiter="\t")   # Loads the data

split = 0.7
d_train = d[:int(split*len(d))]
d_test = d[int((1-split)*len(d)):]      # Splits the data into training and testing subsets

from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()          # Initializes the word-count vectorizer

features = vectorizer.fit_transform(d_train.review)    # Fits the dictionary and generates training set features
test_features = vectorizer.transform(d_test.review)    # Generates features for the testing set
i = 45000
j = 10
words = vectorizer.get_feature_names()[i:i+10]
pandas.DataFrame(features[j:j+7,i:i+10].todense(), columns=words)
```

Take a look at a subset of the features generated in figure 8.3.


Words in the dictionary
Rows in the dataset
Nonzero elements

	producer	producer9and	producers	produces	producing	product	production	productions	productive	productively
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	1	1	0	0	0
4	0	0	0	0	0	0	1	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0

Figure 8.3 A small 7 × 10 subset view of the word-count features that you'll use for building the model. The full dataset is a sparse matrix of size 17,500 × 65,005 (17,500 documents in the training set by 65,005 unique words in the training set). A sparse matrix is useful when most of the values are 0, which is the case in most bag-of-words–based features; in the full dictionary of words, individual words are unlikely to appear in a particular document.

From figure 8.3, it's clear that the dataset consists of mostly zeros with only a few exceptions. We call such a dataset *sparse*, a common attribute of NLP datasets. This has consequences when you want to use the dataset for features in an ML model, something we discuss in the next section before building an actual model to predict the sentiment of reviews.

8.2.2 Building the model with the naïve Bayes algorithm

Now that you have a proper featurized dataset, you can use the features to build the model as usual. For highly sparse datasets like this, some ML algorithms work much better than others. Specifically, some algorithms have built-in support for sparse data, and those algorithms are generally much more efficient, at least in memory usage but often also in CPU usage and time to build. If you inspect the generated feature set from listing 8.1, you'll find that only 0.2% of the cells in the dataset have nonzero elements. Using the dense representation of the dataset would significantly increase the size of the data in memory.

The basics of the naïve Bayes classifier

The naïve Bayes (NB) classifier algorithm is a simple ML algorithm that was created for use in text classification, an area of ML where it can still be competitive with more-advanced general-purpose algorithms. The name stems from the fact that the Bayes formula is applied to the data with very "naïve" assumptions about independence.

This assumption is what usually makes the algorithm less useful for general (dense) problems, because the features are rarely anywhere near independent. For sparse-text features, this assumption still isn't true, but it's true enough for the algorithm to work surprisingly well in practice. The NB classifier is one of the few ML algorithms that's simple enough to derive in a few lines, and we explain some of the highlights in this sidebar.

In this chapter, our goal is to classify a review by finding the probability $p(C_k|x)$ of the review sentiment being "bad" (k = 0) or "good" (k = 1) based on the features x of the instance. In probability theory using the Bayes formula, this can be written like so:

$$p(C_k|x) \sim p(C_k)p(x|C_k)$$

$p(x|C_k)$ is known as the joint probability of the features x if the instance was of class C_k. Because of the independence assumption (the *naïve* part), there's no cross-feature probability, and this becomes simply the product of the probability of each of the features given the class:

$$p(C_k|x) \sim p(C_k)p(x_1|C_k)p(x_2|C_k)p(x_3|C_k)p(x_4|C_k)\ldots$$

$$= p(C_k)\prod_i^N p(x_i|C_k)$$

Because $p(C_k)$ is the marginal class distribution—the overall breakdown of good and bad sentiment reviews—which you can easily find from the data, you only need to figure out what $p(x_i|C_k)$ is. You can read this expression as "the probability of a specific feature for a specific class." For example, you'd expect the probability of having the word *great* in a good-sentiment review being higher than in a bad-sentiment review.

You can imagine learning this from the data by counting the feature (word) presence across all documents in each class. The probability distribution that generates such counts is called the *multinomial* distribution, and $p(x_i|C_k)$ becomes

$$p(x_i|C_k) \sim \prod_i p_{k_i}^{x_i}$$

You use this in the previous equation and move to log space for convenience:

$$\log[p(C_k|x_i)] \sim \log\left[p(C_k)\prod_i p_{k_i}^{x_i}\right]$$

$$= \log[p(C_k)] + \sum_i^n x_i \log(p_{k_i})$$

$$= b + w_k x$$

Here b is $\log[p(C_k)]$ (known from the data), x represents the features of the instance you want to predict, and w_k is $\log(p_{k_i})$—the fraction of times a word appears in a good or bad document, which you'll learn at model build time. Please note that we've left out various constants throughout this calculation, and there are multiple implementation details to consider when coding this algorithm from scratch, but the basics outlined here remain true.

One of the algorithms that works well for classification with sparse natural language processing (NLP) features is the naïve Bayes algorithm, specifically the multinomial (see the sidebar). In the following listing, you build the model on the features from listing 8.1.

Listing 8.2 Building the first review sentiment model using multinomial naïve Bayes

```
from sklearn.naive_bayes import MultinomialNB

model1 = MultinomialNB()
model1.fit(features, d_train.sentiment)
pred1 = model1.predict_proba(test_features)
```

To evaluate the performance of the model, you define a function in listing 8.3 and call it on the initial model predictions. The accuracy metrics that you'll report in this chapter are the general classification accuracy (fraction of correctly classified documents), the receiver operating characteristic (ROC) curve, and the corresponding area under the curve (AUC) number. These were all introduced in chapter 4 and used in many of our examples.

Listing 8.3 Evaluating the initial model

```
from sklearn.metrics import accuracy_score, roc_auc_score, roc_curve

def performance(y_true, pred, color="g", ann=True):
  acc = accuracy_score(y_true, pred[:,1] > 0.5)
  auc = roc_auc_score(y_true, pred[:,1])
  fpr, tpr, thr = roc_curve(y_true, pred[:,1])
  plot(fpr, tpr, color, linewidth="3")
  xlabel("False positive rate")
  ylabel("True positive rate")
  if ann:
    annotate("Acc: %0.2f" % acc, (0.2,0.7), size=14)
    annotate("AUC: %0.2f" % auc, (0.2,0.6), size=14)

performance(d_test.sentiment, pred1)
```

The result of running this code is shown in figure 8.4.

Looking at figure 8.4, you can see that the performance of your bare-bones model isn't bad at all. You classify 88% of the reviews correctly, but you can dial the number of false positives versus true positives up or down, depending on your preference for more noise or better detection rate.

Let's try this with a few new example reviews by passing some text through the vectorizer and model for sentiment predictions:

```
>>> review = "I love this movie"
>>> print model1.predict(vectorizer.transform([review]))[0]
1
```

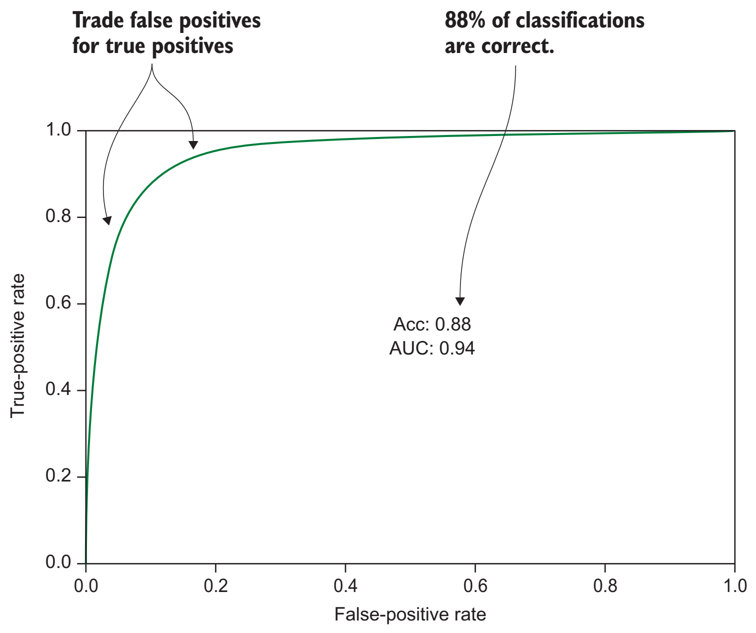


Figure 8.4 ROC curve of the classification performance of the simple bag-of-words model. The classification accuracy—the fraction of correctly classified reviews—as well as the AUC (area under the ROC curve) metrics are printed in the figure. The accuracy shows that you'd expect to correctly classify 88% of the reviews with this model, but by using the ROC curve, you can trade false-positive rate (FPR) for true-positive rate (TPR), and vice versa. If there were many reviews that humans needed to look through based on this classification, you might want to fix the FPR at a low value, which would in turn lower the true-positive detection rate.

A positive sentiment is indicated by 1, so this sounds about right. Let's try another one:

```
>>> review = "This movie is bad"
>>> print model1.predict(vectorizer.transform([review]))[0]
0
```

A negative sentiment is indicated by 0, so again this is indeed correct. Okay, let's try to trick the model:

```
>>> review = "I was going to say something awesome, but I simply can't
     because the movie is so bad."
>>> print model1.predict(vectorizer.transform([review]))[0]
0
```

No luck, the prediction is still correct. Maybe if you introduce more positive words into the negative review?

```
>>> review = "I was going to say something awesome or great or good, but I
     simply can't because the movie is so bad."
>>> print model1.predict(vectorizer.transform([review]))[0]
0
```

Nope, this is one clever model. The word *bad* must have a strong influence on the classification, so perhaps you can cheat the model by using that in a positive review:

```
>>> review = "It might have bad actors, but everything else is good."
>>> print model1.predict(vectorizer.transform([review]))[0]
0
```

Finally, you succeed in somewhat cheating the model. This little exercise is fun, but it also shows the power of the model in understanding arbitrary natural language in the movie review domain. In the next section, you'll try to improve the initial model by going a bit further than our simple word-count features and by finding better values for the parameters of the feature and modeling algorithms.

8.2.3 Normalizing bag-of-words features with the tf-idf algorithm

In the previous chapter, we introduced tf-idf as an upgrade to simple word-count features. In essence, tf-idf normalizes the word counts based on the frequency of how often each word appears across the documents. The main idea is that common words get smaller weighting factors, and relatively rare words get larger weighting factors, which enables you to dig deeper into the (often highly informative) words that appear less often in the dataset.

In this section, you'll use tf-idf for your features to see whether you can gain extra accuracy. The change is easy with scikit-learn, because you simply need to switch out your `CountVectorizer` for a `TfidfVectorizer`. The code is shown in the next listing.

Listing 8.4 Using tf-idf features in your model

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
features = vectorizer.fit_transform(d_train.review)

model2 = MultinomialNB()
model2.fit(features, d_train.sentiment)
pred2 = model2.predict_proba(vectorizer.transform(d_test.review))

performance(d_test.sentiment, pred2)
```

Uses the Tfidf vectorizer to build features

Trains a new naïve Bayes model on the features and makes predictions

Plots the results

The performance of the tf-idf model is shown in figure 8.5. You can see how the tf-idf features improved the model accuracy slightly. Specifically, the ROC curve shows that it should be better at avoiding false positives. Imagine that you had numerous reviews

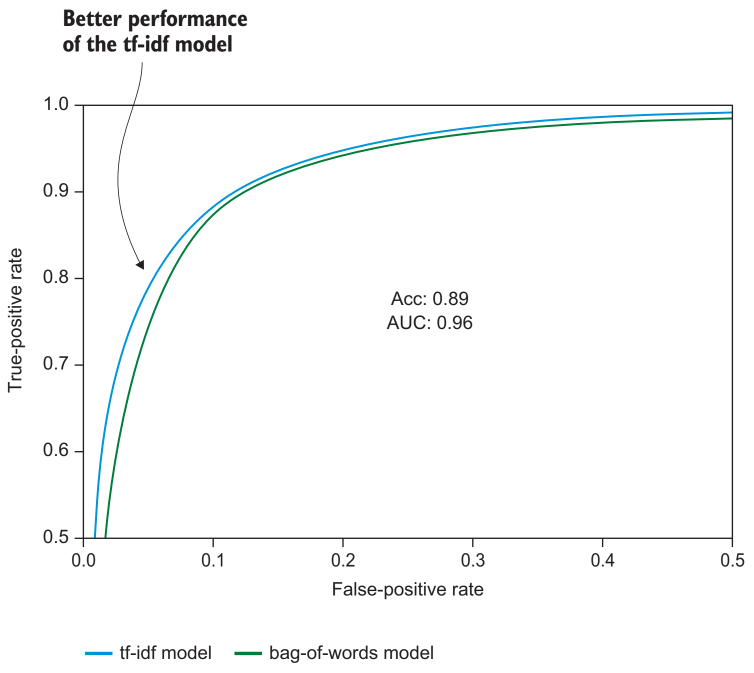


Figure 8.5 ROC curves for the tf-idf model on top of the previous bag-of-words model. You can see a slight improvement in both classification accuracy and AUC (area under the ROC curve). The tf-idf model curve specifically shows improvements in the low FPR range; the model would yield fewer false positives for the same number of correctly classified reviews. If humans were in the classification review loop, you'd have less noise to sift through.

coming in but wanted to flag bad reviews for human inspection. A lower false-positive rate would present fewer reviews to the reviewer that were actually positive, so they could work through the queue faster.

Both our tf-idf NLP feature-extraction algorithm and our naïve Bayes modeling algorithm have knobs that can be turned to tune the algorithm for specific details in the dataset. We call such knobs *hyperparameters*. This comes from the fact that the variables (features) of the model can be considered parameters as well, whereas these algorithm parameters work at a higher level. Before you accept your model performance, it's important that you try different values for these parameters, and this is the topic of the next section.

8.2.4 Optimizing model parameters

The simplest way to find the best parameters of a model is to try to build a bunch of models with different parameters and look at the performance metric of interest. The problem is that you can't assume that the parameters are independent of each other—varying one parameter may affect the optimal value of another. This can be solved in a brute-force way by building a model for any combination of parameters. But if there are many parameters, this quickly becomes intractable, especially if it takes a while to build the model just once. We discussed some solutions in chapter 4, but you'll probably be surprised by how often ML practitioners still rely on the brute-force way. You'll need to build up intuition about which parameters may be more independent of each other and which have the largest effect on which types of dataset. For this exercise, you have three parameters to optimize: two tf-idf parameters (`max_features`, `min_df`) and one naïve Bayes parameter (`nb_alpha`).

The first thing you need is a function that you can call repeatedly to build a model and return the parameters and the metric of interest (in this case, the AUC). The following listing defines this function.

Listing 8.5 Model building method useful for parameter optimization

```
def build_model(max_features=None, min_df=1, nb_alpha=1.0):
  vectorizer = TfidfVectorizer(max_features=max_features, min_df=min_df)
  features = vectorizer.fit_transform(d_train.review)
  model = MultinomialNB(alpha=nb_alpha)
  model.fit(features, d_train.sentiment)
  pred = model.predict_proba(vectorizer.transform(d_test.review))
  return {
    "max_features": max_features,
    "min_df": min_df,
    "nb_alpha": nb_alpha,
    "auc": roc_auc_score(d_test.sentiment, pred[:,1])
  }
```

With the repeatable model building function defined in listing 8.5, you can go ahead and run your optimization pipeline by defining the possible values of your parameters (chosen randomly or by intuition) and run the loop. This is done in the next listing.

Listing 8.6 Parameter optimization loop

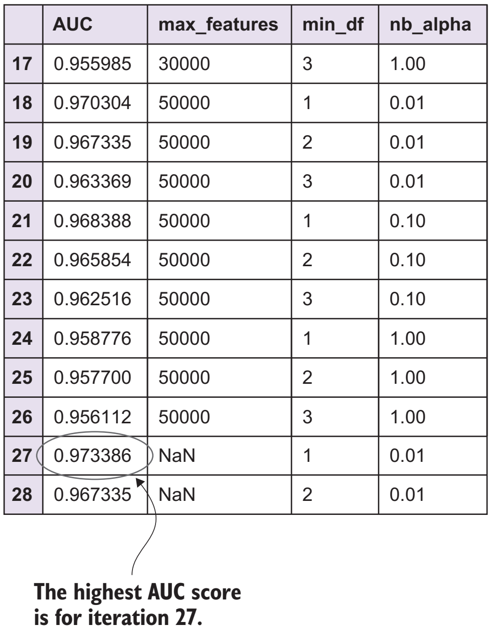
```
from itertools import product

param_values = {                                         Defines parameter
  "max_features": [10000, 30000, 50000, None],           values to try to
  "min_df": [1,2,3],                                     optimize
  "nb_alpha": [0.01, 0.1, 1.0]
}
                                                              For each parameter
results = []                                                  value combination
for p in product(*param_values.values()):                <
  res = build_model(**dict(zip(param_values.keys(), p)))      Builds the model
  results.append( res )                                       and saves the result
  print res
```

The parameters you optimize over are these:

- *`max_features`*—The maximum number of word columns for the tf-idf algorithm to create. From looking at the data, you know that all words amount to about 65,000 columns, so you try out a number of a similar size in a range. `None` specifies to use all words.
- *`min_df`*—The minimum number of times a word must appear in the dataset to be included in the features. This is an example of potential parameter dependency, because the number of words in the dictionary (and hence `max_features`) could be changed by changing `min_df`.
- *`nb_alpha`*—The alpha (smoothing) parameter of the naïve Bayes classifier. This is the only parameter that you can tune on this specific ML algorithm. The values to choose here require a bit more research into what the parameter means and how others have been using it in other circumstances.

The last thing to mention about the code in listing 8.6 is the use of the `product` function from the `itertools` module—a collection of Python functions that makes it easier to work with data. This function is a clever way to generate all combinations of a set of lists (Cartesian product). The results from running the code in listing 8.6 are shown in figure 8.6.

Figure 8.6 shows the output of some of the optimization runs. You had only three parameters with 36 possible value combinations, so this didn't take more than 10 minutes because the naïve Bayes training time is relatively low, but you could easily imagine wanting to try many more values of many more parameters, and the optimization



	AUC	max_features	min_df	nb_alpha
17	0.955985	30000	3	1.00
18	0.970304	50000	1	0.01
19	0.967335	50000	2	0.01
20	0.963369	50000	3	0.01
21	0.968388	50000	1	0.10
22	0.965854	50000	2	0.10
23	0.962516	50000	3	0.10
24	0.958776	50000	1	1.00
25	0.957700	50000	2	1.00
26	0.956112	50000	3	1.00
27	0.973386	NaN	1	0.01
28	0.967335	NaN	2	0.01

The highest AUC score is for iteration 27.

Figure 8.6 A subset of results from the parameter optimization loop. The parameter combination in iteration 27 produces the best model overall.

would take a long time. Another trick for finding the optimal parameters is to start with a broad range of values and then dive more deeply into the optimal value range with subsequent optimization runs over different parameter values. It's clear from the table how different parameters seem to improve the AUC of the model. Iteration 27 had the best results with these values:

- *max_features*—None (all words, default)
- *min_df*—1 (default)
- *nb_alpha*—0.01

So, interestingly, you managed to improve on the model performance quite a bit by finding a better value for the alpha parameter of the naïve Bayes algorithm. Let's look at the evolution of the AUC when varying each parameter (fixing the others at their optimal values) in figure 8.7.

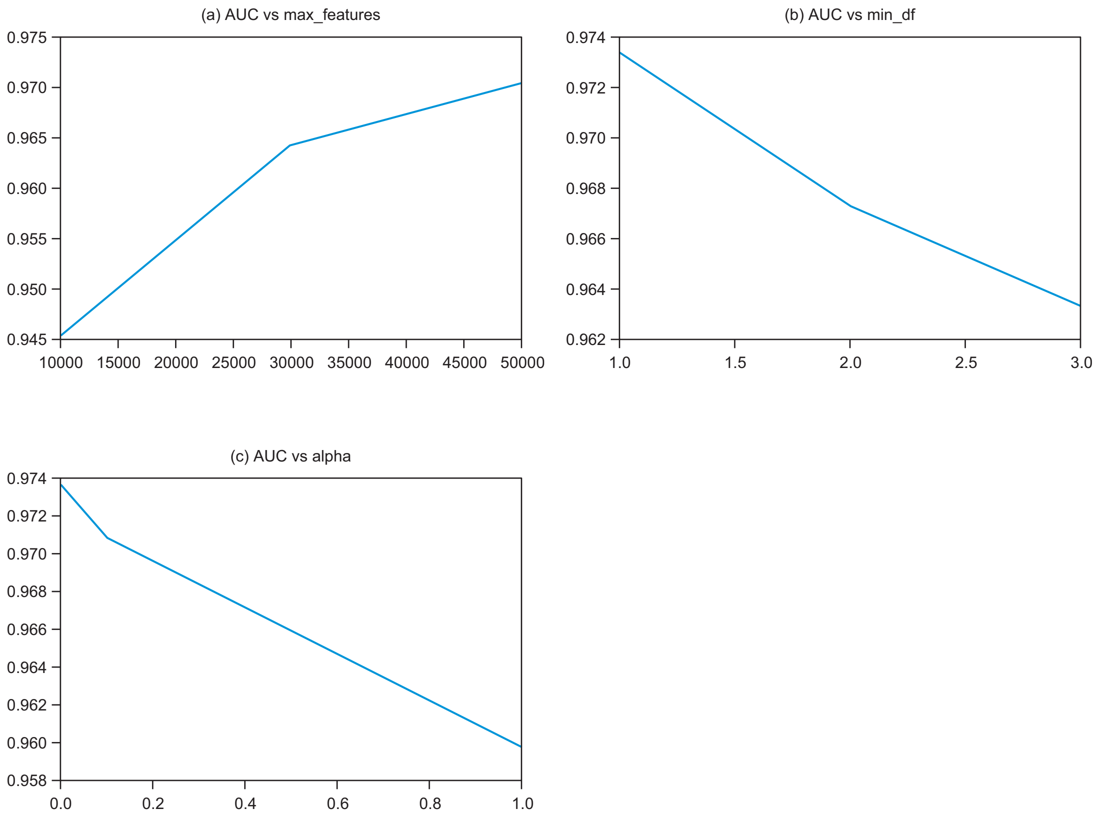


Figure 8.7 The AUC improvements from varying three parameters of the feature and ML algorithms. You can see that (a) a higher `max_features` gives a better AUC, (b) a lower `min_df` gives a better AUC, and (c) a lower alpha gives a better AUC. This doesn't mean that the best values for each of them individually necessarily yields the best combined. The best combined parameters from our optimization run are `max_features=None` (all words, default), `min_df=1` (minimum, default), `alpha=0.01` (main reason for improvement). The best AUC is 0.974. All graphs shown can be reproduced using the code in the accompanying Python notebook.

Each of these plots is only one perspective on the AUC evolution, because you'd need a four-dimensional plot to plot the AUC as a function of all the parameters. But it's still interesting to see how the model responds to varying each value. For instance, the higher the number of features, the better (the largest possible value won). The smaller the number of `min_df`, the better (the smallest possible value won). And then, the smaller the `nb_alpha`, the better. Because this has no theoretical lower limit, this should prompt you to try even lower values in another run. We leave this as an exercise for you (but, anecdotally, we weren't able to find a much better value).

The ROC curve of the optimized model is plotted with the previous models in figure 8.8. You can see a substantial improvement in model performance for both metrics and all points on the ROC curve. This is a great example of how it can pay off to tune your model hyperparameters to gain extra prediction power. One last thing to note here: you could, of course, imagine that new choices of model parameters could, in turn, affect which feature and modeling algorithms (for example, word count versus tf-idf) would perform best, and each algorithm would potentially have a new set of parameters to optimize. To be fully rigorous, you'd need to optimize across all choices of algorithms and their parameters, but this is infeasible for most real-world problems, and the trade-off here is to go through your optimization in milestones. For example, first you fix the NLP algorithm to use and then the ML model, and then you optimize those parameters. Your project could require a different set of milestones—again, you'll develop intuition about these things as you build successive ML models.

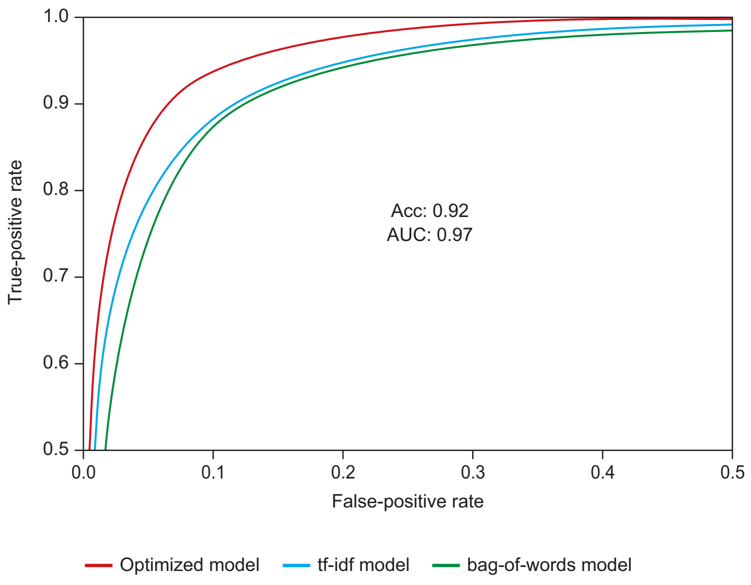


Figure 8.8 The ROC curve of the optimized model versus the previous models. In our test set evaluation, this model seems to be universally better (at every point on the curve), and the expected accuracy increased considerably.

The ROC curves in figure 8.8 conclude our initial modeling experiments. From basic algorithms and very little code, you've managed to build a model with pretty good accuracy on natural-language data alone. In the next section, you'll go a step further in your feature-engineering and modeling efforts and see various aspects of deploying such a model into a real-world production-ready system.

8.3 Advanced algorithms and model deployment considerations

In the previous section, we were concerned with building a model using relatively simple features and ML algorithms. The accuracy of any of the models in that section may have been good enough for our needs. You can try the next idea for optimizing the model, but there's always a trade-off between the time you spend and the potential value brought by incremental improvements in model accuracy. We encourage you to get a handle on the value of each percentage improvement, for example, in the form of saved human-reviewer time, and how much you can afford to spend up front. As you saw, our very first model was certainly capable of understanding review sentiment in many cases and may well have been a good enough model to begin with. Often it's more valuable to put a slightly lower-accuracy model into production and get live feedback from the system if possible.

With that advice out of the way, let's go against it and try to optimize this model a bit further. Next, you'll look into generating features from a new natural-language modeling technique, originally developed by Google: *word2vec*. After you've extracted the word2vec features, you'll switch to the random forest algorithm to better support the new features.

8.3.1 Word2vec features

A relatively new approach to natural language processing has been introduced by Google in the form of the word2vec project. A word2vec model is itself an ML model that's built using deep neural networks, a branch of ML that has recently been producing state-of-the-art results, especially on human-related domains such as natural language, speech, and images.

To build a word2vec model on your training set, you'll use the Gensim NLP library for Python, which has a nice word2vec implementation built in. You previously used Gensim in chapter 7 to work with LDA, another topic model similar to word2vec.

In Gensim, you need to do a bit of extra work to prepare your documents for modeling, because the Gensim algorithms work on sentences (lists of words already split up) instead of arbitrary documents. This can be more work up front, but it also gives you a better understanding of what goes into your model. In listing 8.7, you'll build a simple tokenization function that removes stop words and punctuation characters, and converts all words to lowercase. Note that this was all done automatically in the scikit-learn word vectorizers; we could have used the same functionality or similar functions from the NLTK Python NLP toolkit, but we chose to write it out ourselves here for educational purposes.

Listing 8.7 Document tokenization

```
import re, string

stop_words = set(['all', "she'll", "don't", 'being', 'over', 'through',
'yourselves', 'its', 'before', "he's", "when's", "we've", 'had', 'should',
"he'd", 'to', 'only', "there's", 'those', 'under', 'ours', 'has',
"haven't", 'do', 'them', 'his', "they'll", 'very', "who's", "they'd",
'cannot', "you've", 'they', 'not', 'during', 'yourself', 'him', 'nor',
"we'll", 'did', "they've", 'this', 'she', 'each', "won't", 'where',
"mustn't", "isn't", "i'll", "why's", 'because', "you'd", 'doing', 'some',
'up', 'are', 'further', 'ourselves', 'out', 'what', 'for', 'while',
"wasn't", 'does', "shouldn't", 'above', 'between', 'be', 'we', 'who',
"you're", 'were', 'here', 'hers', "aren't", 'by', 'both', 'about', 'would',
'of', 'could', 'against', "i'd", "weren't", "i'm", 'or', "can't", 'own',
'into', 'whom', 'down', "hadn't", "couldn't", 'your', "doesn't", 'from',
"how's", 'her', 'their', "it's", 'there', 'been', 'why', 'few', 'too',
'themselves', 'was', 'until', 'more', 'himself', "where's", "i've", 'with',
"didn't", "what's", 'but', 'herself', 'than', "here's", 'he', 'me',
"they're", 'myself', 'these', "hasn't", 'below', 'ought', 'theirs', 'my',
"wouldn't", "we'd", 'and', 'then', 'is', 'am', 'it', 'an', 'as', 'itself',
'at', 'have', 'in', 'any', 'if', 'again', 'no', 'that', 'when', 'same',
'how', 'other', 'which', 'you', "shan't", 'our', 'after', "let's", 'most',
'such', 'on', "he'll", 'a', 'off', 'i', "she'd", 'yours', "you'll", 'so',
"we're", "she's", 'the', "that's", 'having', 'once'])

def tokenize(docs):
  pattern = re.compile('[\W_]+', re.UNICODE)
  sentences = []
  for d in docs:
    sentence = d.lower().split(" ")
    sentence = [pattern.sub('', w) for w in sentence]
    sentences.append( [w for w in sentence if w not in stop_words] )
  return sentences
```

Splits the document into words after converting all characters to lowercase

Removes every nonword character, such as punctuation

Removes English stop words

From this function, you can tokenize any list of documents, and you can now proceed to build your first word2vec model. For more information on the parameters of the algorithm, please see the Gensim documentation.[2]

[2] https://radimrehurek.com/gensim/models/word2vec.html

Listing 8.8 Word2vec model

```
from gensim.models.word2vec import Word2Vec

sentences = tokenize(d_train.review)
model = Word2Vec(sentences, size=300, window=10, min_count=1,
   sample=1e-3, workers=2)
model.init_sims(replace=True)
print model['movie']
#> array([ 0.00794919, 0.01277687, -0.04736909, -0.02222243, …])
```

Generates sentences from tokenize function

Builds and normalizes word2vec model

Prints the vector from word2vec model for the word movie

You can see how a single word is represented as a vector (of 300 numbers, in this case). In order to use the word2vec model to generate features for your ML algorithm, you need to convert your reviews into feature vectors. You know how to represent single words as vectors, so a simple idea is to represent a review document (list of words) as the average vector of all the words in the document. In the next listing, you'll build a function to do exactly this.

Listing 8.9 Word2vec featurization

```
def featurize_w2v(model, sentences):
  f = zeros((len(sentences), model.vector_size))
  for i,s in enumerate(sentences):
    for w in s:
      try:
        vec = model[w]
      except KeyError:
        continue
      f[i,:] = f[i,:] + vec
    f[i,:] = f[i,:] / len(s)
  return f
```

Initializes a NumPy array for the feature vectors

Loops over each sentence, add the vectors for each word, and takes the mean

You're now ready to build a model on your newly generated word2vec features. As you may recall from our ML algorithm discussions in section 8.2.2, the naïve Bayes classifier works well with sparse data but not so well with dense data. The word2vec features have indeed converted your documents from the ~65,000 sparse word-count features into only hundreds of dense features. The deep-learning model has learned higher-level topics of the model (listing 8.8), and each document can be represented as a combination of topics (listing 8.9).

8.3.2 Random forest model

The multinomial naïve Bayes algorithm introduced in the previous section is incompatible with the new word2vec features, because they can't be considered generated by a multinomial distribution. You could use other distributions to continue to work with the naïve Bayes algorithm, but you'll instead rely on an old friend of ours: the random forest algorithm. In the following listing, you'll build a 100-tree random forest model on the word2vec features and analyze the performance as usual on the test set.

Listing 8.10 Building a random forest model on the word2vec features

```
features_w2v = featurize_w2v(model, sentences)

model4 = RandomForestClassifier(n_estimators=100, n_jobs=-1)
model4.fit(features_w2v, d_train.sentiment)

test_sentences = tokenize(d_test.review)
test_features_w2v = featurize_w2v(model, test_sentences)
pred4 = model4.predict_proba(test_features_w2v)
performance(d_test.sentiment, pred4, color="c")
```

The performance of the word2vec random forest model is compared to your previous models in figure 8.9. You can see how your new model indeed improves the model accuracy in your chosen evaluation metric and across all points on the ROC curve.

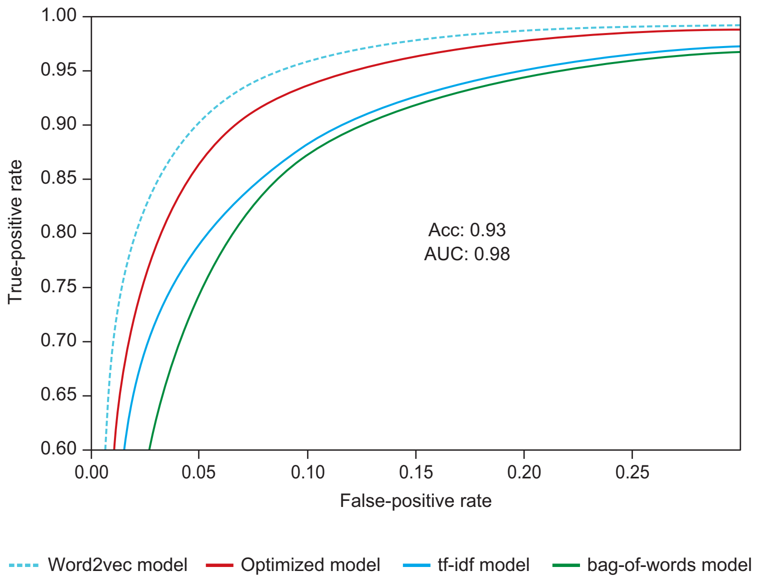


Figure 8.9 The ROC curve of the word2vec model along with previous models. You can see an improvement for all values of the ROC curve, also reflected in the increased accuracy and AUC numbers.

With your final model illustrated in figure 8.9, you're satisfied with the performance and will stop optimization work for now. You could try many more things to improve the accuracy even further. Most likely, not even humans would be capable of correctly classifying the sentiment of all the reviews; there may be some incorrect labels or some reviews for which the sentiment isn't easily understandable.

But the model can likely get much better than what you've achieved so far. We'll leave you, dear reader, with an initial list of things that we would try out, in rough order of priority:

- *Use unlabeled data to build a better topic model.*

  The data section of the Kaggle competition website contains an unlabeled set of reviews that you can use for training. Because you're building a supervised model, they don't seem useful at first. But because you're building a word2vec model that needs to learn the nuances of the world of IMDb movie reviews—and especially the connections between different words and concepts—it would be beneficial to use this data in order to improve your word2vec model that goes into the features of your training set (the one that has labels) before you build the model.
- *Optimize parameters.*

  You saw great improvement in model performance in the initial models of this chapter after finding better values for the hyperparameters of the model. We since introduced a new NLP model (word2vec) and ML algorithm (random forest), so there are many new parameters to optimize.
- *Detect phrases.*

  The Gensim library includes support for detecting phrases in text, such as "New York City," which would be missed in our "dump" word-only tokenization function. The English language tends to include multiword concepts, so this could be an interesting thing to include in your sentence-generation function.
- *Handle multiple languages.*

  If you were uncertain about all the reviews being in a single language (in this case, English), you'd have to deal with multiple languages in various places of the modeling pipeline. First, you'd need to know which language the review was in, or you'd need to detect the language (for which there are several libraries of varying quality available). Then you'd need to use this information in your tokenization process to use different stop words and, potentially, punctuation characters. If you were really unlucky, you'd even have to deal with totally different sentence structures, such as Chinese text, where you can't just split the words when there's a whitespace.

Now, imagine you're satisfied with the model at hand. If this were a real-world use case, you'd want to put the model into production. You should then consider some of the following aspects, depending on the exact use case:

- *How much training data do you have, and does the model get better with more training data?*

  This can affect the choice of ML algorithm because you need to pick a model that scales well with more training data. For example, the naïve Bayes classifier supports partial training, also known as online learning, whereas the random forest algorithm can be difficult to scale to larger datasets.

- *What is the volume of predictions, and do they need to be delivered in real time?*

  We'll talk a great deal more about scaling up predictions with volume and speed in the next chapter, but the takeaway is that this can have consequences for the choice of algorithm and the infrastructure in which it's deployed.

8.4 Summary

In this chapter, you learned how to go end to end on a real machine-learning use case, along with the basics of natural language processing and optimizing model parameters. Key takeaways for this chapter included the following:

- It's essential to focus on the right problem. You should always start by asking, for each possible use case, "What's the value of solving this problem?"
- For each use case, you need to inspect the data and systematically determine whether the data is sufficient to solve the problem at hand.
- Start with simple off-the-shelf algorithms to build an initial model whenever possible. In our example, we predicted review sentiment with almost 90% accuracy.
- Accuracy can be improved by testing and evaluating alternative models and combinations of model parameters.
- There are often trade-offs between different model parameters and evaluation criteria. We looked at how the trade-off between false positive and false negative rates for movie reviews is represented by the model's ROC curve.
- State-of-the-art natural-language and ML modeling techniques like word2vec are examples of how advanced feature engineering may enable you to improve your models.
- Your choice of algorithms may depend on factors other than model accuracy, such as training time and the need to incorporate new data or perform predictions in near-real time.
- In the real world, models can always be improved.

8.5 Terms from this chapter

Word	Definition
word2vec	An NLP modeling framework, initially released by Google and used in many state-of-the-art machine-learning systems involving natural language
hyperparameter optimization	Various techniques for choosing parameters that control ML algorithms' execution to maximize their performance

MACHINE LEARNING/PROGRAMMING

Real-World Machine Learning

Brink • Richards • Fetherolf

Machine learning systems help you find valuable insights and patterns in data, which you'd never recognize with traditional methods. In the real world, ML techniques give you a way to identify trends, forecast behavior, and make fact-based recommendations. It's a hot and growing field, and up-to-speed ML developers are in demand.

Real-World Machine Learning will teach you the concepts and techniques you need to be a successful machine learning practitioner without overdosing you on abstract theory and complex mathematics. By working through immediately relevant examples in Python, you'll build skills in data acquisition and modeling, classification, and regression. You'll also explore the most important tasks like model validation, optimization, scalability, and real-time streaming. When you're done, you'll be ready to successfully build, deploy, and maintain your own powerful ML systems.

What's Inside

- Predicting future behavior
- Performance evaluation and optimization
- Analyzing sentiment and making recommendations

No prior machine learning experience assumed. Readers should know Python.

Henrik Brink, **Joseph Richards**, and **Mark Fetherolf** are experienced data scientists engaged in the daily practice of machine learning.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/real-world-machine-learning

Free eBook
SEE INSERT

"This is that crucial *other* book that many old hands wish they had back in the day."
—From the Foreword by Beau Cronin, 21 Inc.

"A comprehensive guide on how to prepare data for ML and how to choose the appropriate algorithms."
—Michael Lund, iCodeIT

"Very approachable. Great information on data preparation and feature engineering, which are typically ignored."
—Robert Diana
RSI Content Solutions

ISBN-13: 978-1-61729-192-0
ISBN-10: 1-61729-192-7
54999
9 781617 291920

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]