CHAPTER 38

Introducing Scikit-Learn

Several Python libraries provide solid implementations of a range of machine learning algorithms. One of the best known is Scikit-Learn, a package that provides efficient versions of a large number of common algorithms. Scikit-Learn is characterized by a clean, uniform, and streamlined API, as well as by very useful and complete documentation. A benefit of this uniformity is that once you understand the basic use and syntax of Scikit-Learn for one type of model, switching to a new model or algorithm is straightforward.

This chapter provides an overview of the Scikit-Learn API. A solid understanding of these API elements will form the foundation for understanding the deeper practical discussion of machine learning algorithms and approaches in the following chapters.

We will start by covering data representation in Scikit-Learn, then delve into the Estimator API, and finally go through a more interesting example of using these tools for exploring a set of images of handwritten digits.

Data Representation in Scikit-Learn

Machine learning is about creating models from data; for that reason, we'll start by discussing how data can be represented. The best way to think about data within Scikit-Learn is in terms of *tables*.

A basic table is a two-dimensional grid of data, in which the rows represent individual elements of the dataset, and the columns represent quantities related to each of these elements. For example, consider the Iris dataset, famously analyzed by Ronald Fisher in 1936. We can download this dataset in the form of a Pandas `DataFrame` using the Seaborn library, and take a look at the first few items:

```
In [1]: import seaborn as sns
        iris = sns.load_dataset('iris')
        iris.head()
Out[1]:    sepal_length  sepal_width  petal_length  petal_width species
        0           5.1          3.5           1.4          0.2  setosa
        1           4.9          3.0           1.4          0.2  setosa
        2           4.7          3.2           1.3          0.2  setosa
        3           4.6          3.1           1.5          0.2  setosa
        4           5.0          3.6           1.4          0.2  setosa
```

Here each row of the data refers to a single observed flower, and the number of rows is the total number of flowers in the dataset. In general, we will refer to the rows of the matrix as *samples*, and the number of rows as `n_samples`.

Likewise, each column of the data refers to a particular quantitative piece of information that describes each sample. In general, we will refer to the columns of the matrix as *features*, and the number of columns as `n_features`.

The Features Matrix

The table layout makes clear that the information can be thought of as a two-dimensional numerical array or matrix, which we will call the *features matrix*. By convention, this matrix is often stored in a variable named `X`. The features matrix is assumed to be two-dimensional, with shape `[n_samples, n_features]`, and is most often contained in a NumPy array or a Pandas `DataFrame`, though some Scikit-Learn models also accept SciPy sparse matrices.

The samples (i.e., rows) always refer to the individual objects described by the dataset. For example, a sample might represent a flower, a person, a document, an image, a sound file, a video, an astronomical object, or anything else you can describe with a set of quantitative measurements.

The features (i.e., columns) always refer to the distinct observations that describe each sample in a quantitative manner. Features are often real-valued, but may be Boolean or discrete-valued in some cases.

The Target Array

In addition to the feature matrix `X`, we also generally work with a *label* or *target* array, which by convention we will usually call `y`. The target array is usually one-dimensional, with length `n_samples`, and is generally contained in a NumPy array or Pandas `Series`. The target array may have continuous numerical values, or discrete classes/labels. While some Scikit-Learn estimators do handle multiple target values in the form of a two-dimensional, `[n_samples, n_targets]` target array, we will primarily be working with the common case of a one-dimensional target array.

A common point of confusion is how the target array differs from the other feature columns. The distinguishing characteristic of the target array is that it is usually the quantity we want to *predict from the features*: in statistical terms, it is the dependent variable. For example, given the preceding data we may wish to construct a model that can predict the species of flower based on the other measurements; in this case, the `species` column would be considered the target array.

With this target array in mind, we can use Seaborn (discussed in Chapter 36) to conveniently visualize the data (see Figure 38-1).

```
In [2]: %matplotlib inline
        import seaborn as sns
        sns.pairplot(iris, hue='species', height=1.5);
```

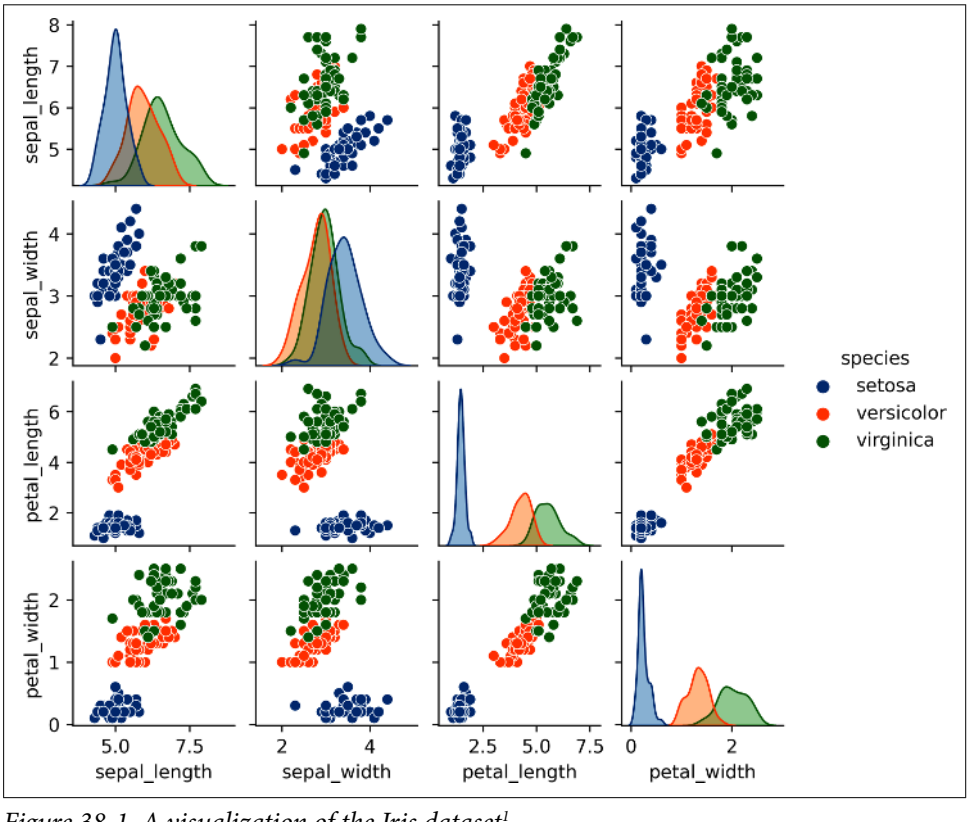

Figure 38-1. A visualization of the Iris dataset[1]

1 A full-size, full-color version of this figure can be found on GitHub.

For use in Scikit-Learn, we will extract the features matrix and target array from the `DataFrame`, which we can do using some of the Pandas `DataFrame` operations discussed in Part III:

```
In [3]: X_iris = iris.drop('species', axis=1)
        X_iris.shape
Out[3]: (150, 4)
```

```
In [4]: y_iris = iris['species']
        y_iris.shape
Out[4]: (150,)
```

To summarize, the expected layout of features and target values is visualized in Figure 38-2.

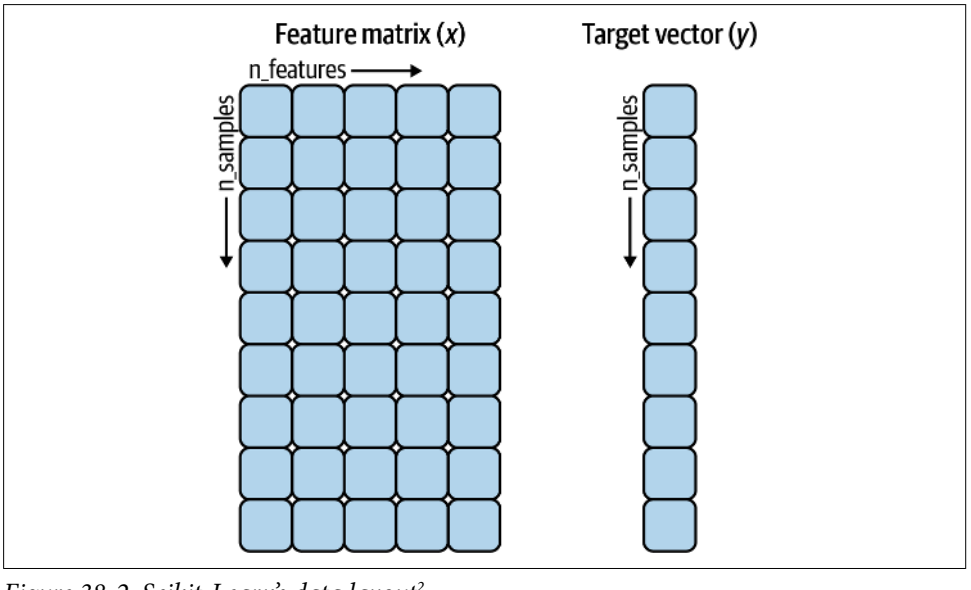

Figure 38-2. Scikit-Learn's data layout[2]

With this data properly formatted, we can move on to consider Scikit-Learn's Estimator API.

The Estimator API

The Scikit-Learn API is designed with the following guiding principles in mind, as outlined in the Scikit-Learn API paper:

2 Code to produce this figure can be found in the online appendix.

Consistency
: All objects share a common interface drawn from a limited set of methods, with consistent documentation.

Inspection
: All specified parameter values are exposed as public attributes.

Limited object hierarchy
: Only algorithms are represented by Python classes; datasets are represented in standard formats (NumPy arrays, Pandas `DataFrame` objects, SciPy sparse matrices) and parameter names use standard Python strings.

Composition
: Many machine learning tasks can be expressed as sequences of more fundamental algorithms, and Scikit-Learn makes use of this wherever possible.

Sensible defaults
: When models require user-specified parameters, the library defines an appropriate default value.

In practice, these principles make Scikit-Learn very easy to use, once the basic principles are understood. Every machine learning algorithm in Scikit-Learn is implemented via the Estimator API, which provides a consistent interface for a wide range of machine learning applications.

Basics of the API

Most commonly, the steps in using the Scikit-Learn Estimator API are as follows:

1. Choose a class of model by importing the appropriate estimator class from Scikit-Learn.
2. Choose model hyperparameters by instantiating this class with desired values.
3. Arrange data into a features matrix and target vector, as outlined earlier in this chapter.
4. Fit the model to your data by calling the `fit` method of the model instance.
5. Apply the model to new data:
   - For supervised learning, often we predict labels for unknown data using the `predict` method.
   - For unsupervised learning, we often transform or infer properties of the data using the `transform` or `predict` method.

We will now step through several simple examples of applying supervised and unsupervised learning methods.

Supervised Learning Example: Simple Linear Regression

As an example of this process, let's consider a simple linear regression—that is, the common case of fitting a line to (x, y) data. We will use the following simple data for our regression example (see Figure 38-3).

```
In [5]: import matplotlib.pyplot as plt
        import numpy as np

        rng = np.random.RandomState(42)
        x = 10 * rng.rand(50)
        y = 2 * x - 1 + rng.randn(50)
        plt.scatter(x, y);
```

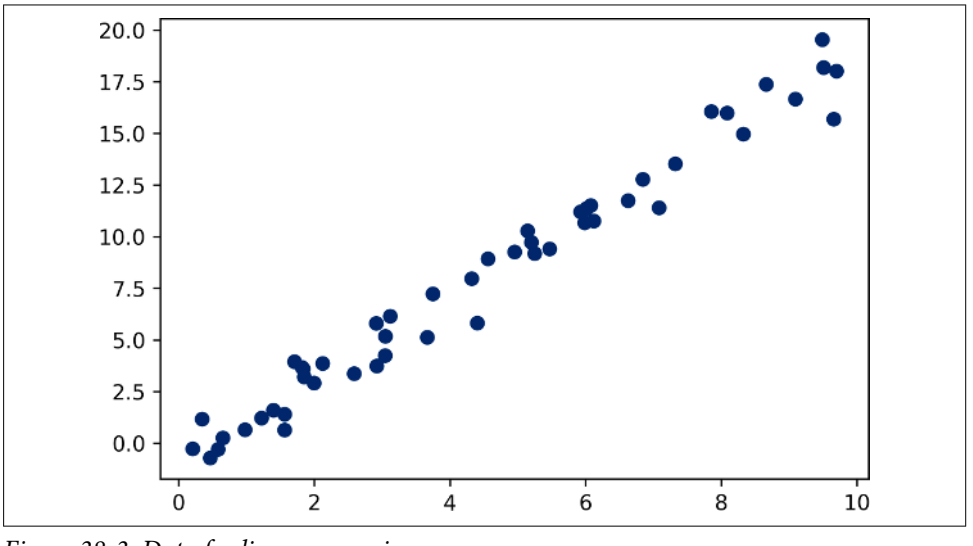

Figure 38-3. Data for linear regression

With this data in place, we can use the recipe outlined earlier. We'll walk through the process in the following sections.

1. Choose a class of model

In Scikit-Learn, every class of model is represented by a Python class. So, for example, if we would like to compute a simple `LinearRegression` model, we can import the linear regression class:

```
In [6]: from sklearn.linear_model import LinearRegression
```

Note that other more general linear regression models exist as well; you can read more about them in the `sklearn.linear_model` module documentation.

2. Choose model hyperparameters

An important point is that *a class of model is not the same as an instance of a model*.

Once we have decided on our model class, there are still some options open to us. Depending on the model class we are working with, we might need to answer one or more questions like the following:

- Would we like to fit for the offset (i.e., *y*-intercept)?
- Would we like the model to be normalized?
- Would we like to preprocess our features to add model flexibility?
- What degree of regularization would we like to use in our model?
- How many model components would we like to use?

These are examples of the important choices that must be made *once the model class is selected*. These choices are often represented as *hyperparameters*, or parameters that must be set before the model is fit to data. In Scikit-Learn, hyperparameters are chosen by passing values at model instantiation. We will explore how you can quantitatively choose hyperparameters in Chapter 39.

For our linear regression example, we can instantiate the `LinearRegression` class and specify that we'd like to fit the intercept using the `fit_intercept` hyperparameter:

```
In [7]: model = LinearRegression(fit_intercept=True)
        model
Out[7]: LinearRegression()
```

Keep in mind that when the model is instantiated, the only action is the storing of these hyperparameter values. In particular, we have not yet applied the model to any data: the Scikit-Learn API makes very clear the distinction between *choice of model* and *application of model to data*.

3. Arrange data into a features matrix and target vector

Previously we examined the Scikit-Learn data representation, which requires a two-dimensional features matrix and a one-dimensional target array. Here our target variable `y` is already in the correct form (a length-`n_samples` array), but we need to massage the data `x` to make it a matrix of size `[n_samples, n_features]`.

In this case, this amounts to a simple reshaping of the one-dimensional array:

```
In [8]: X = x[:, np.newaxis]
        X.shape
Out[8]: (50, 1)
```

4. Fit the model to the data

Now it is time to apply our model to the data. This can be done with the `fit` method of the model:

```
In [9]: model.fit(X, y)
Out[9]: LinearRegression()
```

This `fit` command causes a number of model-dependent internal computations to take place, and the results of these computations are stored in model-specific attributes that the user can explore. In Scikit-Learn, by convention all model parameters that were learned during the `fit` process have trailing underscores; for example in this linear model, we have the following:

```
In [10]: model.coef_
Out[10]: array([1.9776566])
```

```
In [11]: model.intercept_
Out[11]: -0.9033107255311146
```

These two parameters represent the slope and intercept of the simple linear fit to the data. Comparing the results to the data definition, we see that they are close to the values used to generate the data: a slope of 2 and intercept of –1.

One question that frequently comes up regards the uncertainty in such internal model parameters. In general, Scikit-Learn does not provide tools to draw conclusions from internal model parameters themselves: interpreting model parameters is much more a *statistical modeling* question than a *machine learning* question. Machine learning instead focuses on what the model *predicts*. If you would like to dive into the meaning of fit parameters within the model, other tools are available, including the `statsmodels` Python package.

5. Predict labels for unknown data

Once the model is trained, the main task of supervised machine learning is to evaluate it based on what it says about new data that was not part of the training set. In Scikit-Learn, this can be done using the `predict` method. For the sake of this example, our "new data" will be a grid of *x* values, and we will ask what *y* values the model predicts:

```
In [12]: xfit = np.linspace(-1, 11)
```

As before, we need to coerce these *x* values into a `[n_samples, n_features]` features matrix, after which we can feed it to the model:

```
In [13]: Xfit = xfit[:, np.newaxis]
         yfit = model.predict(Xfit)
```

Finally, let's visualize the results by plotting first the raw data, and then this model fit (see Figure 38-4).

```
In [14]: plt.scatter(x, y)
         plt.plot(xfit, yfit);
```

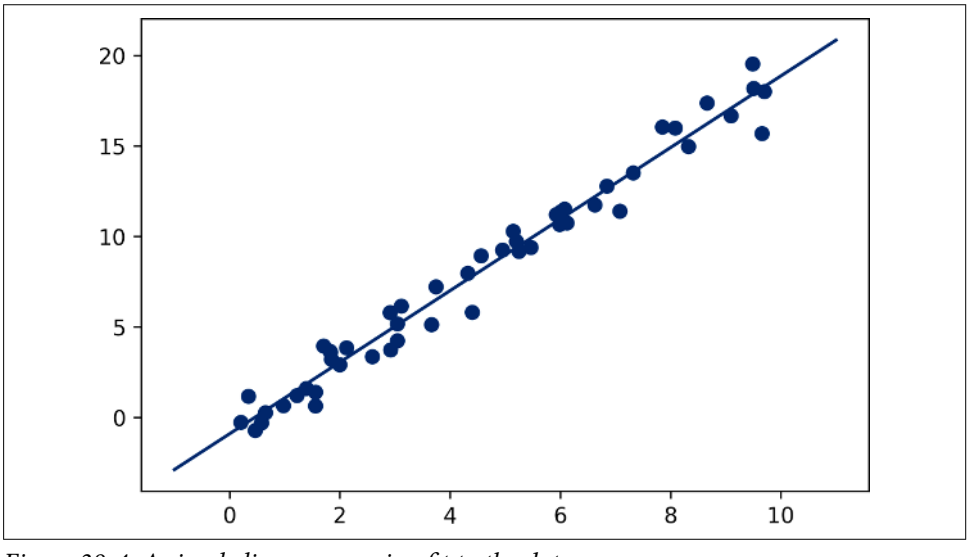

Figure 38-4. A simple linear regression fit to the data

Typically the efficacy of the model is evaluated by comparing its results to some known baseline, as we will see in the next example.

Supervised Learning Example: Iris Classification

Let's take a look at another example of this process, using the Iris dataset we discussed earlier. Our question will be this: given a model trained on a portion of the Iris data, how well can we predict the remaining labels?

For this task, we will use a simple generative model known as *Gaussian naive Bayes*, which proceeds by assuming each class is drawn from an axis-aligned Gaussian distribution (see Chapter 41 for more details). Because it is so fast and has no hyperparameters to choose, Gaussian naive Bayes is often a good model to use as a baseline classification, before exploring whether improvements can be found through more sophisticated models.

We would like to evaluate the model on data it has not seen before, so we will split the data into a *training set* and a *testing set*. This could be done by hand, but it is more convenient to use the `train_test_split` utility function:

```
In [15]: from sklearn.model_selection import train_test_split
         Xtrain, Xtest, ytrain, ytest = train_test_split(X_iris, y_iris,
                                                         random_state=1)
```

With the data arranged, we can follow our recipe to predict the labels:

```
In [16]: from sklearn.naive_bayes import GaussianNB # 1. choose model class
         model = GaussianNB()                       # 2. instantiate model
         model.fit(Xtrain, ytrain)                  # 3. fit model to data
         y_model = model.predict(Xtest)             # 4. predict on new data
```

Finally, we can use the `accuracy_score` utility to see the fraction of predicted labels that match their true values:

```
In [17]: from sklearn.metrics import accuracy_score
         accuracy_score(ytest, y_model)
Out[17]: 0.9736842105263158
```

With an accuracy topping 97%, we see that even this very naive classification algorithm is effective for this particular dataset!

Unsupervised Learning Example: Iris Dimensionality

As an example of an unsupervised learning problem, let's take a look at reducing the dimensionality of the Iris data so as to more easily visualize it. Recall that the Iris data is four-dimensional: there are four features recorded for each sample.

The task of dimensionality reduction centers around determining whether there is a suitable lower-dimensional representation that retains the essential features of the data. Often dimensionality reduction is used as an aid to visualizing data: after all, it is much easier to plot data in two dimensions than in four dimensions or more!

Here we will use *principal component analysis* (PCA; see Chapter 45), which is a fast linear dimensionality reduction technique. We will ask the model to return two components—that is, a two-dimensional representation of the data.

Following the sequence of steps outlined earlier, we have:

```
In [18]: from sklearn.decomposition import PCA  # 1. choose model class
         model = PCA(n_components=2)            # 2. instantiate model
         model.fit(X_iris)                      # 3. fit model to data
         X_2D = model.transform(X_iris)         # 4. transform the data
```

Now let's plot the results. A quick way to do this is to insert the results into the original Iris `DataFrame`, and use Seaborn's `lmplot` to show the results (see Figure 38-5).

```
In [19]: iris['PCA1'] = X_2D[:, 0]
         iris['PCA2'] = X_2D[:, 1]
         sns.lmplot(x="PCA1", y="PCA2", hue='species', data=iris, fit_reg=False);
```

We see that in the two-dimensional representation, the species are fairly well separated, even though the PCA algorithm had no knowledge of the species labels! This suggests to us that a relatively straightforward classification will probably be effective on the dataset, as we saw before.

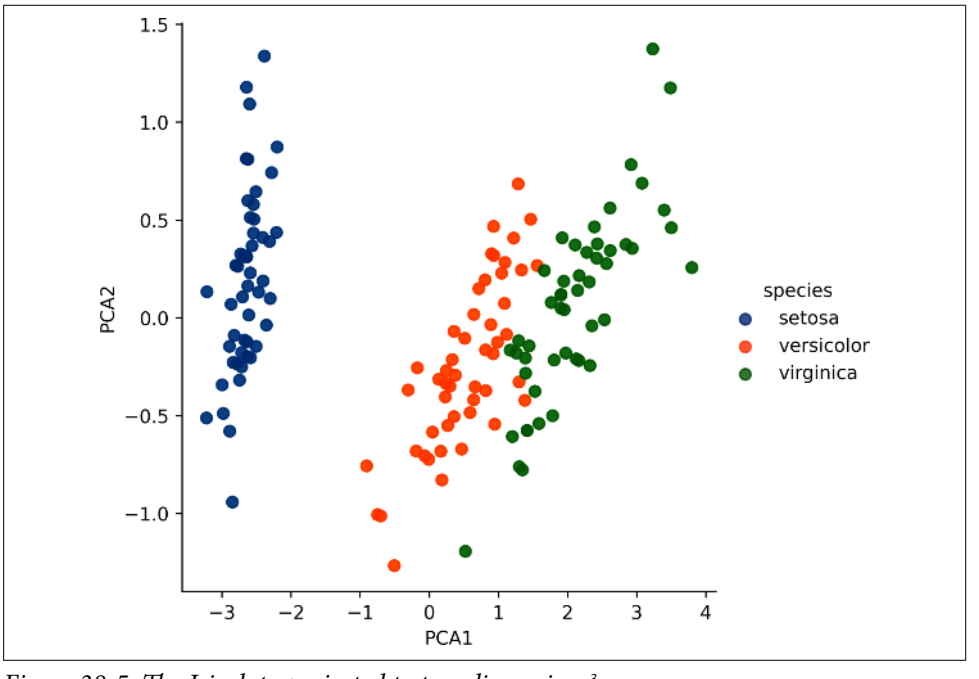

Figure 38-5. The Iris data projected to two dimensions[3]

Unsupervised Learning Example: Iris Clustering

Let's next look at applying clustering to the Iris data. A clustering algorithm attempts to find distinct groups of data without reference to any labels. Here we will use a powerful clustering method called a *Gaussian mixture model* (GMM), discussed in more detail in Chapter 48. A GMM attempts to model the data as a collection of Gaussian blobs.

We can fit the Gaussian mixture model as follows:

```
In [20]: from sklearn.mixture import GaussianMixture      # 1. choose model class
         model = GaussianMixture(n_components=3,
                                 covariance_type='full')  # 2. instantiate model
         model.fit(X_iris)                                # 3. fit model to data
         y_gmm = model.predict(X_iris)                    # 4. determine labels
```

As before, we will add the cluster label to the Iris `DataFrame` and use Seaborn to plot the results (see Figure 38-6).

3 A full-color version of this figure can be found on GitHub.

```
In [21]: iris['cluster'] = y_gmm
         sns.lmplot(x="PCA1", y="PCA2", data=iris, hue='species',
                    col='cluster', fit_reg=False);
```

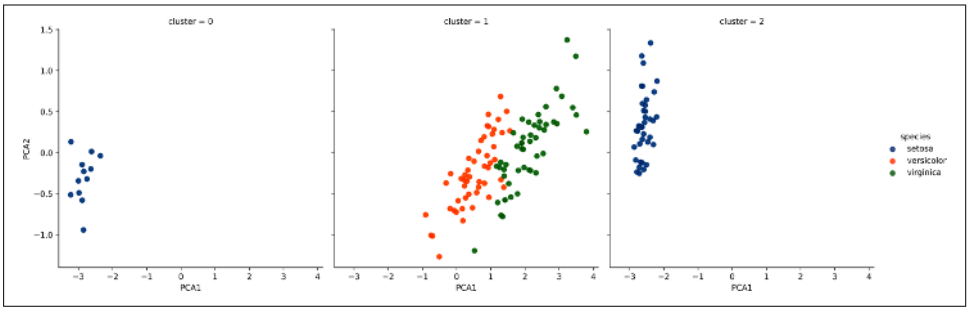

Figure 38-6. k*-means clusters within the Iris data*[4]

By splitting the data by cluster number, we see exactly how well the GMM algorithm has recovered the underlying labels: the *setosa* species is separated perfectly within cluster 0, while there remains a small amount of mixing between *versicolor* and *virginica*. This means that even without an expert to tell us the species labels of the individual flowers, the measurements of these flowers are distinct enough that we could *automatically* identify the presence of these different groups of species with a simple clustering algorithm! This sort of algorithm might further give experts in the field clues as to the relationships between the samples they are observing.

Application: Exploring Handwritten Digits

To demonstrate these principles on a more interesting problem, let's consider one piece of the optical character recognition problem: the identification of handwritten digits. In the wild, this problem involves both locating and identifying characters in an image. Here we'll take a shortcut and use Scikit-Learn's set of preformatted digits, which is built into the library.

Loading and Visualizing the Digits Data

We can use Scikit-Learn's data access interface to take a look at this data:

```
In [22]: from sklearn.datasets import load_digits
         digits = load_digits()
         digits.images.shape
Out[22]: (1797, 8, 8)
```

4 A full-size, full-color version of this figure can be found on GitHub.

The images data is a three-dimensional array: 1,797 samples each consisting of an 8 × 8 grid of pixels. Let's visualize the first hundred of these (see Figure 38-7).

```
In [23]: import matplotlib.pyplot as plt

         fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                                  subplot_kw={'xticks':[], 'yticks':[]},
                                  gridspec_kw=dict(hspace=0.1, wspace=0.1))

         for i, ax in enumerate(axes.flat):
             ax.imshow(digits.images[i], cmap='binary', interpolation='nearest')
             ax.text(0.05, 0.05, str(digits.target[i]),
                     transform=ax.transAxes, color='green')
```

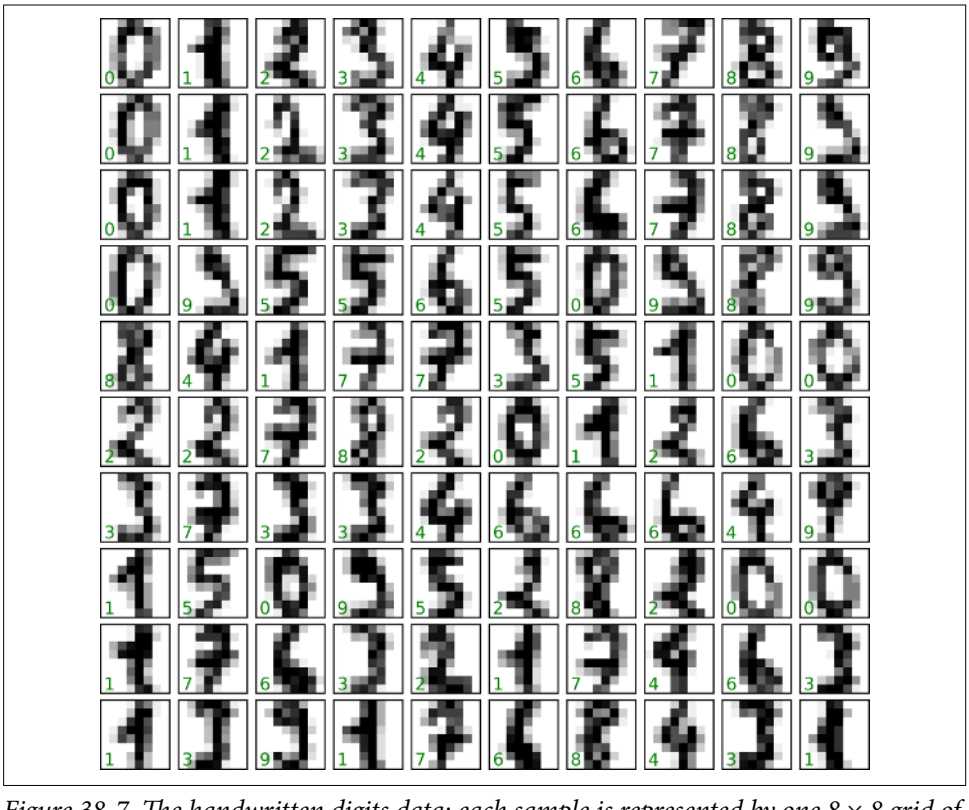

Figure 38-7. The handwritten digits data; each sample is represented by one 8 × 8 grid of pixels

In order to work with this data within Scikit-Learn, we need a two-dimensional, `[n_samples, n_features]` representation. We can accomplish this by treating each pixel in the image as a feature: that is, by flattening out the pixel arrays so that we have a length-64 array of pixel values representing each digit. Additionally, we need the target array, which gives the previously determined label for each digit. These two

quantities are built into the digits dataset under the `data` and `target` attributes, respectively:

```
In [24]: X = digits.data
         X.shape
Out[24]: (1797, 64)
```

```
In [25]: y = digits.target
         y.shape
Out[25]: (1797,)
```

We see here that there are 1,797 samples and 64 features.

Unsupervised Learning Example: Dimensionality Reduction

We'd like to visualize our points within the 64-dimensional parameter space, but it's difficult to effectively visualize points in such a high-dimensional space. Instead, we'll reduce the number of dimensions, using an unsupervised method. Here, we'll make use of a manifold learning algorithm called Isomap (see Chapter 46) and transform the data to two dimensions:

```
In [26]: from sklearn.manifold import Isomap
         iso = Isomap(n_components=2)
         iso.fit(digits.data)
         data_projected = iso.transform(digits.data)
         print(data_projected.shape)
Out[26]: (1797, 2)
```

We see that the projected data is now two-dimensional. Let's plot this data to see if we can learn anything from its structure (see Figure 38-8).

```
In [27]: plt.scatter(data_projected[:, 0], data_projected[:, 1], c=digits.target,
                     edgecolor='none', alpha=0.5,
                     cmap=plt.cm.get_cmap('viridis', 10))
         plt.colorbar(label='digit label', ticks=range(10))
         plt.clim(-0.5, 9.5);
```

This plot gives us some good intuition into how well various numbers are separated in the larger 64-dimensional space. For example, zeros and ones have very little overlap in the parameter space. Intuitively, this makes sense: a zero is empty in the middle of the image, while a one will generally have ink in the middle. On the other hand, there seems to be a more or less continuous spectrum between ones and fours: we can understand this by realizing that some people draw ones with "hats" on them, which causes them to look similar to fours.

Overall, however, despite some mixing at the edges, the different groups appear to be fairly well localized in the parameter space: this suggests that even a very straightforward supervised classification algorithm should perform suitably on the full high-dimensional dataset. Let's give it a try.

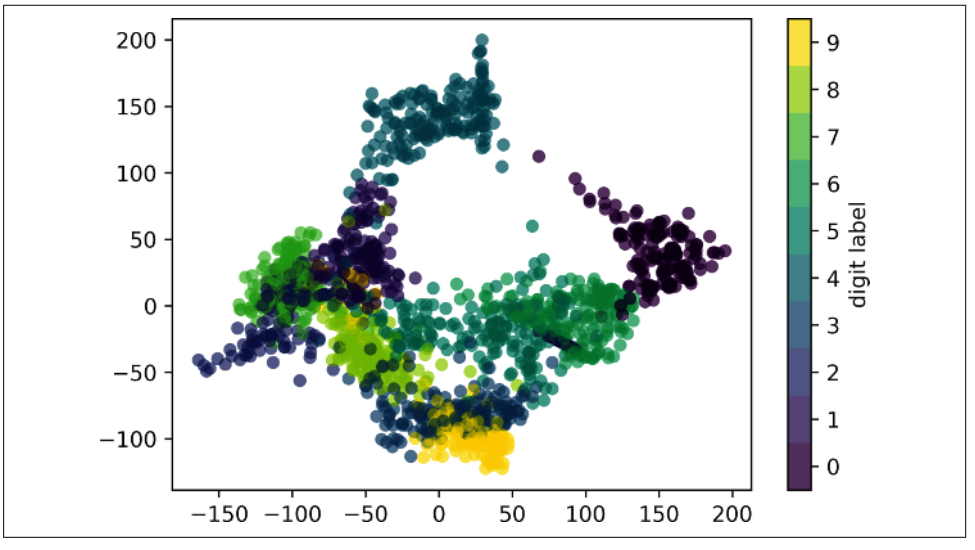

Figure 38-8. An Isomap embedding of the digits data

Classification on Digits

Let's apply a classification algorithm to the digits data. As we did with the Iris data previously, we will split the data into training and testing sets and fit a Gaussian naive Bayes model:

```
In [28]: Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=0)
```

```
In [29]: from sklearn.naive_bayes import GaussianNB
         model = GaussianNB()
         model.fit(Xtrain, ytrain)
         y_model = model.predict(Xtest)
```

Now that we have the model's predictions, we can gauge its accuracy by comparing the true values of the test set to the predictions:

```
In [30]: from sklearn.metrics import accuracy_score
         accuracy_score(ytest, y_model)
Out[30]: 0.8333333333333334
```

With even this very simple model, we find about 83% accuracy for classification of the digits! However, this single number doesn't tell us where we've gone wrong. One nice way to do this is to use the *confusion matrix*, which we can compute with Scikit-Learn and plot with Seaborn (see Figure 38-9).

```
In [31]: from sklearn.metrics import confusion_matrix

         mat = confusion_matrix(ytest, y_model)

         sns.heatmap(mat, square=True, annot=True, cbar=False, cmap='Blues')
```

```
plt.xlabel('predicted value')
plt.ylabel('true value');
```

This shows us where the mislabeled points tend to be: for example, many of the twos here are misclassified as either ones or eights.

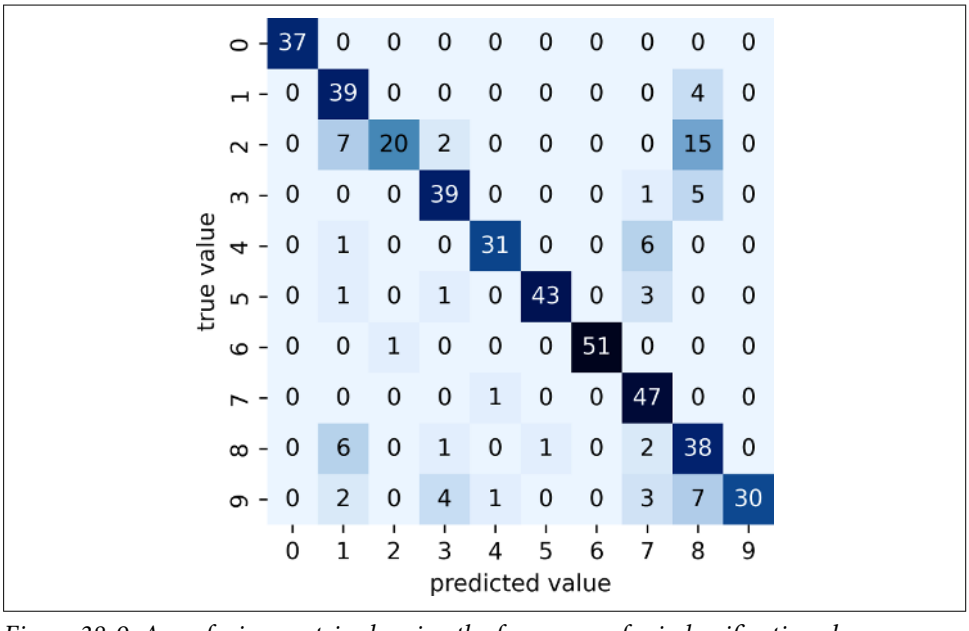

Figure 38-9. A confusion matrix showing the frequency of misclassifications by our classifier

Another way to gain intuition into the characteristics of the model is to plot the inputs again, with their predicted labels. We'll use green for correct labels and red for incorrect labels; see Figure 38-10.

```
In [32]: fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                                  subplot_kw={'xticks':[], 'yticks':[]},
                                  gridspec_kw=dict(hspace=0.1, wspace=0.1))

         test_images = Xtest.reshape(-1, 8, 8)

         for i, ax in enumerate(axes.flat):
             ax.imshow(test_images[i], cmap='binary', interpolation='nearest')
             ax.text(0.05, 0.05, str(y_model[i]),
                     transform=ax.transAxes,
                     color='green' if (ytest[i] == y_model[i]) else 'red')
```

Examining this subset of the data can give us some insight into where the algorithm might be not performing optimally. To go beyond our 83% classification success rate, we might switch to a more sophisticated algorithm such as support vector machines (see Chapter 43), random forests (see Chapter 44), or another classification approach.

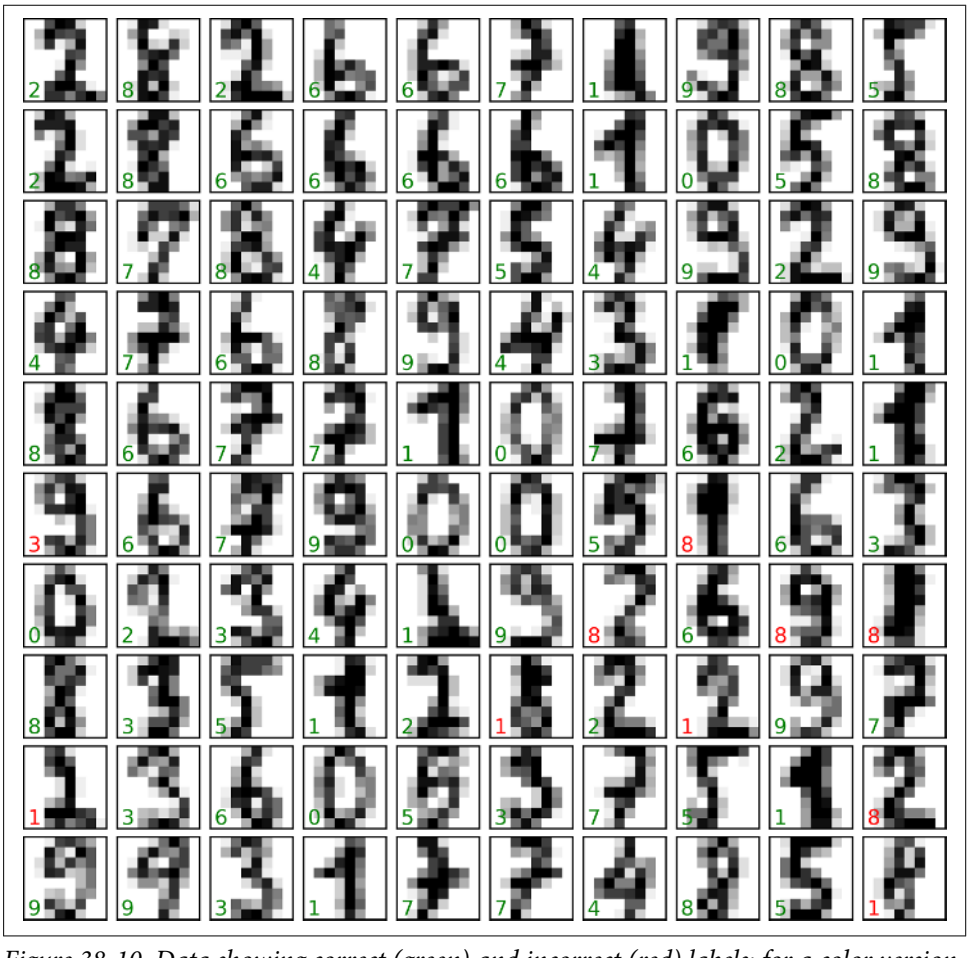

Figure 38-10. Data showing correct (green) and incorrect (red) labels; for a color version of this plot, see the online version of the book

Summary

In this chapter we covered the essential features of the Scikit-Learn data representation and the Estimator API. Regardless of the type of estimator used, the same import/instantiate/fit/predict pattern holds. Armed with this information, you can explore the Scikit-Learn documentation and try out various models on your data.

In the next chapter, we will explore perhaps the most important topic in machine learning: how to select and validate your model.

CHAPTER 39

Hyperparameters and Model Validation

In the previous chapter, we saw the basic recipe for applying a supervised machine learning model:

1. Choose a class of model.
2. Choose model hyperparameters.
3. Fit the model to the training data.
4. Use the model to predict labels for new data.

The first two pieces of this—the choice of model and choice of hyperparameters—are perhaps the most important part of using these tools and techniques effectively. In order to make informed choices, we need a way to *validate* that our model and our hyperparameters are a good fit to the data. While this may sound simple, there are some pitfalls that you must avoid to do this effectively.

Thinking About Model Validation

In principle, model validation is very simple: after choosing a model and its hyperparameters, we can estimate how effective it is by applying it to some of the training data and comparing the predictions to the known values.

This section will first show a naive approach to model validation and why it fails, before exploring the use of holdout sets and cross-validation for more robust model evaluation.

Model Validation the Wrong Way

Let's start with the naive approach to validation using the Iris dataset, which we saw in the previous chapter. We will start by loading the data:

```
In [1]: from sklearn.datasets import load_iris
        iris = load_iris()
        X = iris.data
        y = iris.target
```

Next, we choose a model and hyperparameters. Here we'll use a *k*-nearest neighbors classifier with `n_neighbors=1`. This is a very simple and intuitive model that says "the label of an unknown point is the same as the label of its closest training point":

```
In [2]: from sklearn.neighbors import KNeighborsClassifier
        model = KNeighborsClassifier(n_neighbors=1)
```

Then we train the model, and use it to predict labels for data whose labels we already know:

```
In [3]: model.fit(X, y)
        y_model = model.predict(X)
```

Finally, we compute the fraction of correctly labeled points:

```
In [4]: from sklearn.metrics import accuracy_score
        accuracy_score(y, y_model)
Out[4]: 1.0
```

We see an accuracy score of 1.0, which indicates that 100% of points were correctly labeled by our model! But is this truly measuring the expected accuracy? Have we really come upon a model that we expect to be correct 100% of the time?

As you may have gathered, the answer is no. In fact, this approach contains a fundamental flaw: *it trains and evaluates the model on the same data*. Furthermore, this nearest neighbor model is an *instance-based* estimator that simply stores the training data, and predicts labels by comparing new data to these stored points: except in contrived cases, it will get 100% accuracy every time!

Model Validation the Right Way: Holdout Sets

So what can be done? A better sense of a model's performance can be found by using what's known as a *holdout set*: that is, we hold back some subset of the data from the training of the model, and then use this holdout set to check the model's performance. This splitting can be done using the `train_test_split` utility in Scikit-Learn:

```
In [5]: from sklearn.model_selection import train_test_split
        # split the data with 50% in each set
        X1, X2, y1, y2 = train_test_split(X, y, random_state=0,
                                          train_size=0.5)
```

```
        # fit the model on one set of data
        model.fit(X1, y1)

        # evaluate the model on the second set of data
        y2_model = model.predict(X2)
        accuracy_score(y2, y2_model)
Out[5]: 0.9066666666666666
```

We see here a more reasonable result: the one-nearest-neighbor classifier is about 90% accurate on this holdout set. The holdout set is similar to unknown data, because the model has not "seen" it before.

Model Validation via Cross-Validation

One disadvantage of using a holdout set for model validation is that we have lost a portion of our data to the model training. In the preceding case, half the dataset does not contribute to the training of the model! This is not optimal, especially if the initial set of training data is small.

One way to address this is to use *cross-validation*; that is, to do a sequence of fits where each subset of the data is used both as a training set and as a validation set. Visually, it might look something like Figure 39-1.

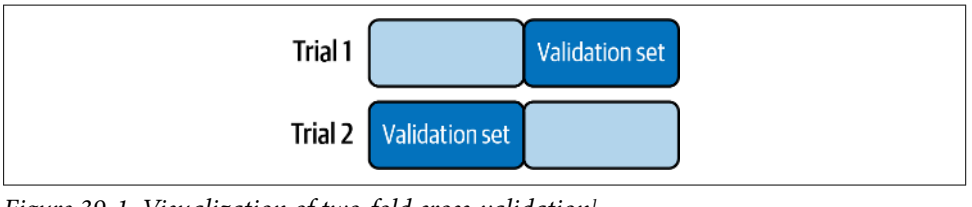

Figure 39-1. Visualization of two-fold cross-validation[1]

Here we do two validation trials, alternately using each half of the data as a holdout set. Using the split data from earlier, we could implement it like this:

```
In [6]: y2_model = model.fit(X1, y1).predict(X2)
        y1_model = model.fit(X2, y2).predict(X1)
        accuracy_score(y1, y1_model), accuracy_score(y2, y2_model)
Out[6]: (0.96, 0.9066666666666666)
```

What comes out are two accuracy scores, which we could combine (by, say, taking the mean) to get a better measure of the global model performance. This particular form of cross-validation is a *two-fold cross-validation*—that is, one in which we have split the data into two sets and used each in turn as a validation set.

1 Code to produce this figure can be found in the online appendix.

We could expand on this idea to use even more trials, and more folds in the data—for example, Figure 39-2 shows a visual depiction of five-fold cross-validation.

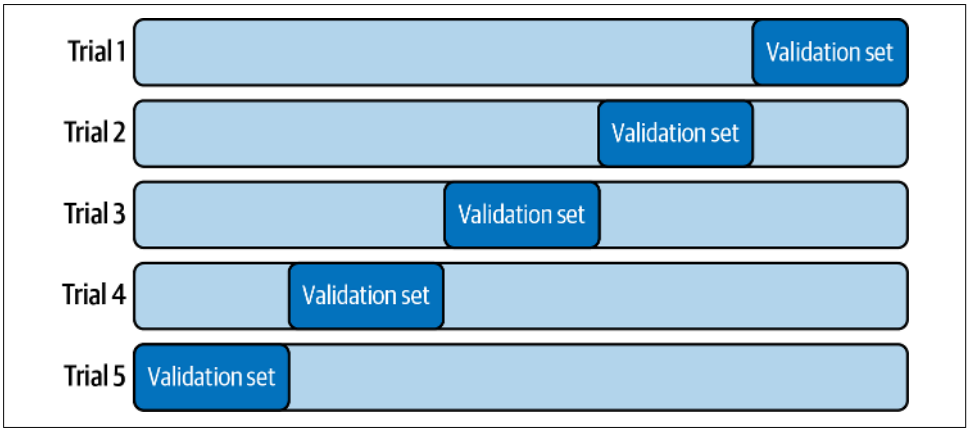

Figure 39-2. Visualization of five-fold cross-validation[2]

Here we split the data into five groups and use each in turn to evaluate the model fit on the other four-fifths of the data. This would be rather tedious to do by hand, but we can use Scikit-Learn's `cross_val_score` convenience routine to do it succinctly:

```
In [7]: from sklearn.model_selection import cross_val_score
        cross_val_score(model, X, y, cv=5)
Out[7]: array([0.96666667, 0.96666667, 0.93333333, 0.93333333, 1.        ])
```

Repeating the validation across different subsets of the data gives us an even better idea of the performance of the algorithm.

Scikit-Learn implements a number of cross-validation schemes that are useful in particular situations; these are implemented via iterators in the `model_selection` module. For example, we might wish to go to the extreme case in which our number of folds is equal to the number of data points: that is, we train on all points but one in each trial. This type of cross-validation is known as *leave-one-out* cross validation, and can be used as follows:

```
In [8]: from sklearn.model_selection import LeaveOneOut
        scores = cross_val_score(model, X, y, cv=LeaveOneOut())
        scores
Out[8]: array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
               1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
               1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
               1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
               1., 1., 0., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 1.,
```

2 Code to produce this figure can be found in the online appendix.

```
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Because we have 150 samples, the leave-one-out cross-validation yields scores for 150 trials, and each score indicates either a successful (1.0) or an unsuccessful (0.0) prediction. Taking the mean of these gives an estimate of the error rate:

```
In [9]: scores.mean()
Out[9]: 0.96
```

Other cross-validation schemes can be used similarly. For a description of what is available in Scikit-Learn, use IPython to explore the `sklearn.model_selection` submodule, or take a look at Scikit-Learn's cross-validation documentation.

Selecting the Best Model

Now that we've explored the basics of validation and cross-validation, we will go into a little more depth regarding model selection and selection of hyperparameters. These issues are some of the most important aspects of the practice of machine learning, but I find that this information is often glossed over in introductory machine learning tutorials.

Of core importance is the following question: *if our estimator is underperforming, how should we move forward?* There are several possible answers:

- Use a more complicated/more flexible model.
- Use a less complicated/less flexible model.
- Gather more training samples.
- Gather more data to add features to each sample.

The answer to this question is often counterintuitive. In particular, sometimes using a more complicated model will give worse results, and adding more training samples may not improve your results! The ability to determine what steps will improve your model is what separates the successful machine learning practitioners from the unsuccessful.

The Bias-Variance Trade-off

Fundamentally, finding "the best model" is about finding a sweet spot in the trade-off between *bias* and *variance*. Consider Figure 39-3, which presents two regression fits to the same dataset.

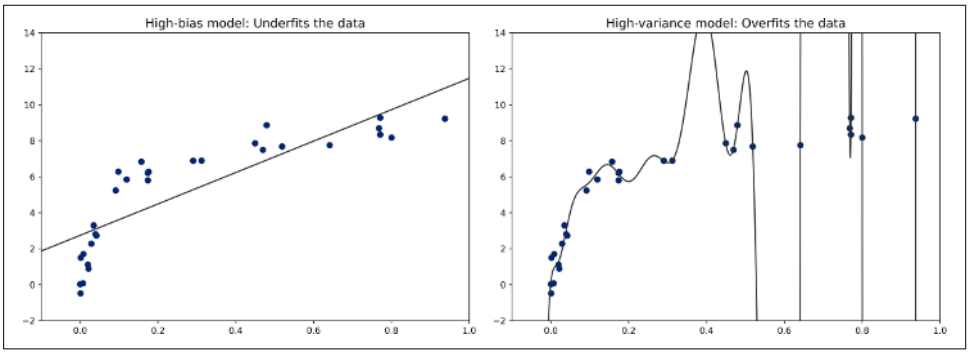

Figure 39-3. High-bias and high-variance regression models[3]

It is clear that neither of these models is a particularly good fit to the data, but they fail in different ways.

The model on the left attempts to find a straight-line fit through the data. Because in this case a straight line cannot accurately split the data, the straight-line model will never be able to describe this dataset well. Such a model is said to *underfit* the data: that is, it does not have enough flexibility to suitably account for all the features in the data. Another way of saying this is that the model has high bias.

The model on the right attempts to fit a high-order polynomial through the data. Here the model fit has enough flexibility to nearly perfectly account for the fine features in the data, but even though it very accurately describes the training data, its precise form seems to be more reflective of the particular noise properties of the data than of the intrinsic properties of whatever process generated that data. Such a model is said to *overfit* the data: that is, it has so much flexibility that the model ends up accounting for random errors as well as the underlying data distribution. Another way of saying this is that the model has high variance.

To look at this in another light, consider what happens if we use these two models to predict the y-values for some new data. In the plots in Figure 39-4, the red/lighter points indicate data that is omitted from the training set.

3 Code to produce this figure can be found in the online appendix.

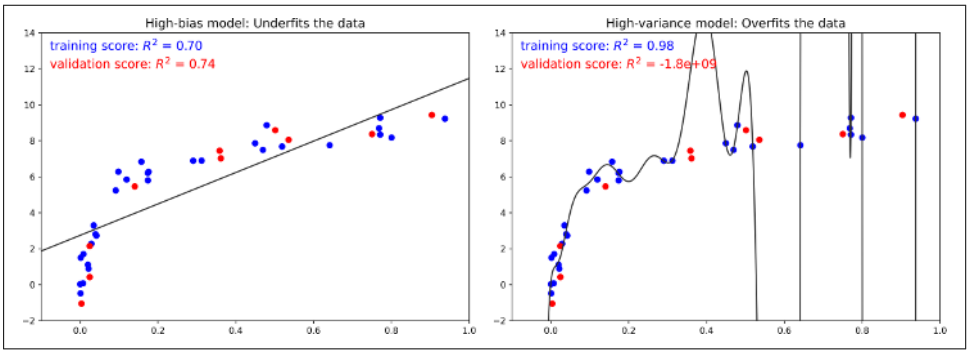

Figure 39-4. Training and validation scores in high-bias and high-variance models[4]

The score here is the R^2 score, or coefficient of determination, which measures how well a model performs relative to a simple mean of the target values. $R^2 = 1$ indicates a perfect match, $R^2 = 0$ indicates the model does no better than simply taking the mean of the data, and negative values mean even worse models. From the scores associated with these two models, we can make an observation that holds more generally:

- For high-bias models, the performance of the model on the validation set is similar to the performance on the training set.
- For high-variance models, the performance of the model on the validation set is far worse than the performance on the training set.

If we imagine that we have some ability to tune the model complexity, we would expect the training score and validation score to behave as illustrated in Figure 39-5, often called a *validation curve*, and we see the following features:

- The training score is everywhere higher than the validation score. This is generally the case: the model will be a better fit to data it has seen than to data it has not seen.
- For very low model complexity (a high-bias model), the training data is underfit, which means that the model is a poor predictor both for the training data and for any previously unseen data.
- For very high model complexity (a high-variance model), the training data is overfit, which means that the model predicts the training data very well, but fails for any previously unseen data.

4 Code to produce this figure can be found in the online appendix.

- For some intermediate value, the validation curve has a maximum. This level of complexity indicates a suitable trade-off between bias and variance.

The means of tuning the model complexity varies from model to model; when we discuss individual models in depth in later chapters, we will see how each model allows for such tuning.

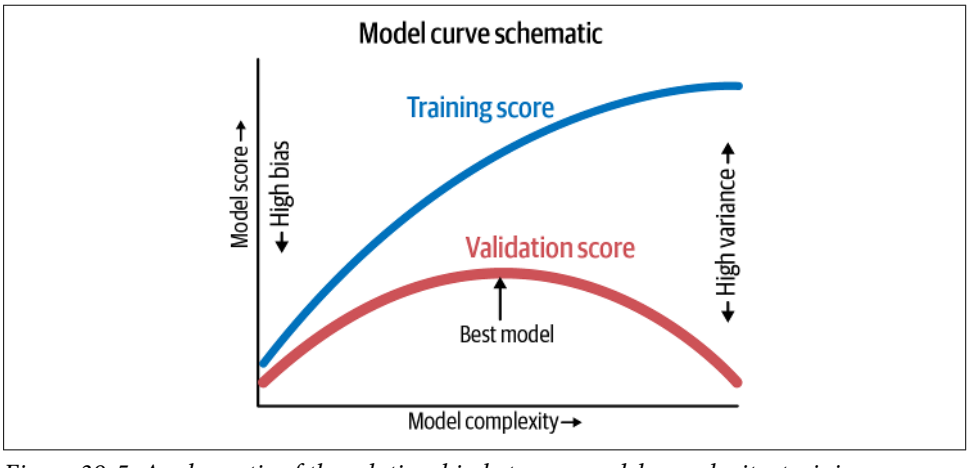

Figure 39-5. A schematic of the relationship between model complexity, training score, and validation score[5]

Validation Curves in Scikit-Learn

Let's look at an example of using cross-validation to compute the validation curve for a class of models. Here we will use a *polynomial regression* model, a generalized linear model in which the degree of the polynomial is a tunable parameter. For example, a degree-1 polynomial fits a straight line to the data; for model parameters a and b:

$$y = ax + b$$

A degree-3 polynomial fits a cubic curve to the data; for model parameters a, b, c, d:

$$y = ax^3 + bx^2 + cx + d$$

5 Code to produce this figure can be found in the online appendix.

We can generalize this to any number of polynomial features. In Scikit-Learn, we can implement this with a linear regression classifier combined with the polynomial preprocessor. We will use a *pipeline* to string these operations together (we will discuss polynomial features and pipelines more fully in Chapter 40):

```
In [10]: from sklearn.preprocessing import PolynomialFeatures
         from sklearn.linear_model import LinearRegression
         from sklearn.pipeline import make_pipeline

         def PolynomialRegression(degree=2, **kwargs):
             return make_pipeline(PolynomialFeatures(degree),
                                  LinearRegression(**kwargs))
```

Now let's create some data to which we will fit our model:

```
In [11]: import numpy as np

         def make_data(N, err=1.0, rseed=1):
             # randomly sample the data
             rng = np.random.RandomState(rseed)
             X = rng.rand(N, 1) ** 2
             y = 10 - 1. / (X.ravel() + 0.1)
             if err > 0:
                 y += err * rng.randn(N)
             return X, y

         X, y = make_data(40)
```

We can now visualize our data, along with polynomial fits of several degrees (see Figure 39-6).

```
In [12]: %matplotlib inline
         import matplotlib.pyplot as plt
         plt.style.use('seaborn-whitegrid')

         X_test = np.linspace(-0.1, 1.1, 500)[:, None]

         plt.scatter(X.ravel(), y, color='black')
         axis = plt.axis()
         for degree in [1, 3, 5]:
             y_test = PolynomialRegression(degree).fit(X, y).predict(X_test)
             plt.plot(X_test.ravel(), y_test, label='degree={0}'.format(degree))
         plt.xlim(-0.1, 1.0)
         plt.ylim(-2, 12)
         plt.legend(loc='best');
```

The knob controlling model complexity in this case is the degree of the polynomial, which can be any nonnegative integer. A useful question to answer is this: what degree of polynomial provides a suitable trade-off between bias (underfitting) and variance (overfitting)?

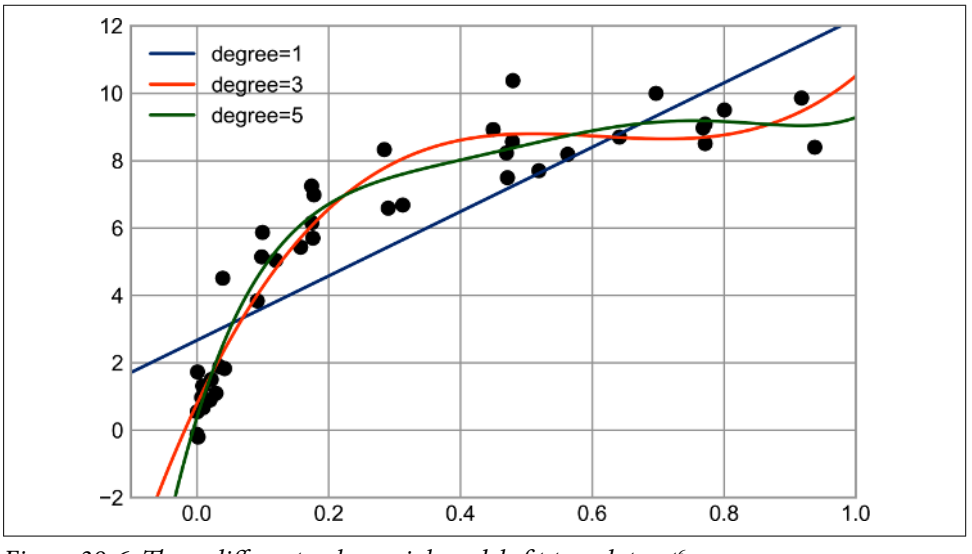

Figure 39-6. Three different polynomial models fit to a dataset[6]

We can make progress in this by visualizing the validation curve for this particular data and model; this can be done straightforwardly using the `validation_curve` convenience routine provided by Scikit-Learn. Given a model, data, parameter name, and a range to explore, this function will automatically compute both the training score and the validation score across the range (see Figure 39-7).

```
In [13]: from sklearn.model_selection import validation_curve
         degree = np.arange(0, 21)
         train_score, val_score = validation_curve(
             PolynomialRegression(), X, y,
             param_name='polynomialfeatures__degree',
             param_range=degree, cv=7)

         plt.plot(degree, np.median(train_score, 1),
                  color='blue', label='training score')
         plt.plot(degree, np.median(val_score, 1),
                  color='red', label='validation score')
         plt.legend(loc='best')
         plt.ylim(0, 1)
         plt.xlabel('degree')
         plt.ylabel('score');
```

6 A full-color version of this figure can be found on GitHub.

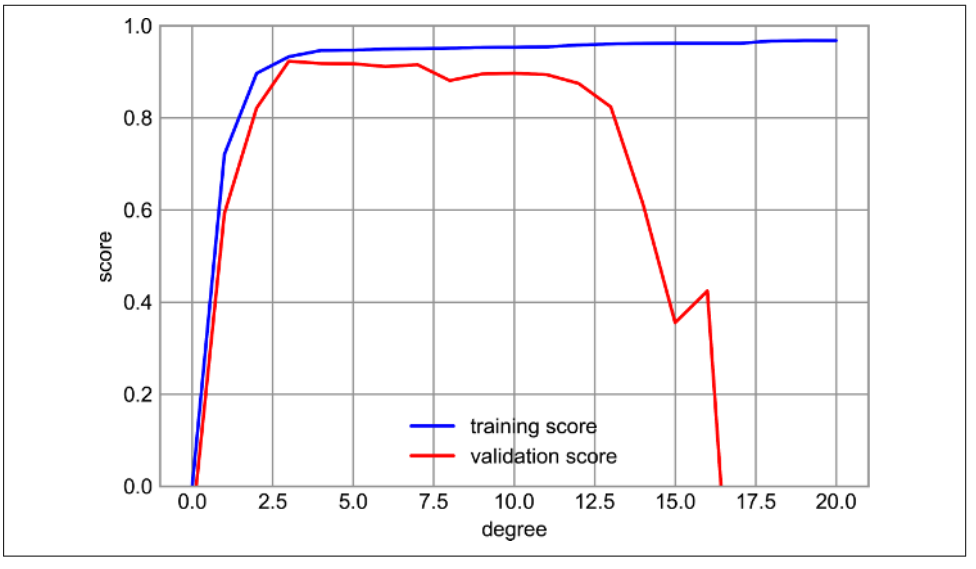

Figure 39-7. The validation curves for the data in Figure 39-9

This shows precisely the qualitative behavior we expect: the training score is everywhere higher than the validation score, the training score is monotonically improving with increased model complexity, and the validation score reaches a maximum before dropping off as the model becomes overfit.

From the validation curve, we can determine that the optimal trade-off between bias and variance is found for a third-order polynomial. We can compute and display this fit over the original data as follows (see Figure 39-8).

```
In [14]: plt.scatter(X.ravel(), y)
         lim = plt.axis()
         y_test = PolynomialRegression(3).fit(X, y).predict(X_test)
         plt.plot(X_test.ravel(), y_test);
         plt.axis(lim);
```

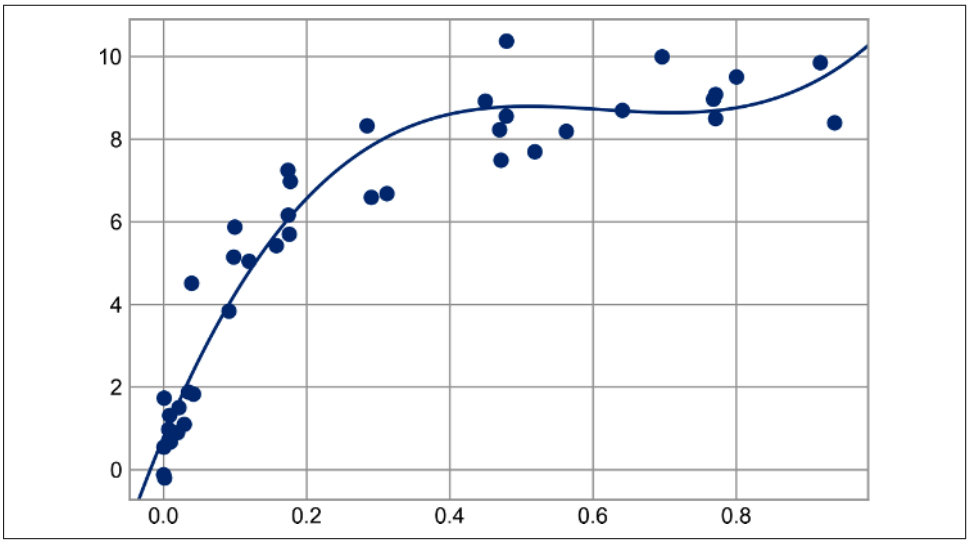

Figure 39-8. The cross-validated optimal model for the data in Figure 39-6

Notice that finding this optimal model did not actually require us to compute the training score, but examining the relationship between the training score and validation score can give us useful insight into the performance of the model.

Learning Curves

One important aspect of model complexity is that the optimal model will generally depend on the size of your training data. For example, let's generate a new dataset with five times as many points (see Figure 39-9).

```
In [15]: X2, y2 = make_data(200)
         plt.scatter(X2.ravel(), y2);
```

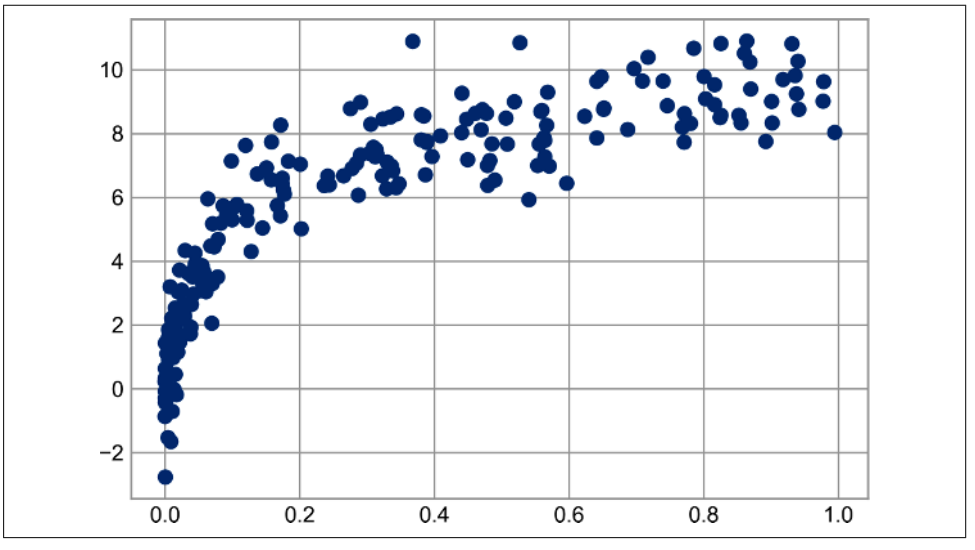

Figure 39-9. Data to demonstrate learning curves

Now let's duplicate the preceding code to plot the validation curve for this larger dataset; for reference, we'll overplot the previous results as well (see Figure 39-10).

```
In [16]: degree = np.arange(21)
         train_score2, val_score2 = validation_curve(
             PolynomialRegression(), X2, y2,
             param_name='polynomialfeatures__degree',
             param_range=degree, cv=7)

         plt.plot(degree, np.median(train_score2, 1),
                  color='blue', label='training score')
         plt.plot(degree, np.median(val_score2, 1),
                  color='red', label='validation score')
         plt.plot(degree, np.median(train_score, 1),
                  color='blue', alpha=0.3, linestyle='dashed')
         plt.plot(degree, np.median(val_score, 1),
                  color='red', alpha=0.3, linestyle='dashed')
         plt.legend(loc='lower center')
         plt.ylim(0, 1)
         plt.xlabel('degree')
         plt.ylabel('score');
```

The solid lines show the new results, while the fainter dashed lines show the results on the previous smaller dataset. It is clear from the validation curve that the larger dataset can support a much more complicated model: the peak here is probably around a degree of 6, but even a degree-20 model isn't seriously overfitting the data—the validation and training scores remain very close.

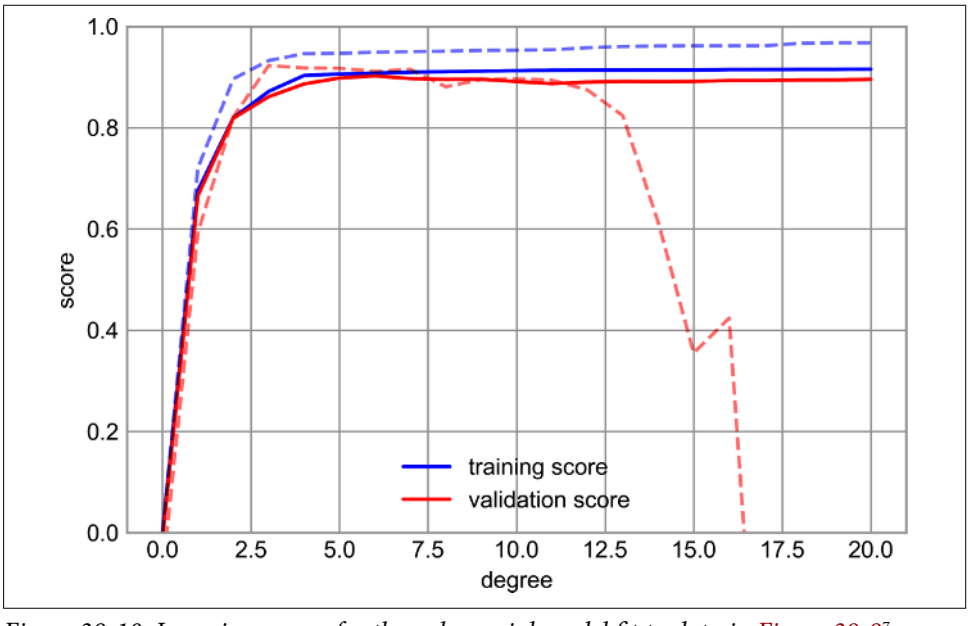

Figure 39-10. Learning curves for the polynomial model fit to data in Figure 39-9[7]

So, the behavior of the validation curve has not one but two important inputs: the model complexity and the number of training points. We can gain further insight by exploring the behavior of the model as a function of the number of training points, which we can do by using increasingly larger subsets of the data to fit our model. A plot of the training/validation score with respect to the size of the training set is sometimes known as a *learning curve*.

The general behavior we would expect from a learning curve is this:

- A model of a given complexity will *overfit* a small dataset: this means the training score will be relatively high, while the validation score will be relatively low.
- A model of a given complexity will *underfit* a large dataset: this means that the training score will decrease, but the validation score will increase.
- A model will never, except by chance, give a better score to the validation set than the training set: this means the curves should keep getting closer together but never cross.

With these features in mind, we would expect a learning curve to look qualitatively like that shown in Figure 39-11.

7 A full-color version of this figure can be found on GitHub.

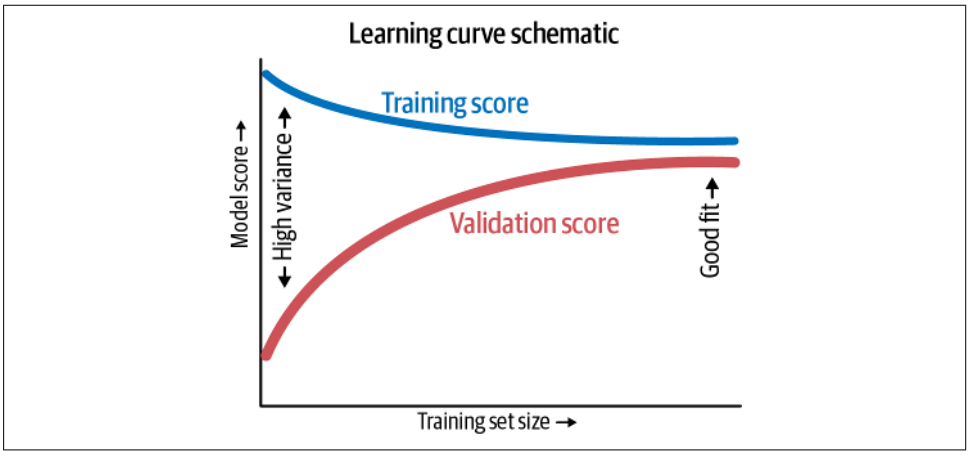

Figure 39-11. Schematic showing the typical interpretation of learning curves[8]

The notable feature of the learning curve is the convergence to a particular score as the number of training samples grows. In particular, once you have enough points that a particular model has converged, *adding more training data will not help you!* The only way to increase model performance in this case is to use another (often more complex) model.

Scikit-Learn offers a convenient utility for computing such learning curves from your models; here we will compute a learning curve for our original dataset with a second-order polynomial model and a ninth-order polynomial (see Figure 39-12).

```
In [17]: from sklearn.model_selection import learning_curve

         fig, ax = plt.subplots(1, 2, figsize=(16, 6))
         fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

         for i, degree in enumerate([2, 9]):
             N, train_lc, val_lc = learning_curve(
                 PolynomialRegression(degree), X, y, cv=7,
                 train_sizes=np.linspace(0.3, 1, 25))

             ax[i].plot(N, np.mean(train_lc, 1),
                        color='blue', label='training score')
             ax[i].plot(N, np.mean(val_lc, 1),
                        color='red', label='validation score')
             ax[i].hlines(np.mean([train_lc[-1], val_lc[-1]]), N[0],
                          N[-1], color='gray', linestyle='dashed')

             ax[i].set_ylim(0, 1)
```

8 Code to produce this figure can be found in the online appendix.

```
ax[i].set_xlim(N[0], N[-1])
ax[i].set_xlabel('training size')
ax[i].set_ylabel('score')
ax[i].set_title('degree = {0}'.format(degree), size=14)
ax[i].legend(loc='best')
```

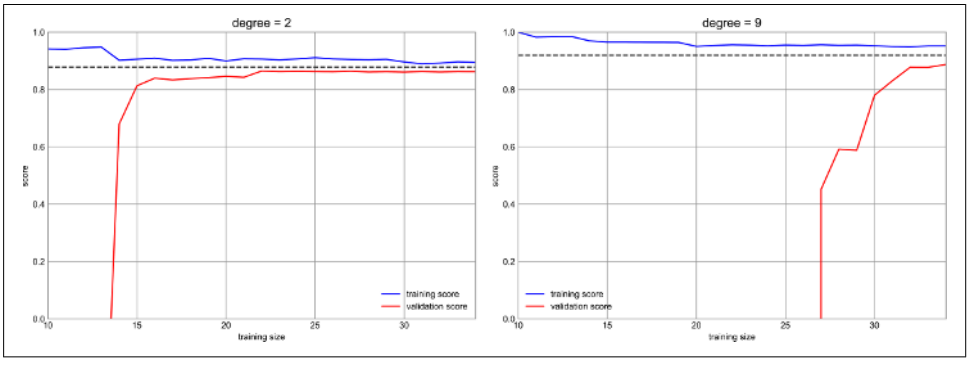

Figure 39-12. Learning curves for a low-complexity model (left) and a high-complexity model (right)[9]

This is a valuable diagnostic, because it gives us a visual depiction of how our model responds to increasing amounts of training data. In particular, when the learning curve has already converged (i.e., when the training and validation curves are already close to each other) *adding more training data will not significantly improve the fit!* This situation is seen in the left panel, with the learning curve for the degree-2 model.

The only way to increase the converged score is to use a different (usually more complicated) model. We see this in the right panel: by moving to a much more complicated model, we increase the score of convergence (indicated by the dashed line), but at the expense of higher model variance (indicated by the difference between the training and validation scores). If we were to add even more data points, the learning curve for the more complicated model would eventually converge.

Plotting a learning curve for your particular choice of model and dataset can help you to make this type of decision about how to move forward in improving your analysis.

The solid lines show the new results, while the fainter dashed lines show the results on the previous smaller dataset. It is clear from the validation curve that the larger dataset can support a much more complicated model: the peak here is probably around a degree of 6, but even a degree-20 model isn't seriously overfitting the data—the validation and training scores remain very close.

9 A full-size version of this figure can be found on GitHub.

Validation in Practice: Grid Search

The preceding discussion is meant to give you some intuition into the trade-off between bias and variance, and its dependence on model complexity and training set size. In practice, models generally have more than one knob to turn, meaning plots of validation and learning curves change from lines to multidimensional surfaces. In these cases, such visualizations are difficult, and we would rather simply find the particular model that maximizes the validation score.

Scikit-Learn provides some tools to make this kind of search more convenient: here we'll consider the use of grid search to find the optimal polynomial model. We will explore a two-dimensional grid of model features, namely the polynomial degree and the flag telling us whether to fit the intercept. This can be set up using Scikit-Learn's `GridSearchCV` meta-estimator:

```
In [18]: from sklearn.model_selection import GridSearchCV

         param_grid = {'polynomialfeatures__degree': np.arange(21),
                       'linearregression__fit_intercept': [True, False]}

         grid = GridSearchCV(PolynomialRegression(), param_grid, cv=7)
```

Notice that like a normal estimator, this has not yet been applied to any data. Calling the `fit` method will fit the model at each grid point, keeping track of the scores along the way:

```
In [19]: grid.fit(X, y);
```

Now that the model is fit, we can ask for the best parameters as follows:

```
In [20]: grid.best_params_
Out[20]: {'linearregression__fit_intercept': False, 'polynomialfeatures__degree': 4}
```

Finally, if we wish, we can use the best model and show the fit to our data using code from before (see Figure 39-13).

```
In [21]: model = grid.best_estimator_

         plt.scatter(X.ravel(), y)
         lim = plt.axis()
         y_test = model.fit(X, y).predict(X_test)
         plt.plot(X_test.ravel(), y_test);
         plt.axis(lim);
```

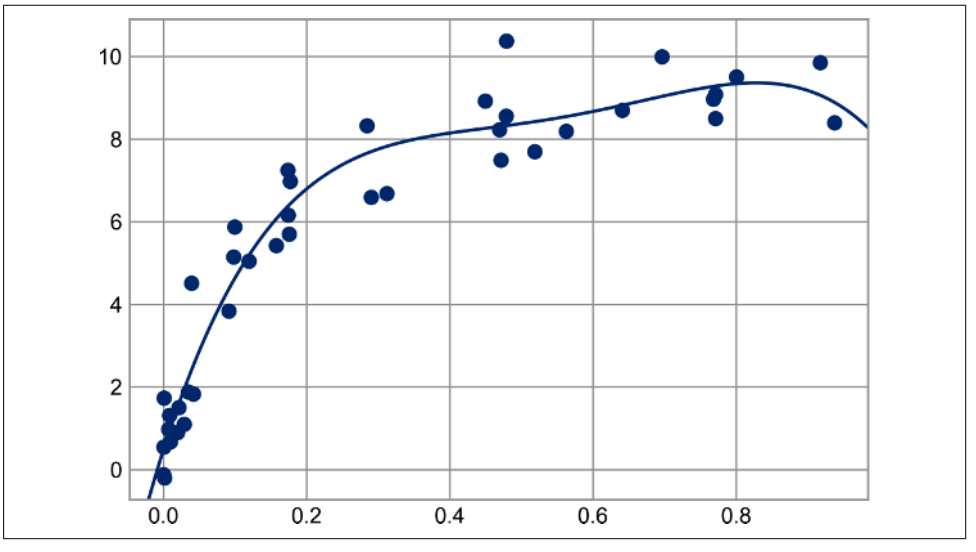

Figure 39-13. The best-fit model determined via an automatic grid search

Other options in `GridSearchCV` include the ability to specify a custom scoring function, to parallelize the computations, to do randomized searches, and more. For more information, see the examples in Chapters 49 and 50, or refer to Scikit-Learn's grid search documentation.

Summary

In this chapter we began to explore the concept of model validation and hyperparameter optimization, focusing on intuitive aspects of the bias–variance trade-off and how it comes into play when fitting models to data. In particular, we found that the use of a validation set or cross-validation approach is vital when tuning parameters in order to avoid overfitting for more complex/flexible models.

In later chapters, we will discuss the details of particularly useful models, what tuning is available for these models, and how these free parameters affect model complexity. Keep the lessons of this chapter in mind as you read on and learn about these machine learning approaches!

CHAPTER 12
k-Nearest Neighbors

If you want to annoy your neighbors, tell the truth about them.
—Pietro Aretino

Imagine that you're trying to predict how I'm going to vote in the next presidential election. If you know nothing else about me (and if you have the data), one sensible approach is to look at how my *neighbors* are planning to vote. Living in Seattle, as I do, my neighbors are invariably planning to vote for the Democratic candidate, which suggests that "Democratic candidate" is a good guess for me as well.

Now imagine you know more about me than just geography—perhaps you know my age, my income, how many kids I have, and so on. To the extent my behavior is influenced (or characterized) by those things, looking just at my neighbors who are close to me among all those dimensions seems likely to be an even better predictor than looking at all my neighbors. This is the idea behind *nearest neighbors classification*.

The Model

Nearest neighbors is one of the simplest predictive models there is. It makes no mathematical assumptions, and it doesn't require any sort of heavy machinery. The only things it requires are:

- Some notion of distance
- An assumption that points that are close to one another are similar

Most of the techniques we'll see in this book look at the dataset as a whole in order to learn patterns in the data. Nearest neighbors, on the other hand, quite consciously neglects a lot of information, since the prediction for each new point depends only on the handful of points closest to it.

What's more, nearest neighbors is probably not going to help you understand the drivers of whatever phenomenon you're looking at. Predicting my votes based on my neighbors' votes doesn't tell you much about what causes me to vote the way I do, whereas some alternative model that predicted my vote based on (say) my income and marital status very well might.

In the general situation, we have some data points and we have a corresponding set of labels. The labels could be `True` and `False`, indicating whether each input satisfies some condition like "is spam?" or "is poisonous?" or "would be enjoyable to watch?" Or they could be categories, like movie ratings (G, PG, PG-13, R, NC-17). Or they could be the names of presidential candidates. Or they could be favorite programming languages.

In our case, the data points will be vectors, which means that we can use the `distance` function from Chapter 4.

Let's say we've picked a number k like 3 or 5. Then, when we want to classify some new data point, we find the k nearest labeled points and let them vote on the new output.

To do this, we'll need a function that counts votes. One possibility is:

```
from typing import List
from collections import Counter

def raw_majority_vote(labels: List[str]) -> str:
    votes = Counter(labels)
    winner, _ = votes.most_common(1)[0]
    return winner

assert raw_majority_vote(['a', 'b', 'c', 'b']) == 'b'
```

But this doesn't do anything intelligent with ties. For example, imagine we're rating movies and the five nearest movies are rated G, G, PG, PG, and R. Then G has two votes and PG also has two votes. In that case, we have several options:

- Pick one of the winners at random.
- Weight the votes by distance and pick the weighted winner.
- Reduce k until we find a unique winner.

We'll implement the third:

```
def majority_vote(labels: List[str]) -> str:
    """Assumes that labels are ordered from nearest to farthest."""
    vote_counts = Counter(labels)
    winner, winner_count = vote_counts.most_common(1)[0]
    num_winners = len([count
                       for count in vote_counts.values()
                       if count == winner_count])
```

```
        if num_winners == 1:
            return winner                     # unique winner, so return it
        else:
            return majority_vote(labels[:-1]) # try again without the farthest

    # Tie, so look at first 4, then 'b'
    assert majority_vote(['a', 'b', 'c', 'b', 'a']) == 'b'
```

This approach is sure to work eventually, since in the worst case we go all the way down to just one label, at which point that one label wins.

With this function it's easy to create a classifier:

```
from typing import NamedTuple
from scratch.linear_algebra import Vector, distance

class LabeledPoint(NamedTuple):
    point: Vector
    label: str

def knn_classify(k: int,
                 labeled_points: List[LabeledPoint],
                 new_point: Vector) -> str:

    # Order the labeled points from nearest to farthest.
    by_distance = sorted(labeled_points,
                         key=lambda lp: distance(lp.point, new_point))

    # Find the labels for the k closest
    k_nearest_labels = [lp.label for lp in by_distance[:k]]

    # and let them vote.
    return majority_vote(k_nearest_labels)
```

Let's take a look at how this works.

Example: The Iris Dataset

The *Iris* dataset is a staple of machine learning. It contains a bunch of measurements for 150 flowers representing three species of iris. For each flower we have its petal length, petal width, sepal length, and sepal width, as well as its species. You can download it from *https://archive.ics.uci.edu/ml/datasets/iris*:

```
import requests

data = requests.get(
  "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
)

with open('iris.dat', 'w') as f:
    f.write(data.text)
```

The data is comma-separated, with fields:

```
sepal_length, sepal_width, petal_length, petal_width, class
```

For example, the first row looks like:

```
5.1,3.5,1.4,0.2,Iris-setosa
```

In this section we'll try to build a model that can predict the class (that is, the species) from the first four measurements.

To start with, let's load and explore the data. Our nearest neighbors function expects a `LabeledPoint`, so let's represent our data that way:

```
from typing import Dict
import csv
from collections import defaultdict

def parse_iris_row(row: List[str]) -> LabeledPoint:
    """
    sepal_length, sepal_width, petal_length, petal_width, class
    """
    measurements = [float(value) for value in row[:-1]]
    # class is e.g. "Iris-virginica"; we just want "virginica"
    label = row[-1].split("-")[-1]

    return LabeledPoint(measurements, label)

with open('iris.data') as f:
    reader = csv.reader(f)
    iris_data = [parse_iris_row(row) for row in reader]

# We'll also group just the points by species/label so we can plot them
points_by_species: Dict[str, List[Vector]] = defaultdict(list)
for iris in iris_data:
    points_by_species[iris.label].append(iris.point)
```

We'd like to plot the measurements so we can see how they vary by species. Unfortunately, they are four-dimensional, which makes them tricky to plot. One thing we can do is look at the scatterplots for each of the six pairs of measurements (Figure 12-1). I won't explain all the details, but it's a nice illustration of more complicated things you can do with matplotlib, so it's worth studying:

```
from matplotlib import pyplot as plt
metrics = ['sepal length', 'sepal width', 'petal length', 'petal width']
pairs = [(i, j) for i in range(4) for j in range(4) if i < j]
marks = ['+', '.', 'x']  # we have 3 classes, so 3 markers

fig, ax = plt.subplots(2, 3)

for row in range(2):
    for col in range(3):
        i, j = pairs[3 * row + col]
```

```
            ax[row][col].set_title(f"{metrics[i]} vs {metrics[j]}", fontsize=8)
            ax[row][col].set_xticks([])
            ax[row][col].set_yticks([])

            for mark, (species, points) in zip(marks, points_by_species.items()):
                xs = [point[i] for point in points]
                ys = [point[j] for point in points]
                ax[row][col].scatter(xs, ys, marker=mark, label=species)

ax[-1][-1].legend(loc='lower right', prop={'size': 6})
plt.show()
```

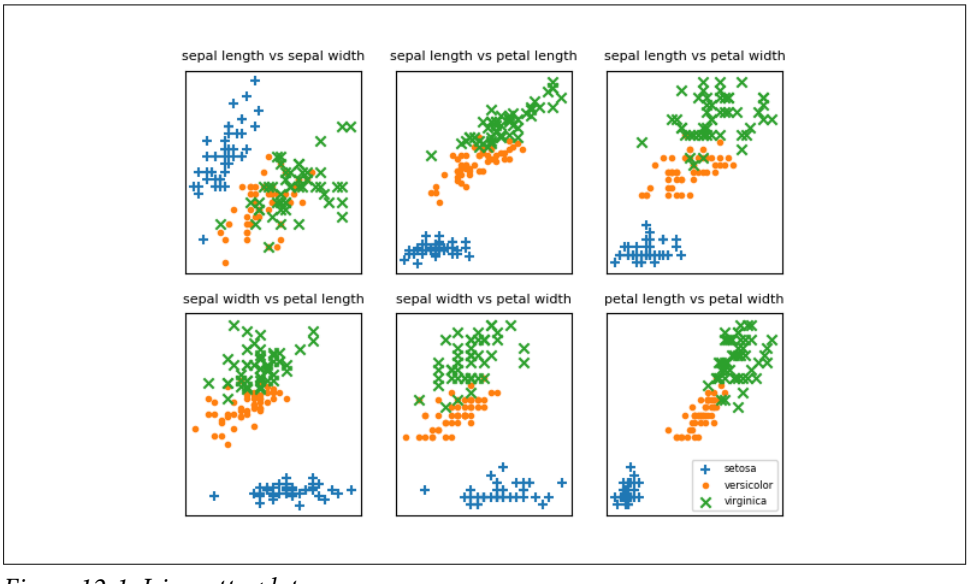

Figure 12-1. Iris scatterplots

If you look at those plots, it seems like the measurements really do cluster by species. For example, looking at sepal length and sepal width alone, you probably couldn't distinguish between *versicolor* and *virginica*. But once you add petal length and width into the mix, it seems like you should be able to predict the species based on the nearest neighbors.

To start with, let's split the data into a test set and a training set:

```
import random
from scratch.machine_learning import split_data

random.seed(12)
iris_train, iris_test = split_data(iris_data, 0.70)
assert len(iris_train) == 0.7 * 150
assert len(iris_test) == 0.3 * 150
```

The training set will be the "neighbors" that we'll use to classify the points in the test set. We just need to choose a value for k, the number of neighbors who get to vote. Too small (think $k = 1$), and we let outliers have too much influence; too large (think $k = 105$), and we just predict the most common class in the dataset.

In a real application (and with more data), we might create a separate validation set and use it to choose k. Here we'll just use $k = 5$:

```
from typing import Tuple

# track how many times we see (predicted, actual)
confusion_matrix: Dict[Tuple[str, str], int] = defaultdict(int)
num_correct = 0

for iris in iris_test:
    predicted = knn_classify(5, iris_train, iris.point)
    actual = iris.label

    if predicted == actual:
        num_correct += 1

    confusion_matrix[(predicted, actual)] += 1

pct_correct = num_correct / len(iris_test)
print(pct_correct, confusion_matrix)
```

On this simple dataset, the model predicts almost perfectly. There's one *versicolor* for which it predicts *virginica*, but otherwise it gets things exactly right.

The Curse of Dimensionality

The k-nearest neighbors algorithm runs into trouble in higher dimensions thanks to the "curse of dimensionality," which boils down to the fact that high-dimensional spaces are *vast*. Points in high-dimensional spaces tend not to be close to one another at all. One way to see this is by randomly generating pairs of points in the d-dimensional "unit cube" in a variety of dimensions, and calculating the distances between them.

Generating random points should be second nature by now:

```
def random_point(dim: int) -> Vector:
    return [random.random() for _ in range(dim)]
```

as is writing a function to generate the distances:

```
def random_distances(dim: int, num_pairs: int) -> List[float]:
    return [distance(random_point(dim), random_point(dim))
            for _ in range(num_pairs)]
```

For every dimension from 1 to 100, we'll compute 10,000 distances and use those to compute the average distance between points and the minimum distance between points in each dimension (Figure 12-2):

```
import tqdm
dimensions = range(1, 101)

avg_distances = []
min_distances = []

random.seed(0)
for dim in tqdm.tqdm(dimensions, desc="Curse of Dimensionality"):
    distances = random_distances(dim, 10000)      # 10,000 random pairs
    avg_distances.append(sum(distances) / 10000)  # track the average
    min_distances.append(min(distances))          # track the minimum
```

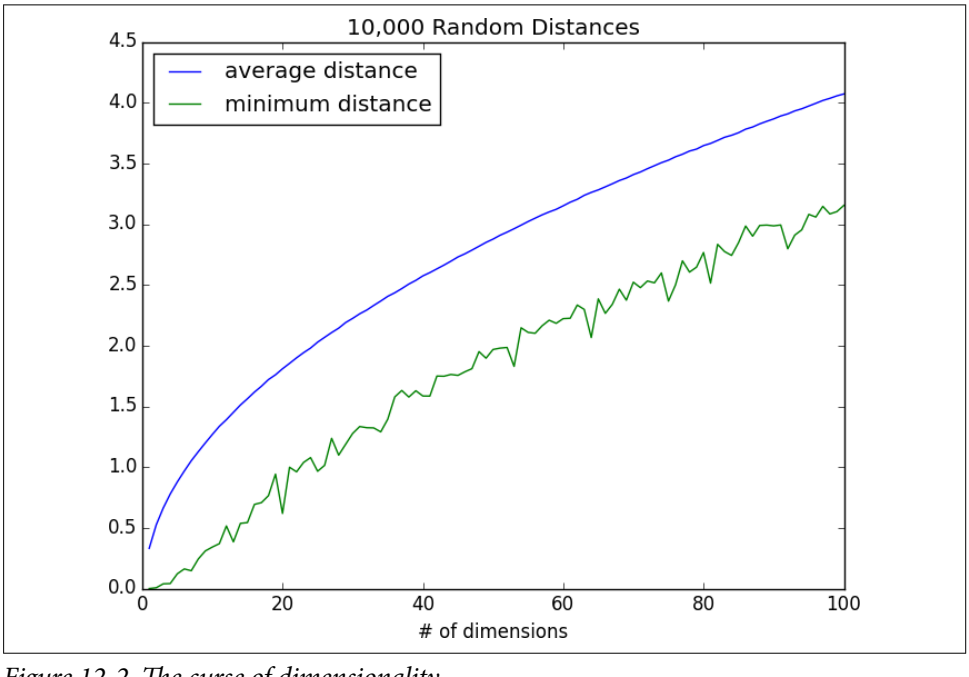

Figure 12-2. The curse of dimensionality

As the number of dimensions increases, the average distance between points increases. But what's more problematic is the ratio between the closest distance and the average distance (Figure 12-3):

```
min_avg_ratio = [min_dist / avg_dist
                 for min_dist, avg_dist in zip(min_distances, avg_distances)]
```

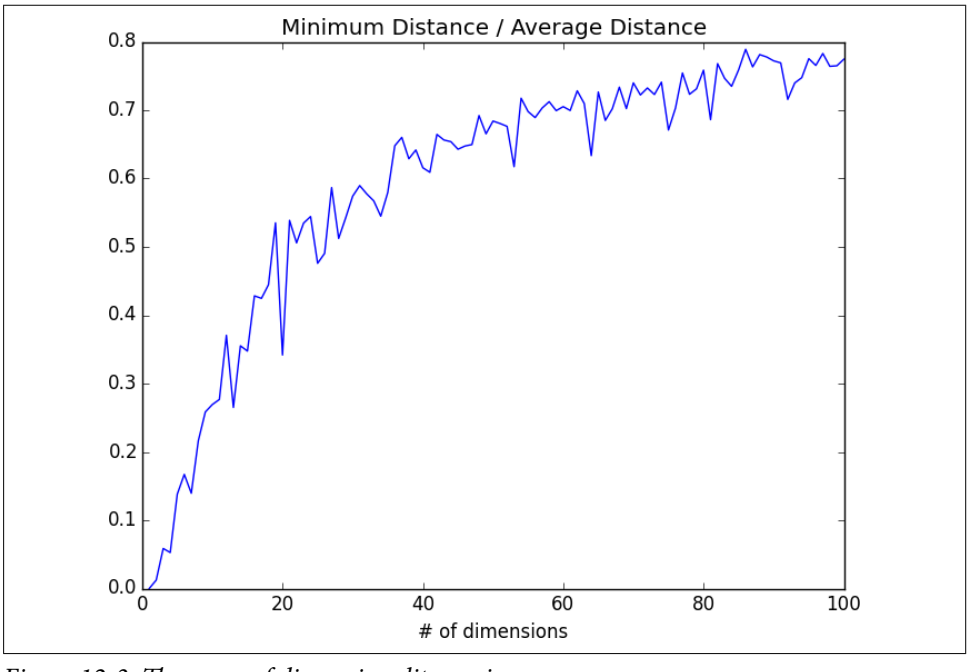

Figure 12-3. The curse of dimensionality again

In low-dimensional datasets, the closest points tend to be much closer than average. But two points are close only if they're close in every dimension, and every extra dimension—even if just noise—is another opportunity for each point to be farther away from every other point. When you have a lot of dimensions, it's likely that the closest points aren't much closer than average, so two points being close doesn't mean very much (unless there's a lot of structure in your data that makes it behave as if it were much lower-dimensional).

A different way of thinking about the problem involves the sparsity of higher-dimensional spaces.

If you pick 50 random numbers between 0 and 1, you'll probably get a pretty good sample of the unit interval (Figure 12-4).

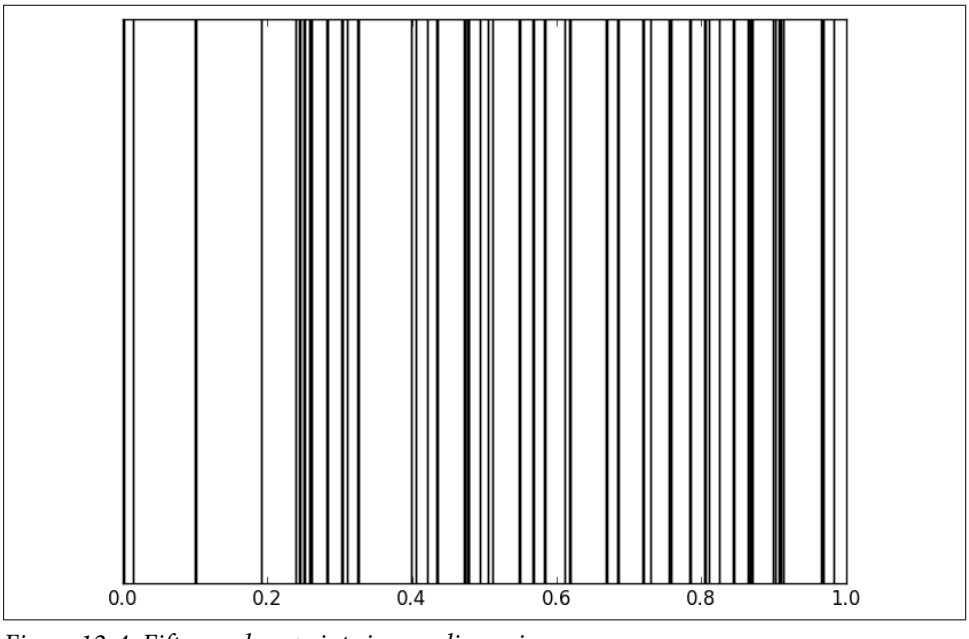

Figure 12-4. Fifty random points in one dimension

If you pick 50 random points in the unit square, you'll get less coverage (Figure 12-5).

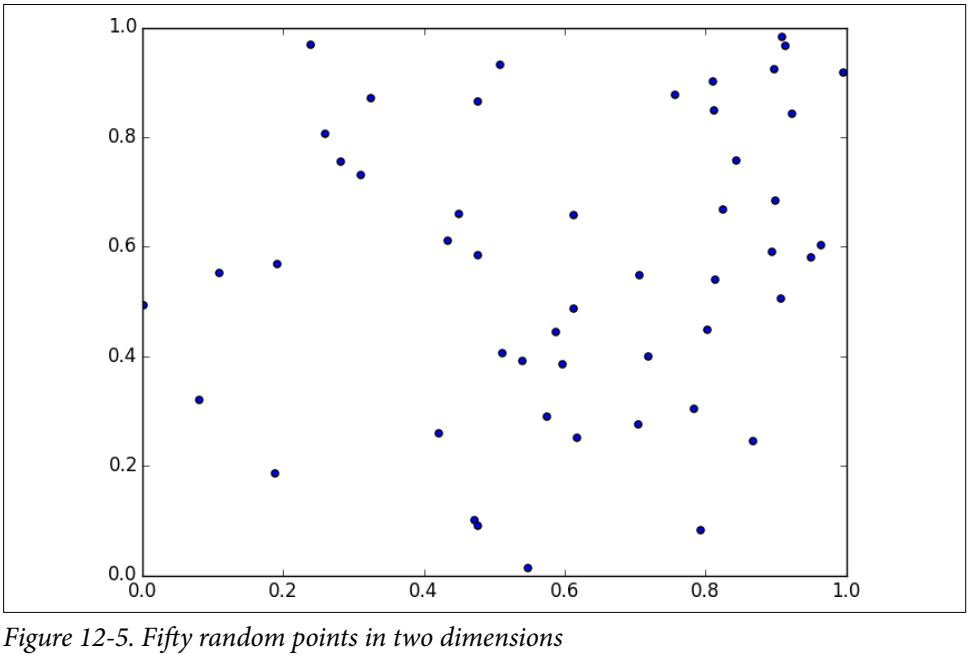

Figure 12-5. Fifty random points in two dimensions

And in three dimensions, less still (Figure 12-6).

matplotlib doesn't graph four dimensions well, so that's as far as we'll go, but you can see already that there are starting to be large empty spaces with no points near them. In more dimensions—unless you get exponentially more data—those large empty spaces represent regions far from all the points you want to use in your predictions.

So if you're trying to use nearest neighbors in higher dimensions, it's probably a good idea to do some kind of dimensionality reduction first.

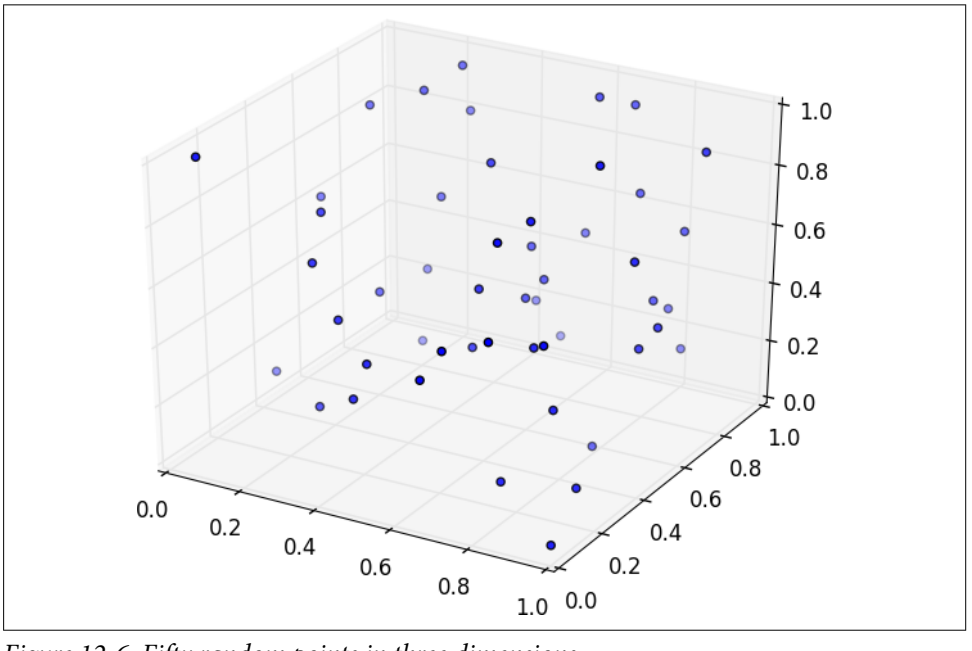

Figure 12-6. Fifty random points in three dimensions

For Further Exploration

scikit-learn has many nearest neighbor models.

CHAPTER 13
Naive Bayes

It is well for the heart to be naive and for the mind not to be.
—Anatole France

A social network isn't much good if people can't network. Accordingly, DataSciencester has a popular feature that allows members to send messages to other members. And while most members are responsible citizens who send only well-received "how's it going?" messages, a few miscreants persistently spam other members about get-rich schemes, no-prescription-required pharmaceuticals, and for-profit data science credentialing programs. Your users have begun to complain, and so the VP of Messaging has asked you to use data science to figure out a way to filter out these spam messages.

A Really Dumb Spam Filter

Imagine a "universe" that consists of receiving a message chosen randomly from all possible messages. Let S be the event "the message is spam" and B be the event "the message contains the word *bitcoin*." Bayes's theorem tells us that the probability that the message is spam conditional on containing the word *bitcoin* is:

$$P(S \mid B) = [P(B \mid S)P(S)] / [P(B \mid S)P(S) + P(B \mid \neg S)P(\neg S)]$$

The numerator is the probability that a message is spam *and* contains *bitcoin*, while the denominator is just the probability that a message contains *bitcoin*. Hence, you can think of this calculation as simply representing the proportion of *bitcoin* messages that are spam.

If we have a large collection of messages we know are spam, and a large collection of messages we know are not spam, then we can easily estimate $P(B|S)$ and $P(B|\neg S)$. If

we further assume that any message is equally likely to be spam or not spam (so that $P(S) = P(\neg S) = 0.5$), then:

$$P(S \mid B) = P(B \mid S)/[P(B \mid S) + P(B \mid \neg S)]$$

For example, if 50% of spam messages have the word *bitcoin*, but only 1% of nonspam messages do, then the probability that any given *bitcoin*-containing email is spam is:

$$0.5/(0.5 + 0.01) = 98\,\%$$

A More Sophisticated Spam Filter

Imagine now that we have a vocabulary of many words, w_1 ..., w_n. To move this into the realm of probability theory, we'll write X_i for the event "a message contains the word w_i." Also imagine that (through some unspecified-at-this-point process) we've come up with an estimate $P(X_i|S)$ for the probability that a spam message contains the *i*th word, and a similar estimate $P(X_i|\neg S)$ for the probability that a nonspam message contains the *i*th word.

The key to Naive Bayes is making the (big) assumption that the presences (or absences) of each word are independent of one another, conditional on a message being spam or not. Intuitively, this assumption means that knowing whether a certain spam message contains the word *bitcoin* gives you no information about whether that same message contains the word *rolex*. In math terms, this means that:

$$P(X_1 = x_1, \ldots, X_n = x_n \mid S) = P(X_1 = x_1 \mid S) \times \cdots \times P(X_n = x_n \mid S)$$

This is an extreme assumption. (There's a reason the technique has *naive* in its name.) Imagine that our vocabulary consists *only* of the words *bitcoin* and *rolex*, and that half of all spam messages are for "earn bitcoin" and that the other half are for "authentic rolex." In this case, the Naive Bayes estimate that a spam message contains both *bitcoin* and *rolex* is:

$$P(X_1 = 1, X_2 = 1 \mid S) = P(X_1 = 1 \mid S)P(X_2 = 1 \mid S) = .5 \times .5 = .25$$

since we've assumed away the knowledge that *bitcoin* and *rolex* actually never occur together. Despite the unrealisticness of this assumption, this model often performs well and has historically been used in actual spam filters.

The same Bayes's theorem reasoning we used for our "bitcoin-only" spam filter tells us that we can calculate the probability a message is spam using the equation:

$$P(S \mid X = x) = P(X = x \mid S) / [P(X = x \mid S) + P(X = x \mid \neg S)]$$

The Naive Bayes assumption allows us to compute each of the probabilities on the right simply by multiplying together the individual probability estimates for each vocabulary word.

In practice, you usually want to avoid multiplying lots of probabilities together, to prevent a problem called *underflow*, in which computers don't deal well with floating-point numbers that are too close to 0. Recalling from algebra that $\log(ab) = \log a + \log b$ and that $\exp(\log x) = x$, we usually compute $p_1 * \cdots * p_n$ as the equivalent (but floating-point-friendlier):

$$\exp(\log(p_1) + \cdots + \log(p_n))$$

The only challenge left is coming up with estimates for $P(X_i \mid S)$ and $P(X_i \mid \neg S)$, the probabilities that a spam message (or nonspam message) contains the word w_i. If we have a fair number of "training" messages labeled as spam and not spam, an obvious first try is to estimate $P(X_i \mid S)$ simply as the fraction of spam messages containing the word w_i.

This causes a big problem, though. Imagine that in our training set the vocabulary word *data* only occurs in nonspam messages. Then we'd estimate $P(\text{data} \mid S) = 0$. The result is that our Naive Bayes classifier would always assign spam probability 0 to *any* message containing the word *data*, even a message like "data on free bitcoin and authentic rolex watches." To avoid this problem, we usually use some kind of smoothing.

In particular, we'll choose a *pseudocount*—*k*—and estimate the probability of seeing the *i*th word in a spam message as:

$$P(X_i \mid S) = (k + \text{number of spams containing } w_i)/(2k + \text{number of spams})$$

We do similarly for $P(X_i \mid \neg S)$. That is, when computing the spam probabilities for the *i*th word, we assume we also saw *k* additional nonspams containing the word and *k* additional nonspams not containing the word.

For example, if *data* occurs in 0/98 spam messages, and if *k* is 1, we estimate $P(\text{data} \mid S)$ as 1/100 = 0.01, which allows our classifier to still assign some nonzero spam probability to messages that contain the word *data*.

Implementation

Now we have all the pieces we need to build our classifier. First, let's create a simple function to tokenize messages into distinct words. We'll first convert each message to lowercase, then use `re.findall` to extract "words" consisting of letters, numbers, and apostrophes. Finally, we'll use `set` to get just the distinct words:

```
from typing import Set
import re

def tokenize(text: str) -> Set[str]:
    text = text.lower()                         # Convert to lowercase,
    all_words = re.findall("[a-z0-9']+", text)  # extract the words, and
    return set(all_words)                       # remove duplicates.

assert tokenize("Data Science is science") == {"data", "science", "is"}
```

We'll also define a type for our training data:

```
from typing import NamedTuple

class Message(NamedTuple):
    text: str
    is_spam: bool
```

As our classifier needs to keep track of tokens, counts, and labels from the training data, we'll make it a class. Following convention, we refer to nonspam emails as *ham* emails.

The constructor will take just one parameter, the pseudocount to use when computing probabilities. It also initializes an empty set of tokens, counters to track how often each token is seen in spam messages and ham messages, and counts of how many spam and ham messages it was trained on:

```
from typing import List, Tuple, Dict, Iterable
import math
from collections import defaultdict

class NaiveBayesClassifier:
    def __init__(self, k: float = 0.5) -> None:
        self.k = k  # smoothing factor

        self.tokens: Set[str] = set()
        self.token_spam_counts: Dict[str, int] = defaultdict(int)
        self.token_ham_counts: Dict[str, int] = defaultdict(int)
        self.spam_messages = self.ham_messages = 0
```

Next, we'll give it a method to train it on a bunch of messages. First, we increment the `spam_messages` and `ham_messages` counts. Then we tokenize each message text, and for each token we increment the `token_spam_counts` or `token_ham_counts` based on the message type:

```
    def train(self, messages: Iterable[Message]) -> None:
        for message in messages:
            # Increment message counts
            if message.is_spam:
                self.spam_messages += 1
            else:
                self.ham_messages += 1

            # Increment word counts
            for token in tokenize(message.text):
                self.tokens.add(token)
                if message.is_spam:
                    self.token_spam_counts[token] += 1
                else:
                    self.token_ham_counts[token] += 1
```

Ultimately we'll want to predict *P*(spam | token). As we saw earlier, to apply Bayes's theorem we need to know *P*(token | spam) and *P*(token | ham) for each token in the vocabulary. So we'll create a "private" helper function to compute those:

```
    def _probabilities(self, token: str) -> Tuple[float, float]:
        """returns P(token | spam) and P(token | ham)"""
        spam = self.token_spam_counts[token]
        ham = self.token_ham_counts[token]

        p_token_spam = (spam + self.k) / (self.spam_messages + 2 * self.k)
        p_token_ham = (ham + self.k) / (self.ham_messages + 2 * self.k)

        return p_token_spam, p_token_ham
```

Finally, we're ready to write our `predict` method. As mentioned earlier, rather than multiplying together lots of small probabilities, we'll instead sum up the log probabilities:

```
    def predict(self, text: str) -> float:
        text_tokens = tokenize(text)
        log_prob_if_spam = log_prob_if_ham = 0.0

        # Iterate through each word in our vocabulary
        for token in self.tokens:
            prob_if_spam, prob_if_ham = self._probabilities(token)

            # If *token* appears in the message,
            # add the log probability of seeing it
            if token in text_tokens:
                log_prob_if_spam += math.log(prob_if_spam)
                log_prob_if_ham += math.log(prob_if_ham)

            # Otherwise add the log probability of _not_ seeing it,
            # which is log(1 - probability of seeing it)
            else:
                log_prob_if_spam += math.log(1.0 - prob_if_spam)
                log_prob_if_ham += math.log(1.0 - prob_if_ham)
```

```
        prob_if_spam = math.exp(log_prob_if_spam)
        prob_if_ham = math.exp(log_prob_if_ham)
        return prob_if_spam / (prob_if_spam + prob_if_ham)
```

And now we have a classifier.

Testing Our Model

Let's make sure our model works by writing some unit tests for it.

```
messages = [Message("spam rules", is_spam=True),
            Message("ham rules", is_spam=False),
            Message("hello ham", is_spam=False)]

model = NaiveBayesClassifier(k=0.5)
model.train(messages)
```

First, let's check that it got the counts right:

```
assert model.tokens == {"spam", "ham", "rules", "hello"}
assert model.spam_messages == 1
assert model.ham_messages == 2
assert model.token_spam_counts == {"spam": 1, "rules": 1}
assert model.token_ham_counts == {"ham": 2, "rules": 1, "hello": 1}
```

Now let's make a prediction. We'll also (laboriously) go through our Naive Bayes logic by hand, and make sure that we get the same result:

```
text = "hello spam"

probs_if_spam = [
    (1 + 0.5) / (1 + 2 * 0.5),      # "spam"  (present)
    1 - (0 + 0.5) / (1 + 2 * 0.5),  # "ham"   (not present)
    1 - (1 + 0.5) / (1 + 2 * 0.5),  # "rules" (not present)
    (0 + 0.5) / (1 + 2 * 0.5)       # "hello" (present)
]

probs_if_ham = [
    (0 + 0.5) / (2 + 2 * 0.5),      # "spam"  (present)
    1 - (2 + 0.5) / (2 + 2 * 0.5),  # "ham"   (not present)
    1 - (1 + 0.5) / (2 + 2 * 0.5),  # "rules" (not present)
    (1 + 0.5) / (2 + 2 * 0.5),      # "hello" (present)
]

p_if_spam = math.exp(sum(math.log(p) for p in probs_if_spam))
p_if_ham = math.exp(sum(math.log(p) for p in probs_if_ham))

# Should be about 0.83
assert model.predict(text) == p_if_spam / (p_if_spam + p_if_ham)
```

This test passes, so it seems like our model is doing what we think it is. If you look at the actual probabilities, the two big drivers are that our message contains *spam*

(which our lone training spam message did) and that it doesn't contain *ham* (which both our training ham messages did).

Now let's try it on some real data.

Using Our Model

A popular (if somewhat old) dataset is the SpamAssassin public corpus. We'll look at the files prefixed with *20021010*.

Here is a script that will download and unpack them to the directory of your choice (or you can do it manually):

```
from io import BytesIO  # So we can treat bytes as a file.
import requests         # To download the files, which
import tarfile          # are in .tar.bz format.

BASE_URL = "https://spamassassin.apache.org/old/publiccorpus"
FILES = ["20021010_easy_ham.tar.bz2",
         "20021010_hard_ham.tar.bz2",
         "20021010_spam.tar.bz2"]

# This is where the data will end up,
# in /spam, /easy_ham, and /hard_ham subdirectories.
# Change this to where you want the data.
OUTPUT_DIR = 'spam_data'

for filename in FILES:
    # Use requests to get the file contents at each URL.
    content = requests.get(f"{BASE_URL}/{filename}").content

    # Wrap the in-memory bytes so we can use them as a "file."
    fin = BytesIO(content)

    # And extract all the files to the specified output dir.
    with tarfile.open(fileobj=fin, mode='r:bz2') as tf:
        tf.extractall(OUTPUT_DIR)
```

It's possible the location of the files will change (this happened between the first and second editions of this book), in which case adjust the script accordingly.

After downloading the data you should have three folders: *spam*, *easy_ham*, and *hard_ham*. Each folder contains many emails, each contained in a single file. To keep things *really* simple, we'll just look at the subject lines of each email.

How do we identify the subject line? When we look through the files, they all seem to start with "Subject:". So we'll look for that:

```
import glob, re

# modify the path to wherever you've put the files
```

```
path = 'spam_data/*/*'

data: List[Message] = []

# glob.glob returns every filename that matches the wildcarded path
for filename in glob.glob(path):
    is_spam = "ham" not in filename

    # There are some garbage characters in the emails; the errors='ignore'
    # skips them instead of raising an exception.
    with open(filename, errors='ignore') as email_file:
        for line in email_file:
            if line.startswith("Subject:"):
                subject = line.lstrip("Subject: ")
                data.append(Message(subject, is_spam))
                break  # done with this file
```

Now we can split the data into training data and test data, and then we're ready to build a classifier:

```
import random
from scratch.machine_learning import split_data

random.seed(0)      # just so you get the same answers as me
train_messages, test_messages = split_data(data, 0.75)

model = NaiveBayesClassifier()
model.train(train_messages)
```

Let's generate some predictions and check how our model does:

```
from collections import Counter

predictions = [(message, model.predict(message.text))
               for message in test_messages]

# Assume that spam_probability > 0.5 corresponds to spam prediction
# and count the combinations of (actual is_spam, predicted is_spam)
confusion_matrix = Counter((message.is_spam, spam_probability > 0.5)
                           for message, spam_probability in predictions)

print(confusion_matrix)
```

This gives 84 true positives (spam classified as "spam"), 25 false positives (ham classified as "spam"), 703 true negatives (ham classified as "ham"), and 44 false negatives (spam classified as "ham"). This means our precision is 84 / (84 + 25) = 77%, and our recall is 84 / (84 + 44) = 65%, which are not bad numbers for such a simple model. (Presumably we'd do better if we looked at more than the subject lines.)

We can also inspect the model's innards to see which words are least and most indicative of spam:

```
def p_spam_given_token(token: str, model: NaiveBayesClassifier) -> float:
    # We probably shouldn't call private methods, but it's for a good cause.
    prob_if_spam, prob_if_ham = model._probabilities(token)

    return prob_if_spam / (prob_if_spam + prob_if_ham)

words = sorted(model.tokens, key=lambda t: p_spam_given_token(t, model))

print("spammiest_words", words[-10:])
print("hammiest_words", words[:10])
```

The spammiest words include things like *sale*, *mortgage*, *money*, and *rates*, whereas the hammiest words include things like *spambayes*, *users*, *apt*, and *perl*. So that also gives us some intuitive confidence that our model is basically doing the right thing.

How could we get better performance? One obvious way would be to get more data to train on. There are a number of ways to improve the model as well. Here are some possibilities that you might try:

- Look at the message content, not just the subject line. You'll have to be careful how you deal with the message headers.
- Our classifier takes into account every word that appears in the training set, even words that appear only once. Modify the classifier to accept an optional `min_count` threshold and ignore tokens that don't appear at least that many times.
- The tokenizer has no notion of similar words (e.g., *cheap* and *cheapest*). Modify the classifier to take an optional `stemmer` function that converts words to *equivalence classes* of words. For example, a really simple stemmer function might be:

  ```
  def drop_final_s(word):
      return re.sub("s$", "", word)
  ```

  Creating a good stemmer function is hard. People frequently use the Porter stemmer.
- Although our features are all of the form "message contains word w_i," there's no reason why this has to be the case. In our implementation, we could add extra features like "message contains a number" by creating phony tokens like *contains:number* and modifying the `tokenizer` to emit them when appropriate.

For Further Exploration

- Paul Graham's articles "A Plan for Spam" and "Better Bayesian Filtering" are interesting and give more insight into the ideas behind building spam filters.

- scikit-learn contains a `BernoulliNB` model that implements the same Naive Bayes algorithm we implemented here, as well as other variations on the model.

CHAPTER 41

In Depth: Naive Bayes Classification

The previous four chapters have given a general overview of the concepts of machine learning. In the rest of Part V, we will be taking a closer look first at four algorithms for supervised learning, and then at four algorithms for unsupervised learning. We start here with our first supervised method, naive Bayes classification.

Naive Bayes models are a group of extremely fast and simple classification algorithms that are often suitable for very high-dimensional datasets. Because they are so fast and have so few tunable parameters, they end up being useful as a quick-and-dirty baseline for a classification problem. This chapter will provide an intuitive explanation of how naive Bayes classifiers work, followed by a few examples of them in action on some datasets.

Bayesian Classification

Naive Bayes classifiers are built on Bayesian classification methods. These rely on Bayes's theorem, which is an equation describing the relationship of conditional probabilities of statistical quantities. In Bayesian classification, we're interested in finding the probability of a label L given some observed features, which we can write as $P(L \mid \text{features})$. Bayes's theorem tells us how to express this in terms of quantities we can compute more directly:

$$P(L \mid \text{features}) = \frac{P(\text{features} \mid L)P(L)}{P(\text{features})}$$

If we are trying to decide between two labels—let's call them L_1 and L_2—then one way to make this decision is to compute the ratio of the posterior probabilities for each label:

$$\frac{P(L_1 \mid \text{features})}{P(L_2 \mid \text{features})} = \frac{P(\text{features} \mid L_1)}{P(\text{features} \mid L_2)}\frac{P(L_1)}{P(L_2)}$$

All we need now is some model by which we can compute $P(\text{features} \mid L_i)$ for each label. Such a model is called a *generative model* because it specifies the hypothetical random process that generates the data. Specifying this generative model for each label is the main piece of the training of such a Bayesian classifier. The general version of such a training step is a very difficult task, but we can make it simpler through the use of some simplifying assumptions about the form of this model.

This is where the "naive" in "naive Bayes" comes in: if we make very naive assumptions about the generative model for each label, we can find a rough approximation of the generative model for each class, and then proceed with the Bayesian classification. Different types of naive Bayes classifiers rest on different naive assumptions about the data, and we will examine a few of these in the following sections.

We begin with the standard imports:

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import seaborn as sns
        plt.style.use('seaborn-whitegrid')
```

Gaussian Naive Bayes

Perhaps the easiest naive Bayes classifier to understand is Gaussian naive Bayes. With this classifier, the assumption is that *data from each label is drawn from a simple Gaussian distribution*. Imagine that we have the following data, shown in Figure 41-1:

```
In [2]: from sklearn.datasets import make_blobs
        X, y = make_blobs(100, 2, centers=2, random_state=2, cluster_std=1.5)
        plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu');
```

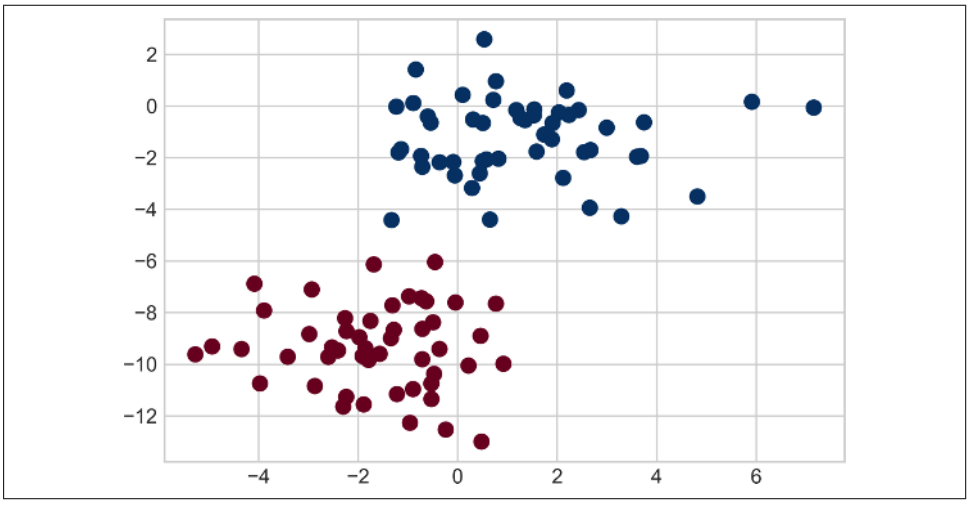

Figure 41-1. Data for Gaussian naive Bayes classification[1]

The simplest Gaussian model is to assume that the data is described by a Gaussian distribution with no covariance between dimensions. This model can be fit by computing the mean and standard deviation of the points within each label, which is all we need to define such a distribution. The result of this naive Gaussian assumption is shown in Figure 41-2.

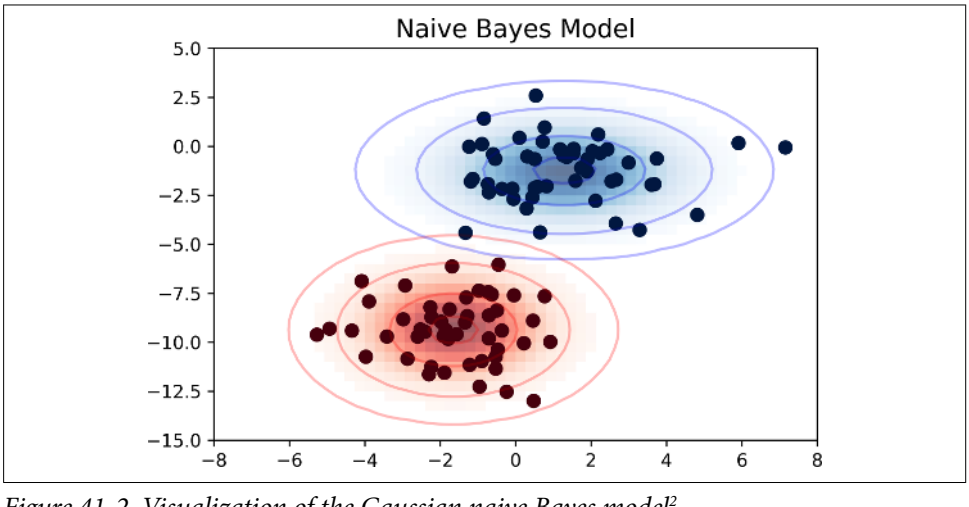

Figure 41-2. Visualization of the Gaussian naive Bayes model[2]

1 A full-color version of this figure can be found on GitHub.

2 Code to produce this figure can be found in the online appendix.

The ellipses here represent the Gaussian generative model for each label, with larger probability toward the center of the ellipses. With this generative model in place for each class, we have a simple recipe to compute the likelihood $P(\text{features} \mid L_1)$ for any data point, and thus we can quickly compute the posterior ratio and determine which label is the most probable for a given point.

This procedure is implemented in Scikit-Learn's `sklearn.naive_bayes.GaussianNB` estimator:

```
In [3]: from sklearn.naive_bayes import GaussianNB
        model = GaussianNB()
        model.fit(X, y);
```

Let's generate some new data and predict the label:

```
In [4]: rng = np.random.RandomState(0)
        Xnew = [-6, -14] + [14, 18] * rng.rand(2000, 2)
        ynew = model.predict(Xnew)
```

Now we can plot this new data to get an idea of where the decision boundary is (see Figure 41-3).

```
In [5]: plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu')
        lim = plt.axis()
        plt.scatter(Xnew[:, 0], Xnew[:, 1], c=ynew, s=20, cmap='RdBu', alpha=0.1)
        plt.axis(lim);
```

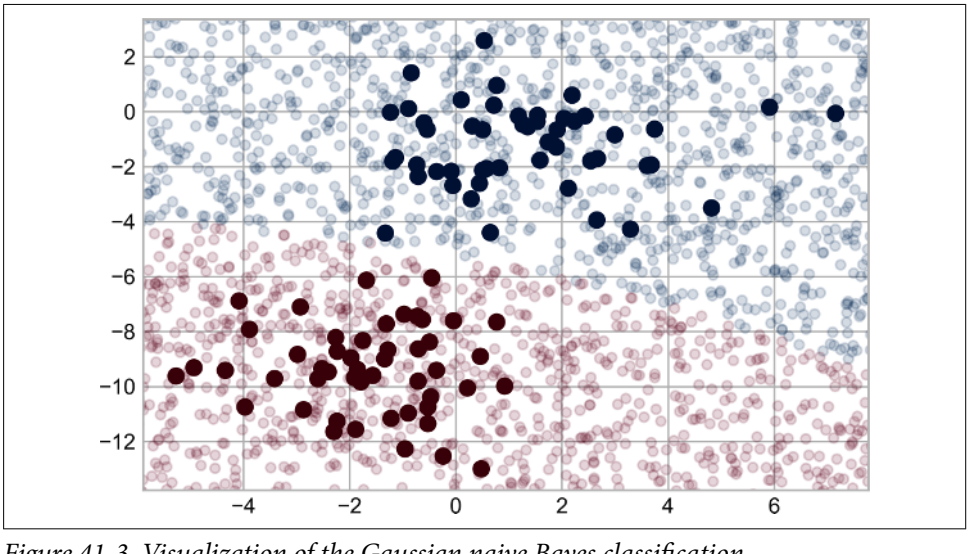

Figure 41-3. Visualization of the Gaussian naive Bayes classification

We see a slightly curved boundary in the classifications—in general, the boundary produced by a Gaussian naive Bayes model will be quadratic.

A nice aspect of this Bayesian formalism is that it naturally allows for probabilistic classification, which we can compute using the `predict_proba` method:

```
In [6]: yprob = model.predict_proba(Xnew)
        yprob[-8:].round(2)
Out[6]: array([[0.89, 0.11],
               [1.  , 0.  ],
               [1.  , 0.  ],
               [1.  , 0.  ],
               [1.  , 0.  ],
               [1.  , 0.  ],
               [0.  , 1.  ],
               [0.15, 0.85]])
```

The columns give the posterior probabilities of the first and second labels, respectively. If you are looking for estimates of uncertainty in your classification, Bayesian approaches like this can be a good place to start.

Of course, the final classification will only be as good as the model assumptions that lead to it, which is why Gaussian naive Bayes often does not produce very good results. Still, in many cases—especially as the number of features becomes large—this assumption is not detrimental enough to prevent Gaussian naive Bayes from being a reliable method.

Multinomial Naive Bayes

The Gaussian assumption just described is by no means the only simple assumption that could be used to specify the generative distribution for each label. Another useful example is multinomial naive Bayes, where the features are assumed to be generated from a simple multinomial distribution. The multinomial distribution describes the probability of observing counts among a number of categories, and thus multinomial naive Bayes is most appropriate for features that represent counts or count rates.

The idea is precisely the same as before, except that instead of modeling the data distribution with the best-fit Gaussian, we model it with a best-fit multinomial distribution.

Example: Classifying Text

One place where multinomial naive Bayes is often used is in text classification, where the features are related to word counts or frequencies within the documents to be classified. We discussed the extraction of such features from text in Chapter 40; here we will use the sparse word count features from the 20 Newsgroups corpus made available through Scikit-Learn to show how we might classify these short documents into categories.

Let's download the data and take a look at the target names:

```
In [7]: from sklearn.datasets import fetch_20newsgroups

        data = fetch_20newsgroups()
        data.target_names
Out[7]: ['alt.atheism',
         'comp.graphics',
         'comp.os.ms-windows.misc',
         'comp.sys.ibm.pc.hardware',
         'comp.sys.mac.hardware',
         'comp.windows.x',
         'misc.forsale',
         'rec.autos',
         'rec.motorcycles',
         'rec.sport.baseball',
         'rec.sport.hockey',
         'sci.crypt',
         'sci.electronics',
         'sci.med',
         'sci.space',
         'soc.religion.christian',
         'talk.politics.guns',
         'talk.politics.mideast',
         'talk.politics.misc',
         'talk.religion.misc']
```

For simplicity here, we will select just a few of these categories and download the training and testing sets:

```
In [8]: categories = ['talk.religion.misc', 'soc.religion.christian',
                      'sci.space', 'comp.graphics']
        train = fetch_20newsgroups(subset='train', categories=categories)
        test = fetch_20newsgroups(subset='test', categories=categories)
```

Here is a representative entry from the data:

```
In [9]: print(train.data[5][48:])
Out[9]: Subject: Federal Hearing
        Originator: dmcgee@uluhe
        Organization: School of Ocean and Earth Science and Technology
        Distribution: usa
        Lines: 10


        Fact or rumor....?  Madalyn Murray O'Hare an atheist who eliminated the
        use of the bible reading and prayer in public schools 15 years ago is now
        going to appear before the FCC with a petition to stop the reading of the
        Gospel on the airways of America.  And she is also campaigning to remove
        Christmas programs, songs, etc from the public schools.  If it is true
        then mail to Federal Communications Commission 1919 H Street Washington DC
        20054 expressing your opposition to her request.  Reference Petition number

        2493.
```

In order to use this data for machine learning, we need to be able to convert the content of each string into a vector of numbers. For this we will use the TF-IDF vectorizer (introduced in Chapter 40), and create a pipeline that attaches it to a multinomial naive Bayes classifier:

```
In [10]: from sklearn.feature_extraction.text import TfidfVectorizer
         from sklearn.naive_bayes import MultinomialNB
         from sklearn.pipeline import make_pipeline

         model = make_pipeline(TfidfVectorizer(), MultinomialNB())
```

With this pipeline, we can apply the model to the training data and predict labels for the test data:

```
In [11]: model.fit(train.data, train.target)
         labels = model.predict(test.data)
```

Now that we have predicted the labels for the test data, we can evaluate them to learn about the performance of the estimator. For example, let's take a look at the confusion matrix between the true and predicted labels for the test data (see Figure 41-4).

```
In [12]: from sklearn.metrics import confusion_matrix
         mat = confusion_matrix(test.target, labels)
         sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
                     xticklabels=train.target_names, yticklabels=train.target_names,
                     cmap='Blues')
         plt.xlabel('true label')
         plt.ylabel('predicted label');
```

Evidently, even this very simple classifier can successfully separate space discussions from computer discussions, but it gets confused between discussions about religion and discussions about Christianity. This is perhaps to be expected!

The cool thing here is that we now have the tools to determine the category for *any* string, using the `predict` method of this pipeline. Here's a utility function that will return the prediction for a single string:

```
In [13]: def predict_category(s, train=train, model=model):
             pred = model.predict([s])
             return train.target_names[pred[0]]
```

Let's try it out:

```
In [14]: predict_category('sending a payload to the ISS')
Out[14]: 'sci.space'

In [15]: predict_category('discussing the existence of God')
Out[15]: 'soc.religion.christian'

In [16]: predict_category('determining the screen resolution')
Out[16]: 'comp.graphics'
```

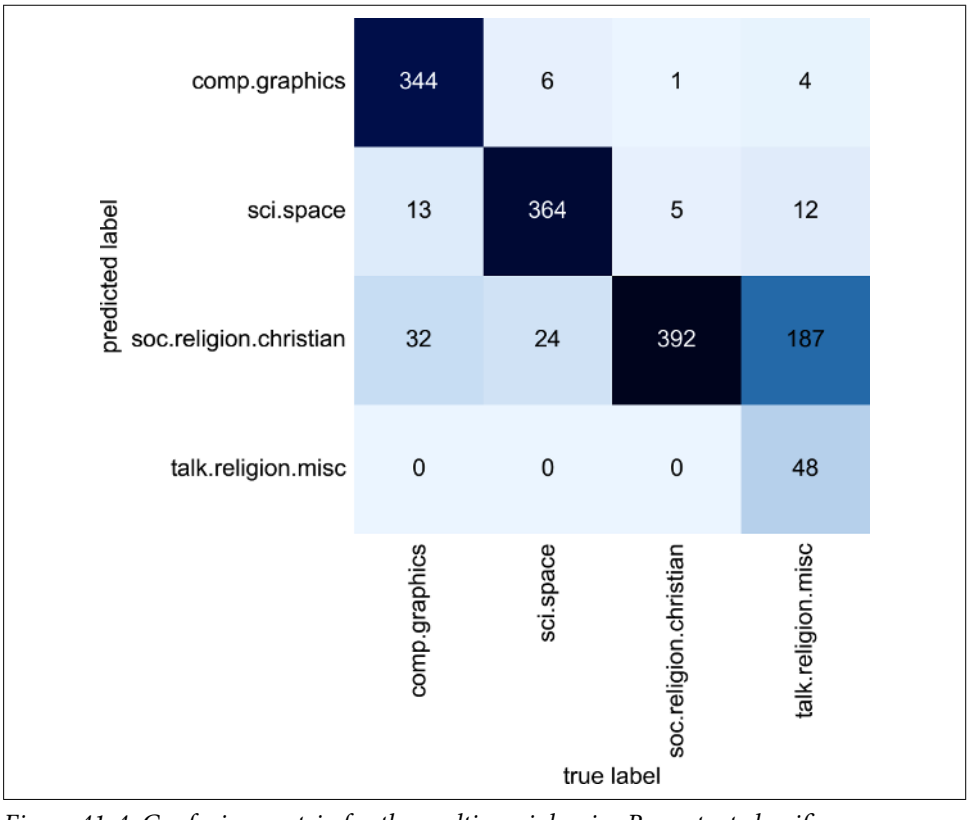

Figure 41-4. Confusion matrix for the multinomial naive Bayes text classifier

Remember that this is nothing more sophisticated than a simple probability model for the (weighted) frequency of each word in the string; nevertheless, the result is striking. Even a very naive algorithm, when used carefully and trained on a large set of high-dimensional data, can be surprisingly effective.

When to Use Naive Bayes

Because naive Bayes classifiers make such stringent assumptions about data, they will generally not perform as well as more complicated models. That said, they have several advantages:

- They are fast for both training and prediction.
- They provide straightforward probabilistic prediction.
- They are often easily interpretable.
- They have few (if any) tunable parameters.

These advantages mean a naive Bayes classifier is often a good choice as an initial baseline classification. If it performs suitably, then congratulations: you have a very fast, very interpretable classifier for your problem. If it does not perform well, then you can begin exploring more sophisticated models, with some baseline knowledge of how well they should perform.

Naive Bayes classifiers tend to perform especially well in the following situations:

- When the naive assumptions actually match the data (very rare in practice)
- For very well-separated categories, when model complexity is less important
- For very high-dimensional data, when model complexity is less important

The last two points seem distinct, but they actually are related: as the dimensionality of a dataset grows, it is much less likely for any two points to be found close together (after all, they must be close in *every single dimension* to be close overall). This means that clusters in high dimensions tend to be more separated, on average, than clusters in low dimensions, assuming the new dimensions actually add information. For this reason, simplistic classifiers like the ones discussed here tend to work as well or better than more complicated classifiers as the dimensionality grows: once you have enough data, even a simple model can be very powerful.

CHAPTER 17

Decision Trees

A tree is an incomprehensible mystery.
—Jim Woodring

DataSciencester's VP of Talent has interviewed a number of job candidates from the site, with varying degrees of success. He's collected a dataset consisting of several (qualitative) attributes of each candidate, as well as whether that candidate interviewed well or poorly. Could you, he asks, use this data to build a model identifying which candidates will interview well, so that he doesn't have to waste time conducting interviews?

This seems like a good fit for a *decision tree*, another predictive modeling tool in the data scientist's kit.

What Is a Decision Tree?

A decision tree uses a tree structure to represent a number of possible *decision paths* and an outcome for each path.

If you have ever played the game Twenty Questions, then you are familiar with decision trees. For example:

- "I am thinking of an animal."
- "Does it have more than five legs?"
- "No."
- "Is it delicious?"
- "No."
- "Does it appear on the back of the Australian five-cent coin?"

- "Yes."
- "Is it an echidna?"
- "Yes, it is!"

This corresponds to the path:

"Not more than 5 legs" → "Not delicious" → "On the 5-cent coin" → "Echidna!"

in an idiosyncratic (and not very comprehensive) "guess the animal" decision tree (Figure 17-1).

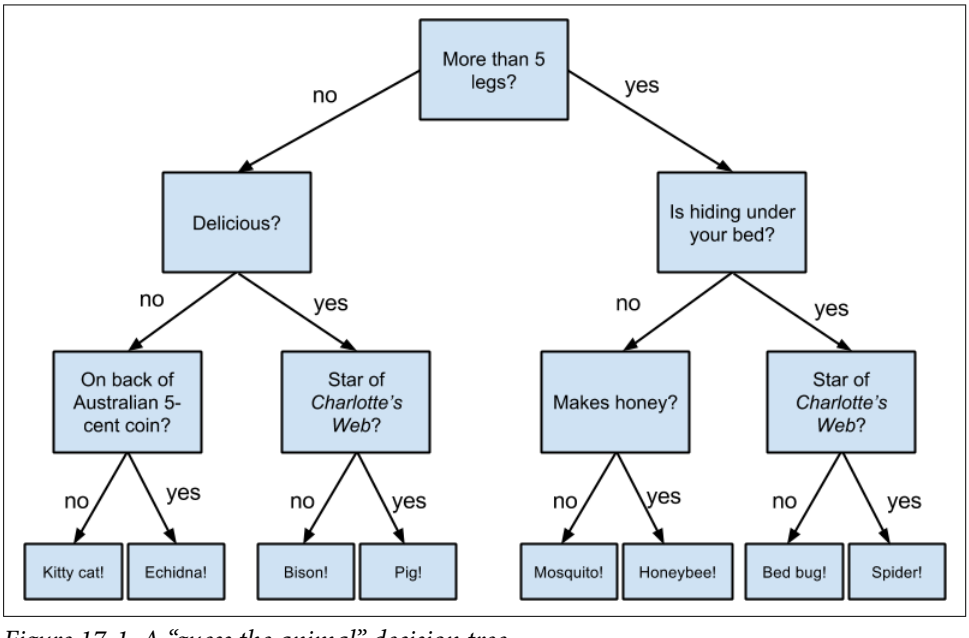

Figure 17-1. A "guess the animal" decision tree

Decision trees have a lot to recommend them. They're very easy to understand and interpret, and the process by which they reach a prediction is completely transparent. Unlike the other models we've looked at so far, decision trees can easily handle a mix of numeric (e.g., number of legs) and categorical (e.g., delicious/not delicious) attributes and can even classify data for which attributes are missing.

At the same time, finding an "optimal" decision tree for a set of training data is computationally a very hard problem. (We will get around this by trying to build a good-enough tree rather than an optimal one, although for large datasets this can still be a lot of work.) More important, it is very easy (and very bad) to build decision trees that are *overfitted* to the training data, and that don't generalize well to unseen data. We'll look at ways to address this.

Most people divide decision trees into *classification trees* (which produce categorical outputs) and *regression trees* (which produce numeric outputs). In this chapter, we'll focus on classification trees, and we'll work through the ID3 algorithm for learning a decision tree from a set of labeled data, which should help us understand how decision trees actually work. To make things simple, we'll restrict ourselves to problems with binary outputs like "Should I hire this candidate?" or "Should I show this website visitor advertisement A or advertisement B?" or "Will eating this food I found in the office fridge make me sick?"

Entropy

In order to build a decision tree, we will need to decide what questions to ask and in what order. At each stage of the tree there are some possibilities we've eliminated and some that we haven't. After learning that an animal doesn't have more than five legs, we've eliminated the possibility that it's a grasshopper. We haven't eliminated the possibility that it's a duck. Each possible question partitions the remaining possibilities according to its answer.

Ideally, we'd like to choose questions whose answers give a lot of information about what our tree should predict. If there's a single yes/no question for which "yes" answers always correspond to `True` outputs and "no" answers to `False` outputs (or vice versa), this would be an awesome question to pick. Conversely, a yes/no question for which neither answer gives you much new information about what the prediction should be is probably not a good choice.

We capture this notion of "how much information" with *entropy*. You have probably heard this term used to mean disorder. We use it to represent the uncertainty associated with data.

Imagine that we have a set S of data, each member of which is labeled as belonging to one of a finite number of classes $C_1, ..., C_n$. If all the data points belong to a single class, then there is no real uncertainty, which means we'd like there to be low entropy. If the data points are evenly spread across the classes, there is a lot of uncertainty and we'd like there to be high entropy.

In math terms, if p_i is the proportion of data labeled as class c_i, we define the entropy as:

$$H(S) = -p_1 \log_2 p_1 - ... - p_n \log_2 p_n$$

with the (standard) convention that $0 \log 0 = 0$.

Without worrying too much about the grisly details, each term $-p_i \log_2 p_i$ is non-negative and is close to 0 precisely when p_i is either close to 0 or close to 1 (Figure 17-2).

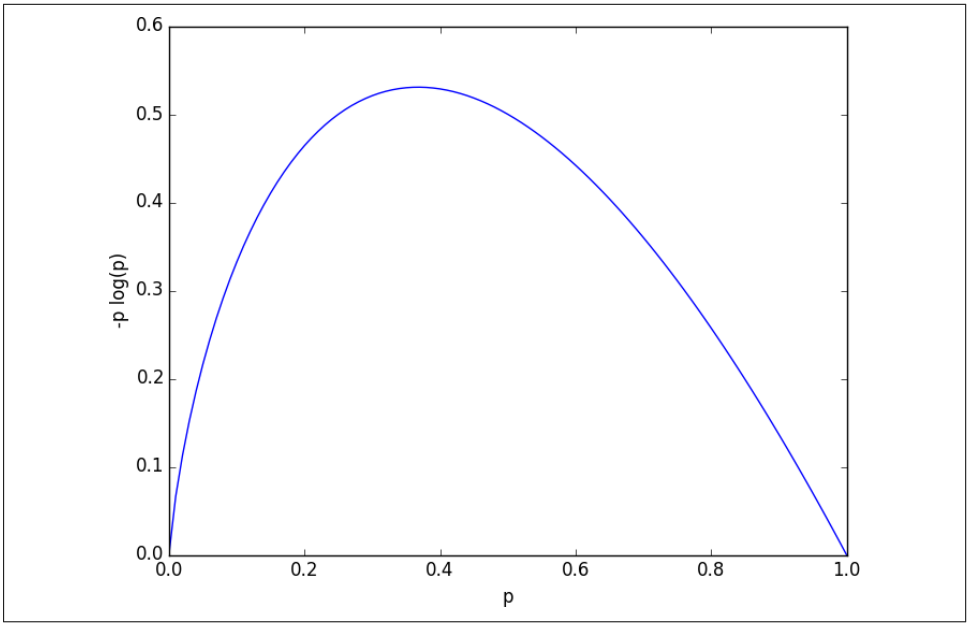

Figure 17-2. A graph of -p log p

This means the entropy will be small when every p_i is close to 0 or 1 (i.e., when most of the data is in a single class), and it will be larger when many of the p_i's are not close to 0 (i.e., when the data is spread across multiple classes). This is exactly the behavior we desire.

It is easy enough to roll all of this into a function:

```
from typing import List
import math

def entropy(class_probabilities: List[float]) -> float:
    """Given a list of class probabilities, compute the entropy"""
    return sum(-p * math.log(p, 2)
               for p in class_probabilities
               if p > 0)                     # ignore zero probabilities

assert entropy([1.0]) == 0
assert entropy([0.5, 0.5]) == 1
assert 0.81 < entropy([0.25, 0.75]) < 0.82
```

Our data will consist of pairs `(input, label)`, which means that we'll need to compute the class probabilities ourselves. Notice that we don't actually care which label is associated with each probability, only what the probabilities are:

```
from typing import Any
from collections import Counter

def class_probabilities(labels: List[Any]) -> List[float]:
    total_count = len(labels)
    return [count / total_count
            for count in Counter(labels).values()]

def data_entropy(labels: List[Any]) -> float:
    return entropy(class_probabilities(labels))

assert data_entropy(['a']) == 0
assert data_entropy([True, False]) == 1
assert data_entropy([3, 4, 4, 4]) == entropy([0.25, 0.75])
```

The Entropy of a Partition

What we've done so far is compute the entropy (think "uncertainty") of a single set of labeled data. Now, each stage of a decision tree involves asking a question whose answer partitions data into one or (hopefully) more subsets. For instance, our "does it have more than five legs?" question partitions animals into those that have more than five legs (e.g., spiders) and those that don't (e.g., echidnas).

Correspondingly, we'd like some notion of the entropy that results from partitioning a set of data in a certain way. We want a partition to have low entropy if it splits the data into subsets that themselves have low entropy (i.e., are highly certain), and high entropy if it contains subsets that (are large and) have high entropy (i.e., are highly uncertain).

For example, my "Australian five-cent coin" question was pretty dumb (albeit pretty lucky!), as it partitioned the remaining animals at that point into S_1 = {echidna} and S_2 = {everything else}, where S_2 is both large and high-entropy. (S_1 has no entropy, but it represents a small fraction of the remaining "classes.")

Mathematically, if we partition our data S into subsets $S_1, ..., S_m$ containing proportions $q_1, ..., q_m$ of the data, then we compute the entropy of the partition as a weighted sum:

$$H = q_1 H(S_1) + ... + q_m H(S_m)$$

which we can implement as:

```
def partition_entropy(subsets: List[List[Any]]) -> float:
    """Returns the entropy from this partition of data into subsets"""
    total_count = sum(len(subset) for subset in subsets)

    return sum(data_entropy(subset) * len(subset) / total_count
               for subset in subsets)
```

One problem with this approach is that partitioning by an attribute with many different values will result in a very low entropy due to overfitting. For example, imagine you work for a bank and are trying to build a decision tree to predict which of your customers are likely to default on their mortgages, using some historical data as your training set. Imagine further that the dataset contains each customer's Social Security number. Partitioning on SSN will produce one-person subsets, each of which necessarily has zero entropy. But a model that relies on SSN is *certain* not to generalize beyond the training set. For this reason, you should probably try to avoid (or bucket, if appropriate) attributes with large numbers of possible values when creating decision trees.

Creating a Decision Tree

The VP provides you with the interviewee data, consisting of (per your specification) a `NamedTuple` of the relevant attributes for each candidate—her level, her preferred language, whether she is active on Twitter, whether she has a PhD, and whether she interviewed well:

```
from typing import NamedTuple, Optional

class Candidate(NamedTuple):
    level: str
    lang: str
    tweets: bool
    phd: bool
    did_well: Optional[bool] = None  # allow unlabeled data

                  #  level     lang     tweets  phd  did_well
inputs = [Candidate('Senior', 'Java',   False, False, False),
          Candidate('Senior', 'Java',   False, True,  False),
          Candidate('Mid',    'Python', False, False, True),
          Candidate('Junior', 'Python', False, False, True),
          Candidate('Junior', 'R',      True,  False, True),
          Candidate('Junior', 'R',      True,  True,  False),
          Candidate('Mid',    'R',      True,  True,  True),
          Candidate('Senior', 'Python', False, False, False),
          Candidate('Senior', 'R',      True,  False, True),
          Candidate('Junior', 'Python', True,  False, True),
          Candidate('Senior', 'Python', True,  True,  True),
          Candidate('Mid',    'Python', False, True,  True),
```

```
    Candidate('Mid',    'Java',   True,  False, True),
    Candidate('Junior', 'Python', False, True,  False)
]
```

Our tree will consist of *decision nodes* (which ask a question and direct us differently depending on the answer) and *leaf nodes* (which give us a prediction). We will build it using the relatively simple *ID3* algorithm, which operates in the following manner. Let's say we're given some labeled data, and a list of attributes to consider branching on:

- If the data all have the same label, create a leaf node that predicts that label and then stop.
- If the list of attributes is empty (i.e., there are no more possible questions to ask), create a leaf node that predicts the most common label and then stop.
- Otherwise, try partitioning the data by each of the attributes.
- Choose the partition with the lowest partition entropy.
- Add a decision node based on the chosen attribute.
- Recur on each partitioned subset using the remaining attributes.

This is what's known as a "greedy" algorithm because, at each step, it chooses the most immediately best option. Given a dataset, there may be a better tree with a worse-looking first move. If so, this algorithm won't find it. Nonetheless, it is relatively easy to understand and implement, which makes it a good place to begin exploring decision trees.

Let's manually go through these steps on the interviewee dataset. The dataset has both `True` and `False` labels, and we have four attributes we can split on. So our first step will be to find the partition with the least entropy. We'll start by writing a function that does the partitioning:

```
from typing import Dict, TypeVar
from collections import defaultdict

T = TypeVar('T')  # generic type for inputs

def partition_by(inputs: List[T], attribute: str) -> Dict[Any, List[T]]:
    """Partition the inputs into lists based on the specified attribute."""
    partitions: Dict[Any, List[T]] = defaultdict(list)
    for input in inputs:
        key = getattr(input, attribute)  # value of the specified attribute
        partitions[key].append(input)    # add input to the correct partition
    return partitions
```

and one that uses it to compute entropy:

```
def partition_entropy_by(inputs: List[Any],
                         attribute: str,
```

```
                          label_attribute: str) -> float:
    """Compute the entropy corresponding to the given partition"""
    # partitions consist of our inputs
    partitions = partition_by(inputs, attribute)

    # but partition_entropy needs just the class labels
    labels = [[getattr(input, label_attribute) for input in partition]
              for partition in partitions.values()]

    return partition_entropy(labels)
```

Then we just need to find the minimum-entropy partition for the whole dataset:

```
for key in ['level','lang','tweets','phd']:
    print(key, partition_entropy_by(inputs, key, 'did_well'))

assert 0.69 < partition_entropy_by(inputs, 'level', 'did_well')  < 0.70
assert 0.86 < partition_entropy_by(inputs, 'lang', 'did_well')   < 0.87
assert 0.78 < partition_entropy_by(inputs, 'tweets', 'did_well') < 0.79
assert 0.89 < partition_entropy_by(inputs, 'phd', 'did_well')    < 0.90
```

The lowest entropy comes from splitting on `level`, so we'll need to make a subtree for each possible `level` value. Every `Mid` candidate is labeled `True`, which means that the `Mid` subtree is simply a leaf node predicting `True`. For `Senior` candidates, we have a mix of `True`s and `False`s, so we need to split again:

```
senior_inputs = [input for input in inputs if input.level == 'Senior']

assert 0.4 == partition_entropy_by(senior_inputs, 'lang', 'did_well')
assert 0.0 == partition_entropy_by(senior_inputs, 'tweets', 'did_well')
assert 0.95 < partition_entropy_by(senior_inputs, 'phd', 'did_well') < 0.96
```

This shows us that our next split should be on `tweets`, which results in a zero-entropy partition. For these `Senior`-level candidates, "yes" tweets always result in `True` while "no" tweets always result in `False`.

Finally, if we do the same thing for the `Junior` candidates, we end up splitting on `phd`, after which we find that no PhD always results in `True` and PhD always results in `False`.

Figure 17-3 shows the complete decision tree.

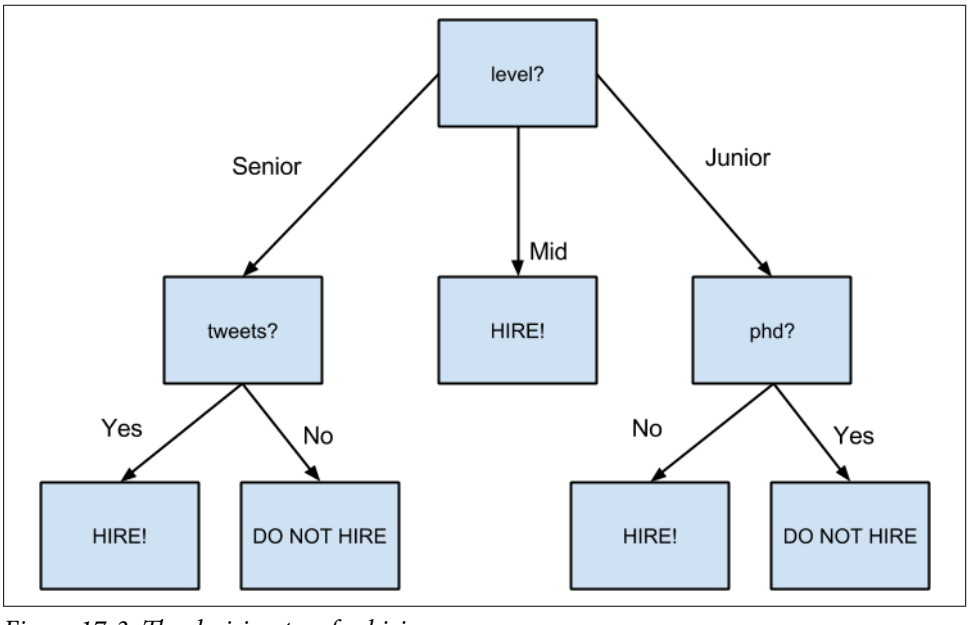

Figure 17-3. The decision tree for hiring

Putting It All Together

Now that we've seen how the algorithm works, we would like to implement it more generally. This means we need to decide how we want to represent trees. We'll use pretty much the most lightweight representation possible. We define a *tree* to be either:

- a `Leaf` (that predicts a single value), or
- a `Split` (containing an attribute to split on, subtrees for specific values of that attribute, and possibly a default value to use if we see an unknown value).

```
from typing import NamedTuple, Union, Any

class Leaf(NamedTuple):
    value: Any

class Split(NamedTuple):
    attribute: str
    subtrees: dict
    default_value: Any = None

DecisionTree = Union[Leaf, Split]
```

With this representation, our hiring tree would look like:

```
hiring_tree = Split('level', {   # first, consider "level"
    'Junior': Split('phd', {     # if level is "Junior", next look at "phd"
        False: Leaf(True),       #   if "phd" is False, predict True
        True: Leaf(False)        #   if "phd" is True, predict False
    }),
    'Mid': Leaf(True),           # if level is "Mid", just predict True
    'Senior': Split('tweets', {  # if level is "Senior", look at "tweets"
        False: Leaf(False),      #   if "tweets" is False, predict False
        True: Leaf(True)         #   if "tweets" is True, predict True
    })
})
```

There's still the question of what to do if we encounter an unexpected (or missing) attribute value. What should our hiring tree do if it encounters a candidate whose `level` is `Intern`? We'll handle this case by populating the `default_value` attribute with the most common label.

Given such a representation, we can classify an input with:

```
def classify(tree: DecisionTree, input: Any) -> Any:
    """classify the input using the given decision tree"""

    # If this is a leaf node, return its value
    if isinstance(tree, Leaf):
        return tree.value

    # Otherwise this tree consists of an attribute to split on
    # and a dictionary whose keys are values of that attribute
    # and whose values are subtrees to consider next
    subtree_key = getattr(input, tree.attribute)

    if subtree_key not in tree.subtrees:   # If no subtree for key,
        return tree.default_value          # return the default value.

    subtree = tree.subtrees[subtree_key]   # Choose the appropriate subtree
    return classify(subtree, input)        # and use it to classify the input.
```

All that's left is to build the tree representation from our training data:

```
def build_tree_id3(inputs: List[Any],
                   split_attributes: List[str],
                   target_attribute: str) -> DecisionTree:
    # Count target labels
    label_counts = Counter(getattr(input, target_attribute)
                           for input in inputs)
    most_common_label = label_counts.most_common(1)[0][0]

    # If there's a unique label, predict it
    if len(label_counts) == 1:
        return Leaf(most_common_label)

    # If no split attributes left, return the majority label
    if not split_attributes:
```

```
        return Leaf(most_common_label)

    # Otherwise split by the best attribute

    def split_entropy(attribute: str) -> float:
        """Helper function for finding the best attribute"""
        return partition_entropy_by(inputs, attribute, target_attribute)

    best_attribute = min(split_attributes, key=split_entropy)

    partitions = partition_by(inputs, best_attribute)
    new_attributes = [a for a in split_attributes if a != best_attribute]

    # Recursively build the subtrees
    subtrees = {attribute_value : build_tree_id3(subset,
                                                 new_attributes,
                                                 target_attribute)
                for attribute_value, subset in partitions.items()}

    return Split(best_attribute, subtrees, default_value=most_common_label)
```

In the tree we built, every leaf consisted entirely of `True` inputs or entirely of `False` inputs. This means that the tree predicts perfectly on the training dataset. But we can also apply it to new data that wasn't in the training set:

```
tree = build_tree_id3(inputs,
                      ['level', 'lang', 'tweets', 'phd'],
                      'did_well')

# Should predict True
assert classify(tree, Candidate("Junior", "Java", True, False))

# Should predict False
assert not classify(tree, Candidate("Junior", "Java", True, True))
```

And also to data with unexpected values:

```
# Should predict True
assert classify(tree, Candidate("Intern", "Java", True, True))
```

Since our goal was mainly to demonstrate *how* to build a tree, we built the tree using the entire dataset. As always, if we were really trying to create a good model for something, we would have collected more data and split it into train/validation/test subsets.

Random Forests

Given how closely decision trees can fit themselves to their training data, it's not surprising that they have a tendency to overfit. One way of avoiding this is a technique called *random forests*, in which we build multiple decision trees and combine their

outputs. If they're classification trees, we might let them vote; if they're regression trees, we might average their predictions.

Our tree-building process was deterministic, so how do we get random trees?

One piece involves bootstrapping data (recall "Digression: The Bootstrap" on page 190). Rather than training each tree on all the `inputs` in the training set, we train each tree on the result of `bootstrap_sample(inputs)`. Since each tree is built using different data, each tree will be different from every other tree. (A side benefit is that it's totally fair to use the nonsampled data to test each tree, which means you can get away with using all of your data as the training set if you are clever in how you measure performance.) This technique is known as *bootstrap aggregating* or *bagging*.

A second source of randomness involves changing the way we choose the `best_attribute` to split on. Rather than looking at all the remaining attributes, we first choose a random subset of them and then split on whichever of those is best:

```
# if there are already few enough split candidates, look at all of them
if len(split_candidates) <= self.num_split_candidates:
    sampled_split_candidates = split_candidates
# otherwise pick a random sample
else:
    sampled_split_candidates = random.sample(split_candidates,
                                             self.num_split_candidates)

# now choose the best attribute only from those candidates
best_attribute = min(sampled_split_candidates, key=split_entropy)

partitions = partition_by(inputs, best_attribute)
```

This is an example of a broader technique called *ensemble learning* in which we combine several *weak learners* (typically high-bias, low-variance models) in order to produce an overall strong model.

For Further Exploration

- scikit-learn has many decision tree models. It also has an `ensemble` module that includes a `RandomForestClassifier` as well as other ensemble methods.
- XGBoost is a library for training *gradient boosted* decision trees that tends to win a lot of Kaggle-style machine learning competitions.
- We've barely scratched the surface of decision trees and their algorithms. Wikipedia is a good starting point for broader exploration.

CHAPTER 44

In Depth: Decision Trees and Random Forests

Previously we have looked in depth at a simple generative classifier (naive Bayes; see Chapter 41) and a powerful discriminative classifier (support vector machines; see Chapter 43). Here we'll take a look at another powerful algorithm: a nonparametric algorithm called *random forests*. Random forests are an example of an *ensemble* method, meaning one that relies on aggregating the results of a set of simpler estimators. The somewhat surprising result with such ensemble methods is that the sum can be greater than the parts: that is, the predictive accuracy of a majority vote among a number of estimators can end up being better than that of any of the individual estimators doing the voting! We will see examples of this in the following sections.

We begin with the standard imports:

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
```

Motivating Random Forests: Decision Trees

Random forests are an example of an ensemble learner built on decision trees. For this reason, we'll start by discussing decision trees themselves.

Decision trees are extremely intuitive ways to classify or label objects: you simply ask a series of questions designed to zero in on the classification. For example, if you wanted to build a decision tree to classify animals you come across while on a hike, you might construct the one shown in Figure 44-1.

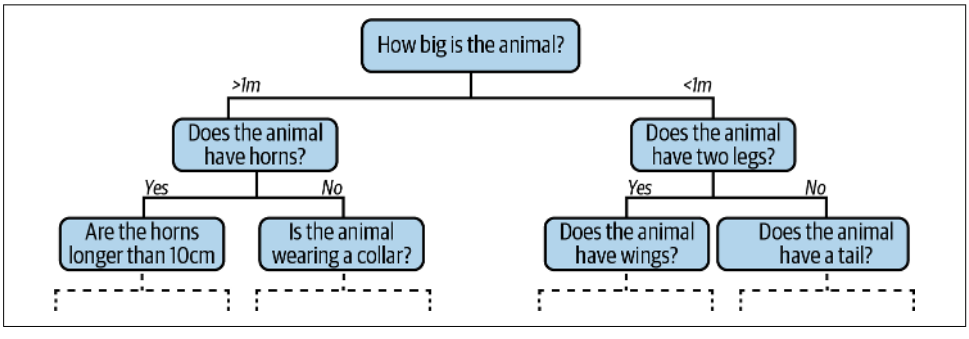

Figure 44-1. An example of a binary decision tree[1]

The binary splitting makes this extremely efficient: in a well-constructed tree, each question will cut the number of options by approximately half, very quickly narrowing the options even among a large number of classes. The trick, of course, comes in deciding which questions to ask at each step. In machine learning implementations of decision trees, the questions generally take the form of axis-aligned splits in the data: that is, each node in the tree splits the data into two groups using a cutoff value within one of the features. Let's now look at an example of this.

Creating a Decision Tree

Consider the following two-dimensional data, which has one of four class labels (see Figure 44-2).

```
In [2]: from sklearn.datasets import make_blobs

        X, y = make_blobs(n_samples=300, centers=4,
                          random_state=0, cluster_std=1.0)
        plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='rainbow');
```

1 Code to produce this figure can be found in the online appendix.

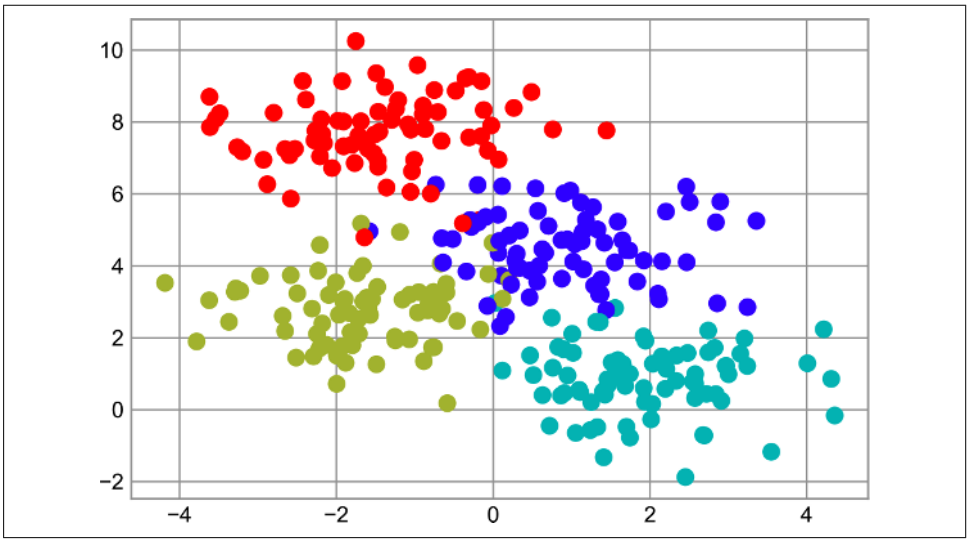

Figure 44-2. Data for the decision tree classifier

A simple decision tree built on this data will iteratively split the data along one or the other axis according to some quantitative criterion, and at each level assign the label of the new region according to a majority vote of points within it. Figure 44-3 presents a visualization of the first four levels of a decision tree classifier for this data.

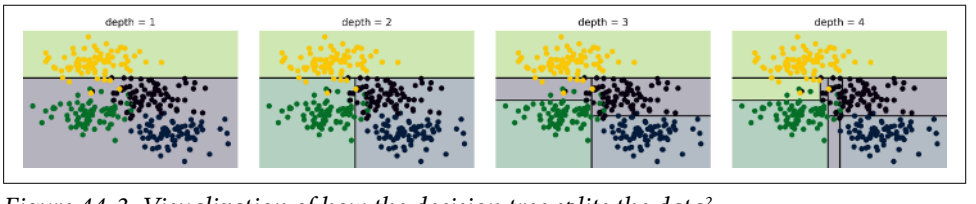

Figure 44-3. Visualization of how the decision tree splits the data[2]

Notice that after the first split, every point in the upper branch remains unchanged, so there is no need to further subdivide this branch. Except for nodes that contain all of one color, at each level *every* region is again split along one of the two features.

This process of fitting a decision tree to our data can be done in Scikit-Learn with the `DecisionTreeClassifier` estimator:

```
In [3]: from sklearn.tree import DecisionTreeClassifier
        tree = DecisionTreeClassifier().fit(X, y)
```

2 Code to produce this figure can be found in the online appendix.

Let's write a utility function to help us visualize the output of the classifier:

```
In [4]: def visualize_classifier(model, X, y, ax=None, cmap='rainbow'):
            ax = ax or plt.gca()

            # Plot the training points
            ax.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=cmap,
                       clim=(y.min(), y.max()), zorder=3)
            ax.axis('tight')
            ax.axis('off')
            xlim = ax.get_xlim()
            ylim = ax.get_ylim()

            # fit the estimator
            model.fit(X, y)
            xx, yy = np.meshgrid(np.linspace(*xlim, num=200),
                                 np.linspace(*ylim, num=200))
            Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

            # Create a color plot with the results
            n_classes = len(np.unique(y))
            contours = ax.contourf(xx, yy, Z, alpha=0.3,
                                   levels=np.arange(n_classes + 1) - 0.5,
                                   cmap=cmap, zorder=1)

            ax.set(xlim=xlim, ylim=ylim)
```

Now we can examine what the decision tree classification looks like (see Figure 44-4).

```
In [5]: visualize_classifier(DecisionTreeClassifier(), X, y)
```

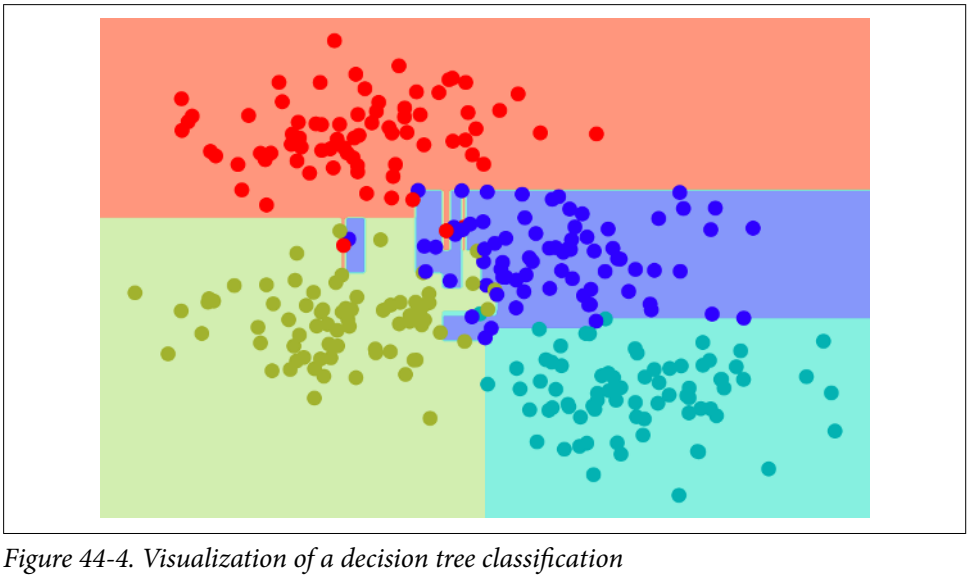

Figure 44-4. Visualization of a decision tree classification

If you're running this notebook live, you can use the helper script included in the online appendix to bring up an interactive visualization of the decision tree building process:

```
In [6]: # helpers_05_08 is found in the online appendix
        import helpers_05_08
        helpers_05_08.plot_tree_interactive(X, y);
Out[6]: interactive(children=(Dropdown(description='depth', index=1, options=(1, 5),
         > value=5), Output()), _dom_classes...
```

Notice that as the depth increases, we tend to get very strangely shaped classification regions; for example, at a depth of five, there is a tall and skinny purple region between the yellow and blue regions. It's clear that this is less a result of the true, intrinsic data distribution, and more a result of the particular sampling or noise properties of the data. That is, this decision tree, even at only five levels deep, is clearly overfitting our data.

Decision Trees and Overfitting

Such overfitting turns out to be a general property of decision trees: it is very easy to go too deep in the tree, and thus to fit details of the particular data rather than the overall properties of the distributions it is drawn from. Another way to see this overfitting is to look at models trained on different subsets of the data—for example, in Figure 44-5 we train two different trees, each on half of the original data.

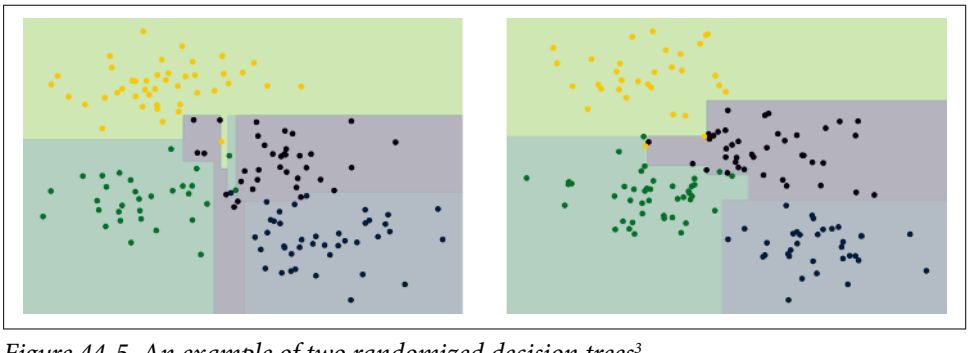

Figure 44-5. An example of two randomized decision trees[3]

It is clear that in some places the two trees produce consistent results (e.g., in the four corners), while in other places the two trees give very different classifications (e.g., in the regions between any two clusters). The key observation is that the inconsistencies tend to happen where the classification is less certain, and thus by using information from *both* of these trees, we might come up with a better result!

3 Code to produce this figure can be found in the online appendix.

If you are running this notebook live, the following function will allow you to interactively display the fits of trees trained on a random subset of the data:

```
In [7]: # helpers_05_08 is found in the online appendix
        import helpers_05_08
        helpers_05_08.randomized_tree_interactive(X, y)
Out[7]: interactive(children=(Dropdown(description='random_state', options=(0, 100),
         > value=0), Output()), _dom_classes...
```

Just as using information from two trees improves our results, we might expect that using information from many trees would improve our results even further.

Ensembles of Estimators: Random Forests

This notion—that multiple overfitting estimators can be combined to reduce the effect of this overfitting—is what underlies an ensemble method called *bagging*. Bagging makes use of an ensemble (a grab bag, perhaps) of parallel estimators, each of which overfits the data, and averages the results to find a better classification. An ensemble of randomized decision trees is known as a *random forest*.

This type of bagging classification can be done manually using Scikit-Learn's `Bagging Classifier` meta-estimator, as shown here (see Figure 44-6).

```
In [8]: from sklearn.tree import DecisionTreeClassifier
        from sklearn.ensemble import BaggingClassifier

        tree = DecisionTreeClassifier()
        bag = BaggingClassifier(tree, n_estimators=100, max_samples=0.8,
                                random_state=1)

        bag.fit(X, y)
        visualize_classifier(bag, X, y)
```

In this example, we have randomized the data by fitting each estimator with a random subset of 80% of the training points. In practice, decision trees are more effectively randomized by injecting some stochasticity in how the splits are chosen: this way all the data contributes to the fit each time, but the results of the fit still have the desired randomness. For example, when determining which feature to split on, the randomized tree might select from among the top several features. You can read more technical details about these randomization strategies in the Scikit-Learn documentation and references within.

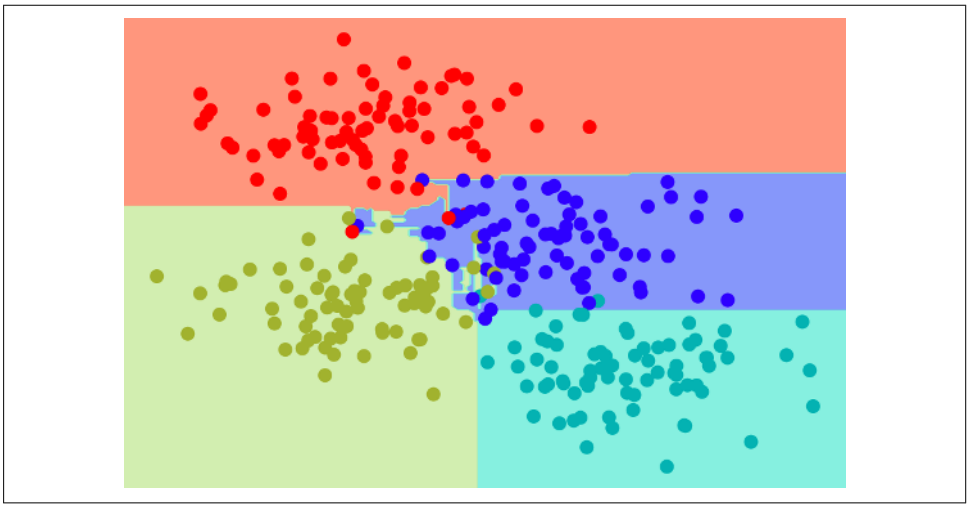

Figure 44-6. Decision boundaries for an ensemble of random decision trees

In Scikit-Learn, such an optimized ensemble of randomized decision trees is implemented in the `RandomForestClassifier` estimator, which takes care of all the randomization automatically. All you need to do is select a number of estimators, and it will very quickly—in parallel, if desired—fit the ensemble of trees (see Figure 44-7).

```
In [9]: from sklearn.ensemble import RandomForestClassifier

        model = RandomForestClassifier(n_estimators=100, random_state=0)
        visualize_classifier(model, X, y);
```

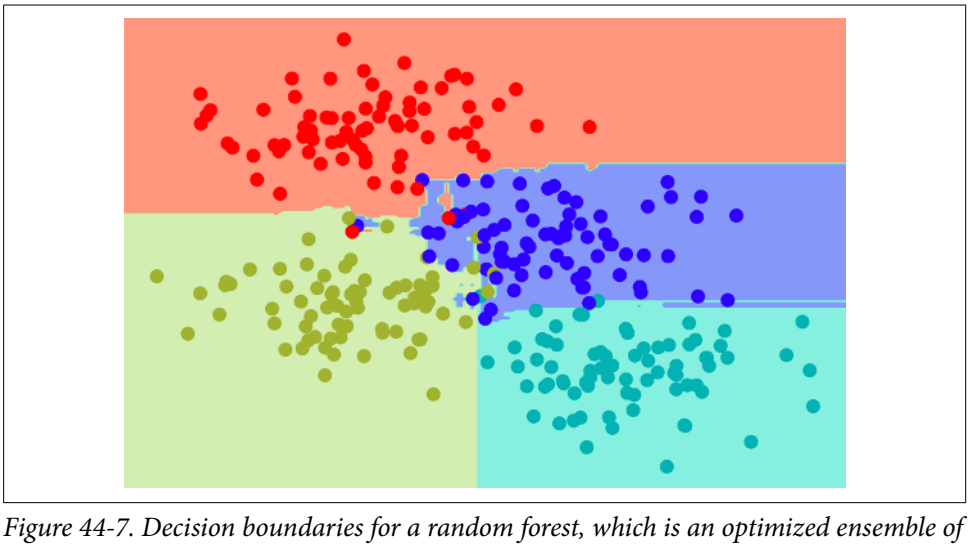

Figure 44-7. Decision boundaries for a random forest, which is an optimized ensemble of decision trees

We see that by averaging over one hundred randomly perturbed models, we end up with an overall model that is much closer to our intuition about how the parameter space should be split.

Random Forest Regression

In the previous section we considered random forests within the context of classification. Random forests can also be made to work in the case of regression (that is, with continuous rather than categorical variables). The estimator to use for this is the `RandomForestRegressor`, and the syntax is very similar to what we saw earlier.

Consider the following data, drawn from the combination of a fast and slow oscillation (see Figure 44-8).

```
In [10]: rng = np.random.RandomState(42)
         x = 10 * rng.rand(200)

         def model(x, sigma=0.3):
             fast_oscillation = np.sin(5 * x)
             slow_oscillation = np.sin(0.5 * x)
             noise = sigma * rng.randn(len(x))

             return slow_oscillation + fast_oscillation + noise

         y = model(x)
         plt.errorbar(x, y, 0.3, fmt='o');
```

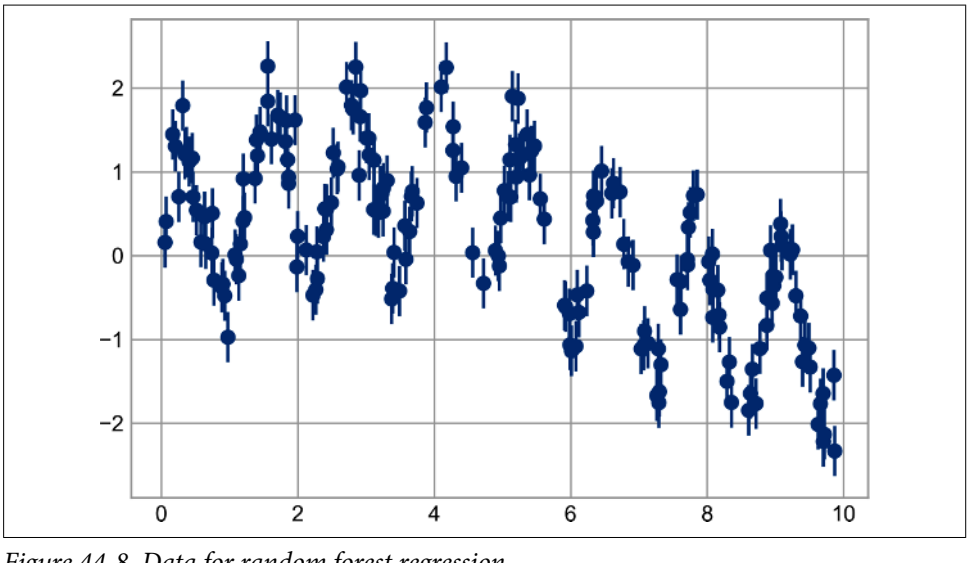

Figure 44-8. Data for random forest regression

Using the random forest regressor, we can find the best-fit curve (Figure 44-9).

```
In [11]: from sklearn.ensemble import RandomForestRegressor
         forest = RandomForestRegressor(200)
         forest.fit(x[:, None], y)

         xfit = np.linspace(0, 10, 1000)
         yfit = forest.predict(xfit[:, None])
         ytrue = model(xfit, sigma=0)

         plt.errorbar(x, y, 0.3, fmt='o', alpha=0.5)
         plt.plot(xfit, yfit, '-r');
         plt.plot(xfit, ytrue, '-k', alpha=0.5);
```

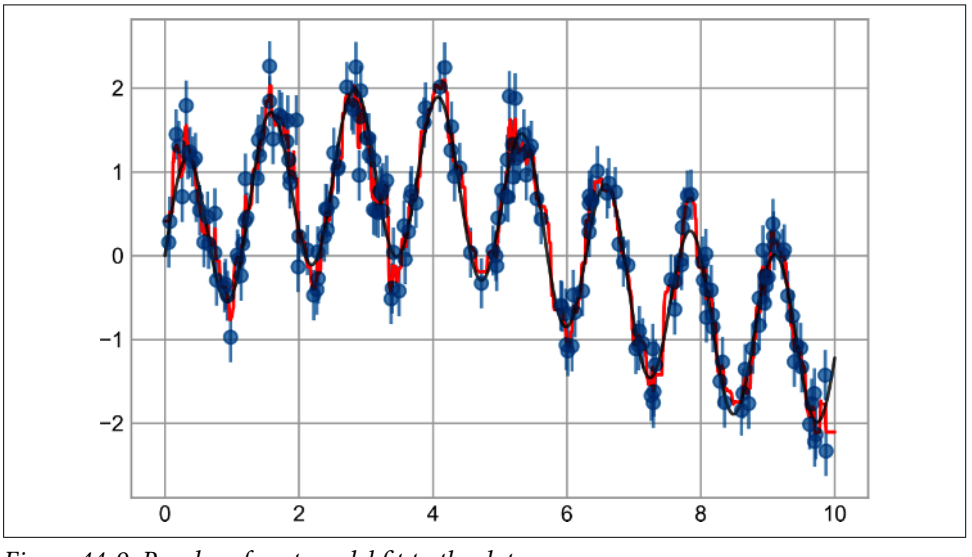

Figure 44-9. Random forest model fit to the data

Here the true model is shown in the smooth gray curve, while the random forest model is shown by the jagged red curve. The nonparametric random forest model is flexible enough to fit the multiperiod data, without us needing to specifying a multi-period model!

Example: Random Forest for Classifying Digits

In Chapter 38, we worked through an example using the digits dataset included with Scikit-Learn. Let's use that again here to see how the random forest classifier can be applied in this context:

```
In [12]: from sklearn.datasets import load_digits
         digits = load_digits()
         digits.keys()
Out[12]: dict_keys(['data', 'target', 'frame', 'feature_names', 'target_names',
          > 'images', 'DESCR'])
```

To remind us what we're looking at, we'll visualize the first few data points (see Figure 44-10).

```
In [13]: # set up the figure
         fig = plt.figure(figsize=(6, 6))  # figure size in inches
         fig.subplots_adjust(left=0, right=1, bottom=0, top=1,
                             hspace=0.05, wspace=0.05)

         # plot the digits: each image is 8x8 pixels
         for i in range(64):
             ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
             ax.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='nearest')

             # label the image with the target value
             ax.text(0, 7, str(digits.target[i]))
```

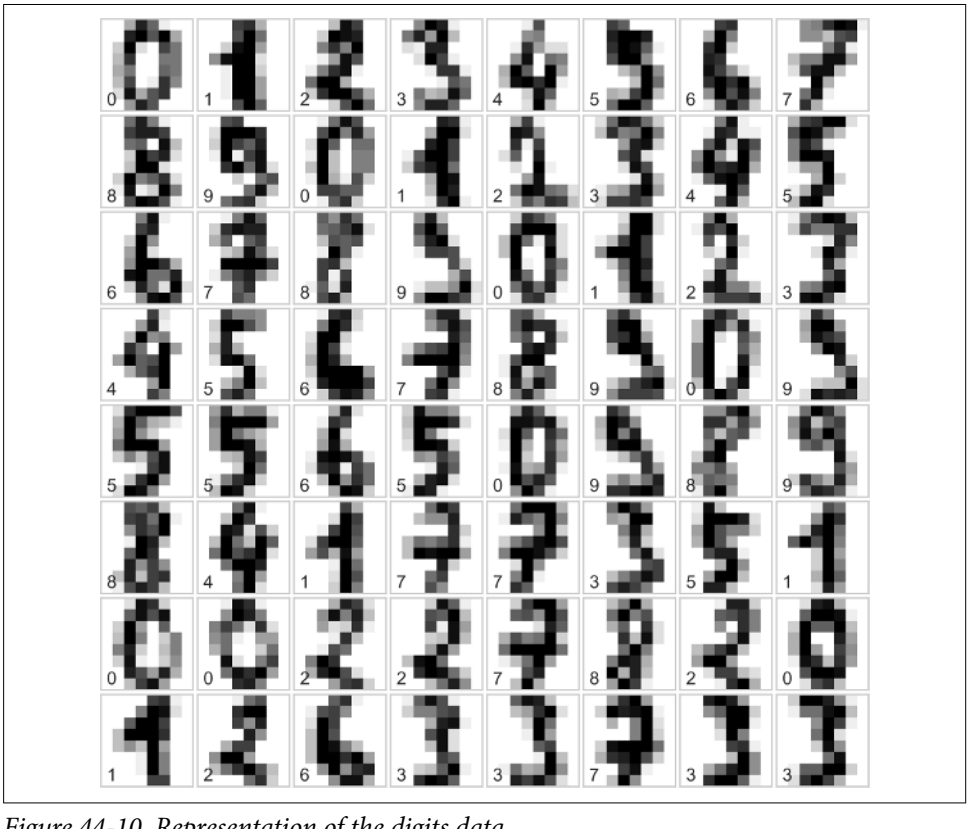

Figure 44-10. Representation of the digits data

We can classify the digits using a random forest as follows:

```
In [14]: from sklearn.model_selection import train_test_split

         Xtrain, Xtest, ytrain, ytest = train_test_split(digits.data, digits.target,
                                                         random_state=0)
         model = RandomForestClassifier(n_estimators=1000)
         model.fit(Xtrain, ytrain)
         ypred = model.predict(Xtest)
```

Let's look at the classification report for this classifier:

```
In [15]: from sklearn import metrics
         print(metrics.classification_report(ypred, ytest))
Out[15]:               precision    recall  f1-score   support

                    0       1.00      0.97      0.99        38
                    1       0.98      0.98      0.98        43
                    2       0.95      1.00      0.98        42
                    3       0.98      0.96      0.97        46
                    4       0.97      1.00      0.99        37
                    5       0.98      0.96      0.97        49
                    6       1.00      1.00      1.00        52
                    7       1.00      0.96      0.98        50
                    8       0.94      0.98      0.96        46
                    9       0.98      0.98      0.98        47

             accuracy                           0.98       450
            macro avg       0.98      0.98      0.98       450
         weighted avg       0.98      0.98      0.98       450
```

And for good measure, plot the confusion matrix (see Figure 44-11).

```
In [16]: from sklearn.metrics import confusion_matrix
         import seaborn as sns
         mat = confusion_matrix(ytest, ypred)
         sns.heatmap(mat.T, square=True, annot=True, fmt='d',
                     cbar=False, cmap='Blues')
         plt.xlabel('true label')
         plt.ylabel('predicted label');
```

We find that a simple, untuned random forest results in a quite accurate classification of the digits data.

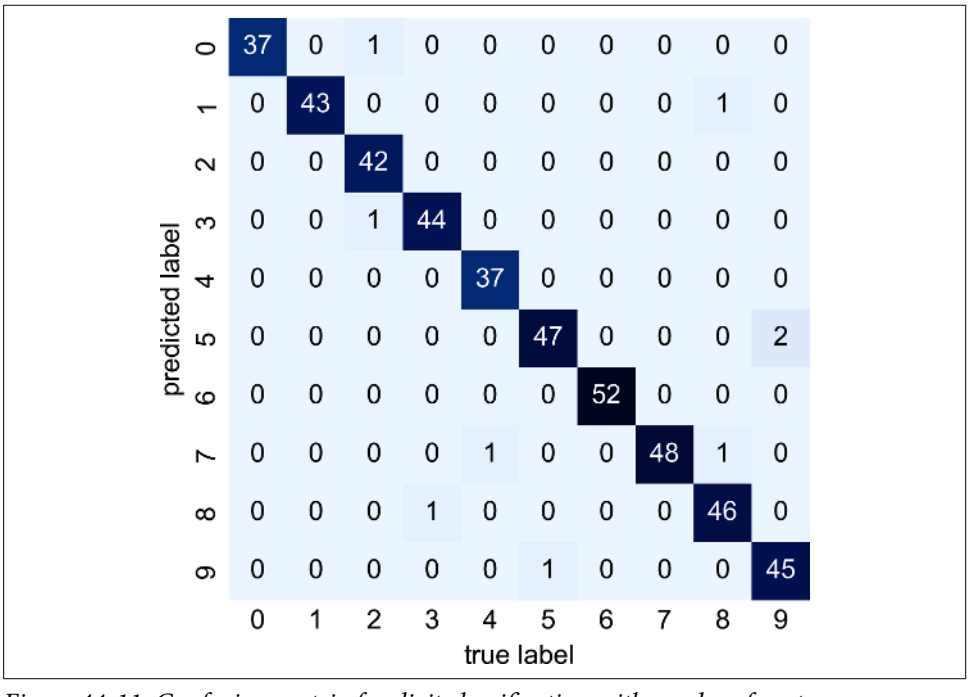

Figure 44-11. Confusion matrix for digit classification with random forests

Summary

This chapter provided a brief introduction to the concept of ensemble estimators, and in particular the random forest, an ensemble of randomized decision trees. Random forests are a powerful method with several advantages:

- Both training and prediction are very fast, because of the simplicity of the underlying decision trees. In addition, both tasks can be straightforwardly parallelized, because the individual trees are entirely independent entities.
- The multiple trees allow for a probabilistic classification: a majority vote among estimators gives an estimate of the probability (accessed in Scikit-Learn with the `predict_proba` method).
- The nonparametric model is extremely flexible and can thus perform well on tasks that are underfit by other estimators.

A primary disadvantage of random forests is that the results are not easily interpretable: that is, if you would like to draw conclusions about the *meaning* of the classification model, random forests may not be the best choice.

CHAPTER 43

In Depth: Support Vector Machines

Support vector machines (SVMs) are a particularly powerful and flexible class of supervised algorithms for both classification and regression. In this chapter, we will explore the intuition behind SVMs and their use in classification problems.

We begin with the standard imports:

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
        from scipy import stats
```

Full-size, full-color figures are available in the supplemental materials on GitHub.

Motivating Support Vector Machines

As part of our discussion of Bayesian classification (see Chapter 41), we learned about a simple kind of model that describes the distribution of each underlying class, and experimented with using it to probabilistically determine labels for new points. That was an example of *generative classification*; here we will consider instead *discriminative classification*. That is, rather than modeling each class, we will simply find a line or curve (in two dimensions) or manifold (in multiple dimensions) that divides the classes from each other.

As an example of this, consider the simple case of a classification task in which the two classes of points are well separated (see Figure 43-1).

```
In [2]: from sklearn.datasets import make_blobs
        X, y = make_blobs(n_samples=50, centers=2,
                          random_state=0, cluster_std=0.60)
        plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');
```

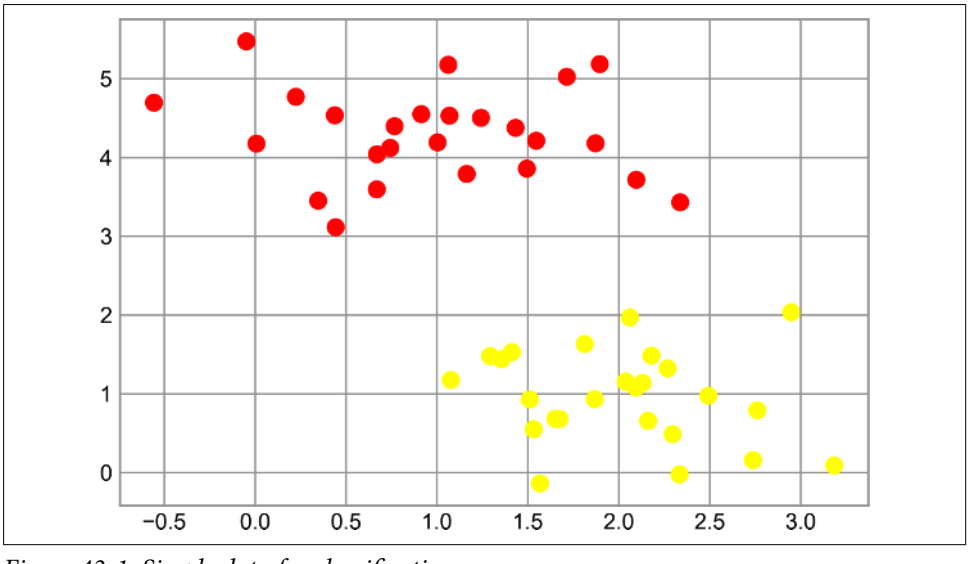

Figure 43-1. Simple data for classification

A linear discriminative classifier would attempt to draw a straight line separating the two sets of data, and thereby create a model for classification. For two-dimensional data like that shown here, this is a task we could do by hand. But immediately we see a problem: there is more than one possible dividing line that can perfectly discriminate between the two classes!

We can draw some of them as follows; Figure 43-2 shows the result:

```
In [3]: xfit = np.linspace(-1, 3.5)
        plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
        plt.plot([0.6], [2.1], 'x', color='red', markeredgewidth=2, markersize=10)

        for m, b in [(1, 0.65), (0.5, 1.6), (-0.2, 2.9)]:
            plt.plot(xfit, m * xfit + b, '-k')

        plt.xlim(-1, 3.5);
```

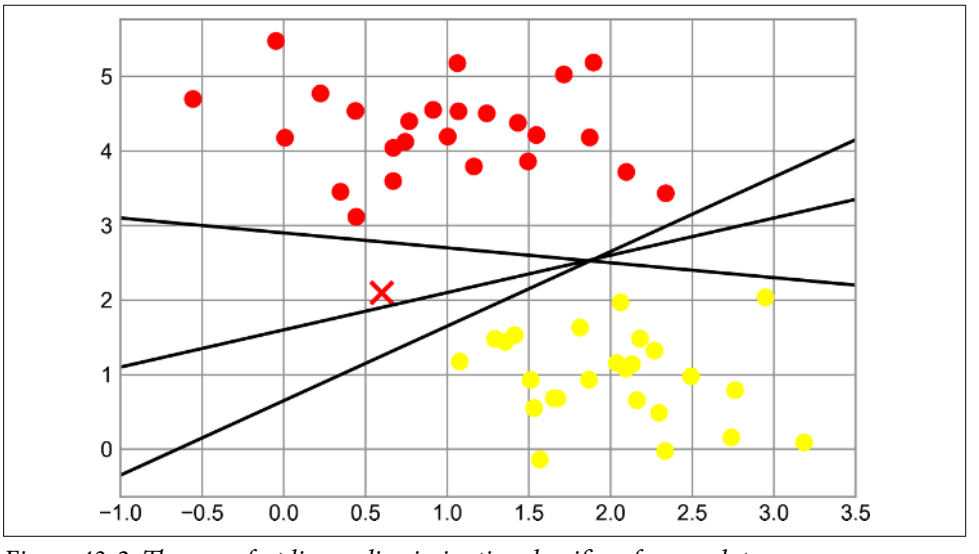

Figure 43-2. Three perfect linear discriminative classifiers for our data

These are three *very* different separators which, nevertheless, perfectly discriminate between these samples. Depending on which you choose, a new data point (e.g., the one marked by the "X" in this plot) will be assigned a different label! Evidently our simple intuition of "drawing a line between classes" is not good enough, and we need to think a bit more deeply.

Support Vector Machines: Maximizing the Margin

Support vector machines offer one way to improve on this. The intuition is this: rather than simply drawing a zero-width line between the classes, we can draw around each line a *margin* of some width, up to the nearest point. Here is an example of how this might look (Figure 43-3).

```
In [4]: xfit = np.linspace(-1, 3.5)
        plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')

        for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:
            yfit = m * xfit + b
            plt.plot(xfit, yfit, '-k')
            plt.fill_between(xfit, yfit - d, yfit + d, edgecolor='none',
                             color='lightgray', alpha=0.5)

        plt.xlim(-1, 3.5);
```

The line that maximizes this margin is the one we will choose as the optimal model.

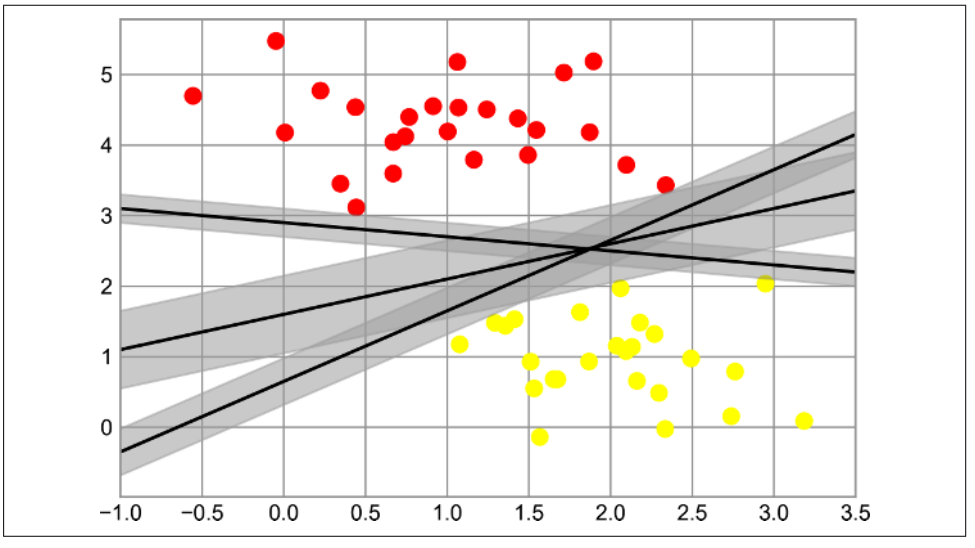

Figure 43-3. Visualization of "margins" within discriminative classifiers

Fitting a Support Vector Machine

Let's see the result of an actual fit to this data: we will use Scikit-Learn's support vector classifier (`SVC`) to train an SVM model on this data. For the time being, we will use a linear kernel and set the `C` parameter to a very large number (we'll discuss the meaning of these in more depth momentarily):

```
In [5]: from sklearn.svm import SVC # "Support vector classifier"
        model = SVC(kernel='linear', C=1E10)
        model.fit(X, y)
Out[5]: SVC(C=10000000000.0, kernel='linear')
```

To better visualize what's happening here, let's create a quick convenience function that will plot SVM decision boundaries for us (Figure 43-4).

```
In [6]: def plot_svc_decision_function(model, ax=None, plot_support=True):
            """Plot the decision function for a 2D SVC"""
            if ax is None:
                ax = plt.gca()
            xlim = ax.get_xlim()
            ylim = ax.get_ylim()

            # create grid to evaluate model
            x = np.linspace(xlim[0], xlim[1], 30)
            y = np.linspace(ylim[0], ylim[1], 30)
            Y, X = np.meshgrid(y, x)
            xy = np.vstack([X.ravel(), Y.ravel()]).T
            P = model.decision_function(xy).reshape(X.shape)

            # plot decision boundary and margins
```

```
            ax.contour(X, Y, P, colors='k',
                       levels=[-1, 0, 1], alpha=0.5,
                       linestyles=['--', '-', '--'])

            # plot support vectors
            if plot_support:
                ax.scatter(model.support_vectors_[:, 0],
                           model.support_vectors_[:, 1],
                           s=300, linewidth=1, edgecolors='black',
                           facecolors='none');
            ax.set_xlim(xlim)
            ax.set_ylim(ylim)
```

```
In [7]: plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
        plot_svc_decision_function(model);
```

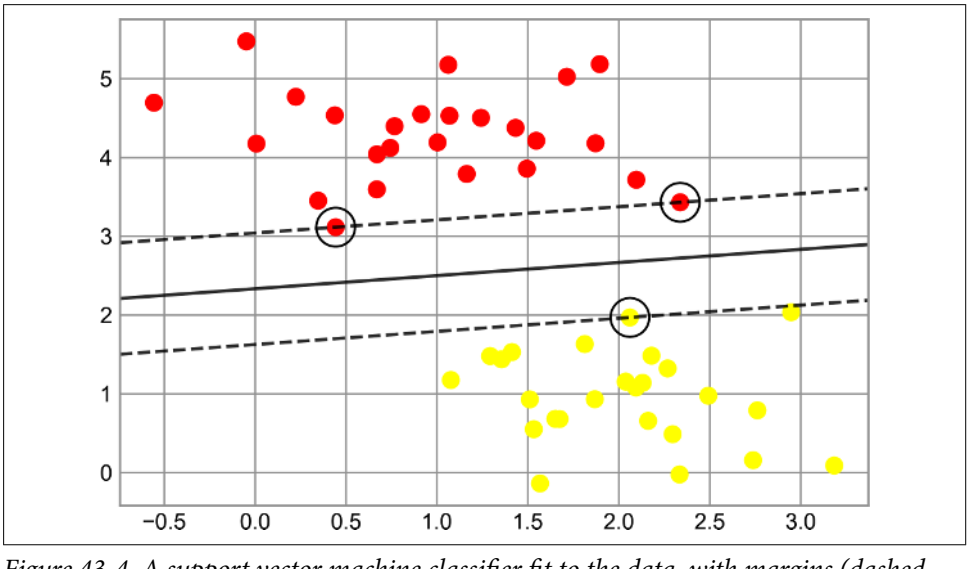

Figure 43-4. A support vector machine classifier fit to the data, with margins (dashed lines) and support vectors (circles) shown

This is the dividing line that maximizes the margin between the two sets of points. Notice that a few of the training points just touch the margin: they are circled in Figure 43-5. These points are the pivotal elements of this fit; they are known as the *support vectors*, and give the algorithm its name. In Scikit-Learn, the identities of these points are stored in the `support_vectors_` attribute of the classifier:

```
In [8]: model.support_vectors_
Out[8]: array([[0.44359863, 3.11530945],
               [2.33812285, 3.43116792],
               [2.06156753, 1.96918596]])
```

A key to this classifier's success is that for the fit, only the positions of the support vectors matter; any points further from the margin that are on the correct side do not modify the fit. Technically, this is because these points do not contribute to the loss function used to fit the model, so their position and number do not matter so long as they do not cross the margin.

We can see this, for example, if we plot the model learned from the first 60 points and first 120 points of this dataset (Figure 43-5).

```
In [9]: def plot_svm(N=10, ax=None):
            X, y = make_blobs(n_samples=200, centers=2,
                              random_state=0, cluster_std=0.60)
            X = X[:N]
            y = y[:N]
            model = SVC(kernel='linear', C=1E10)
            model.fit(X, y)

            ax = ax or plt.gca()
            ax.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
            ax.set_xlim(-1, 4)
            ax.set_ylim(-1, 6)
            plot_svc_decision_function(model, ax)

        fig, ax = plt.subplots(1, 2, figsize=(16, 6))
        fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)
        for axi, N in zip(ax, [60, 120]):
            plot_svm(N, axi)
            axi.set_title('N = {0}'.format(N))
```

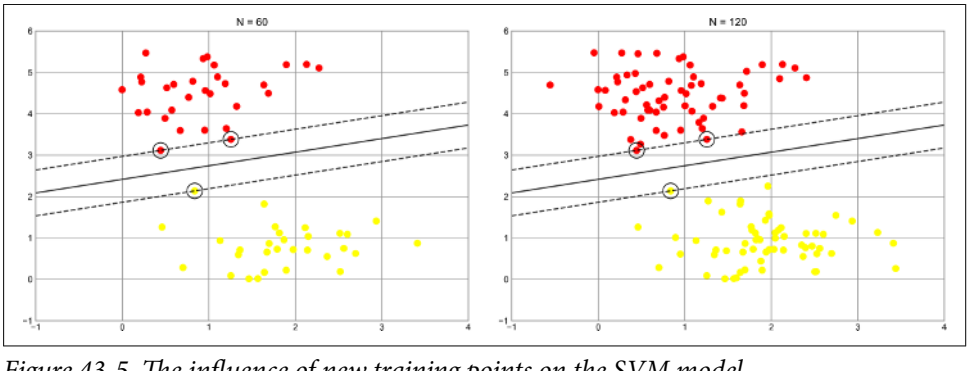

Figure 43-5. The influence of new training points on the SVM model

In the left panel, we see the model and the support vectors for 60 training points. In the right panel, we have doubled the number of training points, but the model has not changed: the three support vectors in the left panel are the same as the support vectors in the right panel. This insensitivity to the exact behavior of distant points is one of the strengths of the SVM model.

If you are running this notebook live, you can use IPython's interactive widgets to view this feature of the SVM model interactively:

```
In [10]: from ipywidgets import interact, fixed
         interact(plot_svm, N=(10, 200), ax=fixed(None));
Out[10]: interactive(children=(IntSlider(value=10, description='N', max=200, min=10),
          > Output()), _dom_classes=('widget-...
```

Beyond Linear Boundaries: Kernel SVM

Where SVM can become quite powerful is when it is combined with *kernels*. We have seen a version of kernels before, in the basis function regressions of Chapter 42. There we projected our data into a higher-dimensional space defined by polynomials and Gaussian basis functions, and thereby were able to fit for nonlinear relationships with a linear classifier.

In SVM models, we can use a version of the same idea. To motivate the need for kernels, let's look at some data that is not linearly separable (Figure 43-6).

```
In [11]: from sklearn.datasets import make_circles
         X, y = make_circles(100, factor=.1, noise=.1)

         clf = SVC(kernel='linear').fit(X, y)

         plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
         plot_svc_decision_function(clf, plot_support=False);
```

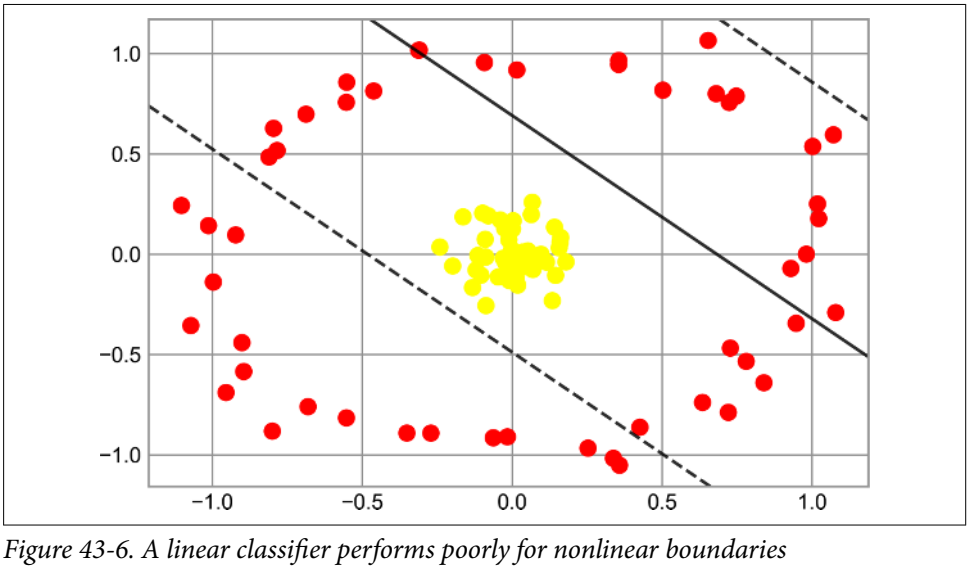

Figure 43-6. A linear classifier performs poorly for nonlinear boundaries

It is clear that no linear discrimination will *ever* be able to separate this data. But we can draw a lesson from the basis function regressions in Chapter 42, and think about

how we might project the data into a higher dimension such that a linear separator *would* be sufficient. For example, one simple projection we could use would be to compute a *radial basis function* (RBF) centered on the middle clump:

```
In [12]: r = np.exp(-(X ** 2).sum(1))
```

We can visualize this extra data dimension using a three-dimensional plot, as seen in Figure 43-7.

```
In [13]: from mpl_toolkits import mplot3d

         ax = plt.subplot(projection='3d')
         ax.scatter3D(X[:, 0], X[:, 1], r, c=y, s=50, cmap='autumn')
         ax.view_init(elev=20, azim=30)
         ax.set_xlabel('x')
         ax.set_ylabel('y')
         ax.set_zlabel('r');
```

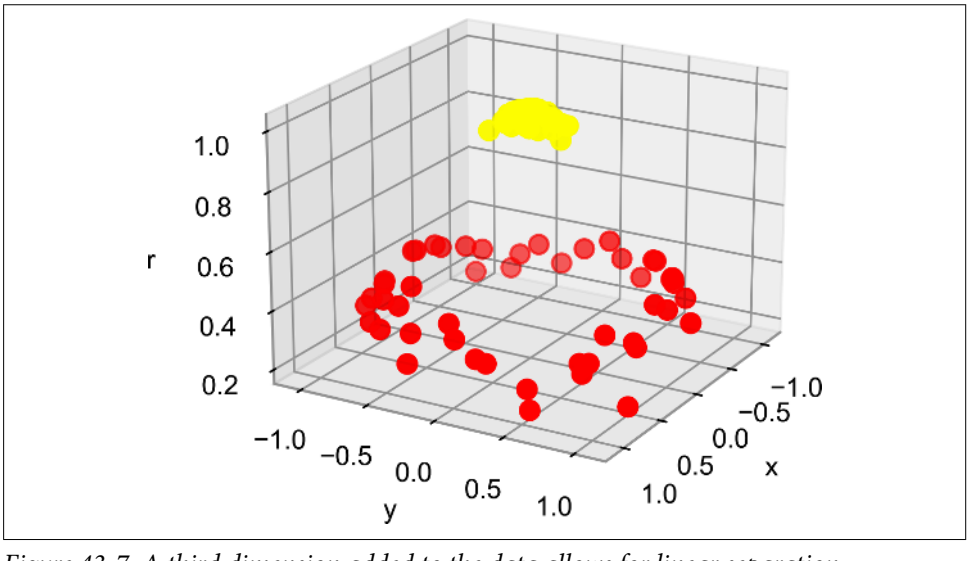

Figure 43-7. A third dimension added to the data allows for linear separation

We can see that with this additional dimension, the data becomes trivially linearly separable, by drawing a separating plane at, say, r=0.7.

In this case we had to choose and carefully tune our projection: if we had not centered our radial basis function in the right location, we would not have seen such clean, linearly separable results. In general, the need to make such a choice is a problem: we would like to somehow automatically find the best basis functions to use.

One strategy to this end is to compute a basis function centered at *every* point in the dataset, and let the SVM algorithm sift through the results. This type of basis function

transformation is known as a *kernel transformation*, as it is based on a similarity relationship (or kernel) between each pair of points.

A potential problem with this strategy—projecting N points into N dimensions—is that it might become very computationally intensive as N grows large. However, because of a neat little procedure known as the *kernel trick*, a fit on kernel-transformed data can be done implicitly—that is, without ever building the full N-dimensional representation of the kernel projection. This kernel trick is built into the SVM, and is one of the reasons the method is so powerful.

In Scikit-Learn, we can apply kernelized SVM simply by changing our linear kernel to an RBF kernel, using the `kernel` model hyperparameter:

```
In [14]: clf = SVC(kernel='rbf', C=1E6)
         clf.fit(X, y)
Out[14]: SVC(C=1000000.0)
```

Let's use our previously defined function to visualize the fit and identify the support vectors (Figure 43-8).

```
In [15]: plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
         plot_svc_decision_function(clf)
         plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
                     s=300, lw=1, facecolors='none');
```

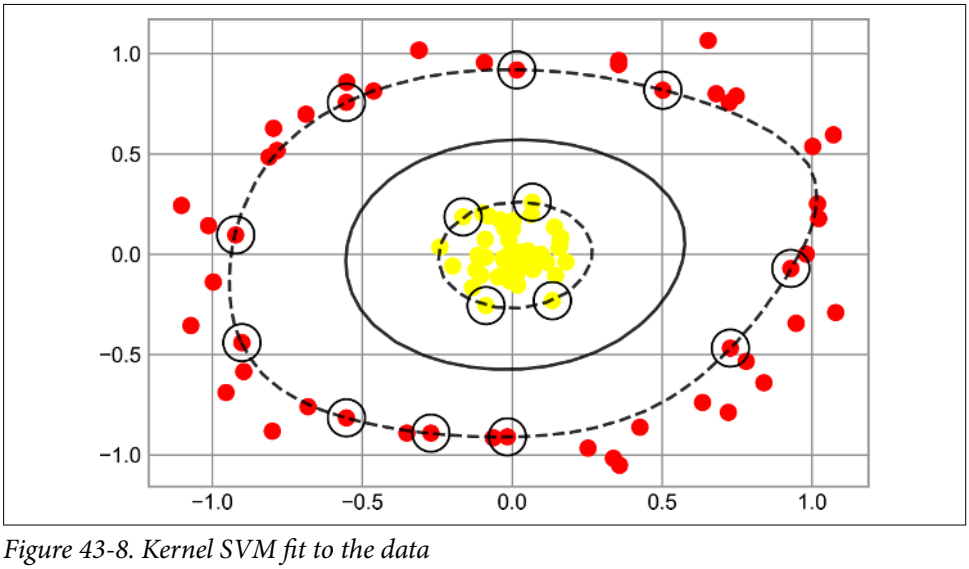

Figure 43-8. Kernel SVM fit to the data

Using this kernelized support vector machine, we learn a suitable nonlinear decision boundary. This kernel transformation strategy is used often in machine learning to turn fast linear methods into fast nonlinear methods, especially for models in which the kernel trick can be used.

Tuning the SVM: Softening Margins

Our discussion thus far has centered around very clean datasets, in which a perfect decision boundary exists. But what if your data has some amount of overlap? For example, you may have data like this (see Figure 43-9).

```
In [16]: X, y = make_blobs(n_samples=100, centers=2,
                           random_state=0, cluster_std=1.2)
         plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');
```

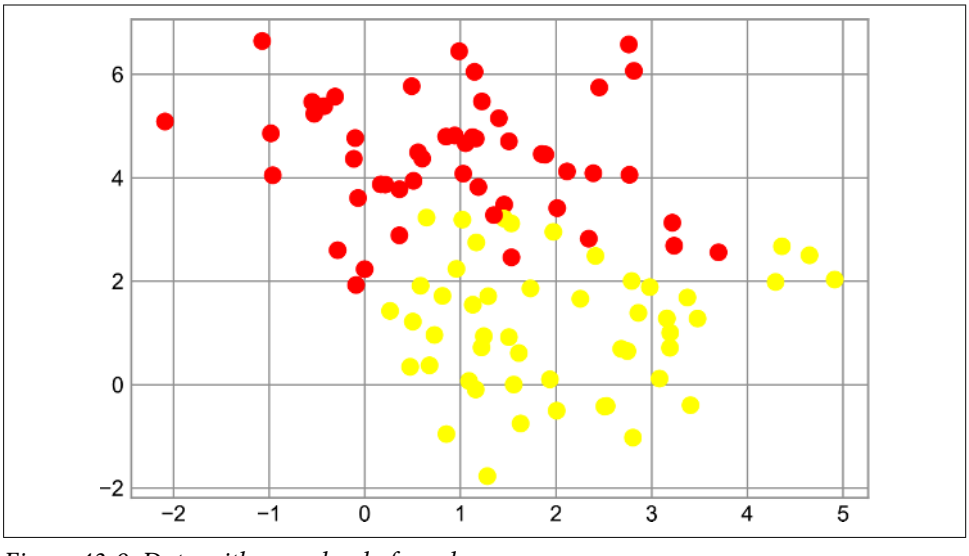

Figure 43-9. Data with some level of overlap

To handle this case, the SVM implementation has a bit of a fudge factor that "softens" the margin: that is, it allows some of the points to creep into the margin if that allows a better fit. The hardness of the margin is controlled by a tuning parameter, most often known as `C`. For a very large `C`, the margin is hard, and points cannot lie in it. For a smaller `C`, the margin is softer and can grow to encompass some points.

The plot shown in Figure 43-10 gives a visual picture of how a changing `C` affects the final fit via the softening of the margin:

```
In [17]: X, y = make_blobs(n_samples=100, centers=2,
                           random_state=0, cluster_std=0.8)

         fig, ax = plt.subplots(1, 2, figsize=(16, 6))
         fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

         for axi, C in zip(ax, [10.0, 0.1]):
             model = SVC(kernel='linear', C=C).fit(X, y)
             axi.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
             plot_svc_decision_function(model, axi)
```

```
            axi.scatter(model.support_vectors_[:, 0],
                        model.support_vectors_[:, 1],
                        s=300, lw=1, facecolors='none');
            axi.set_title('C = {0:.1f}'.format(C), size=14)
```

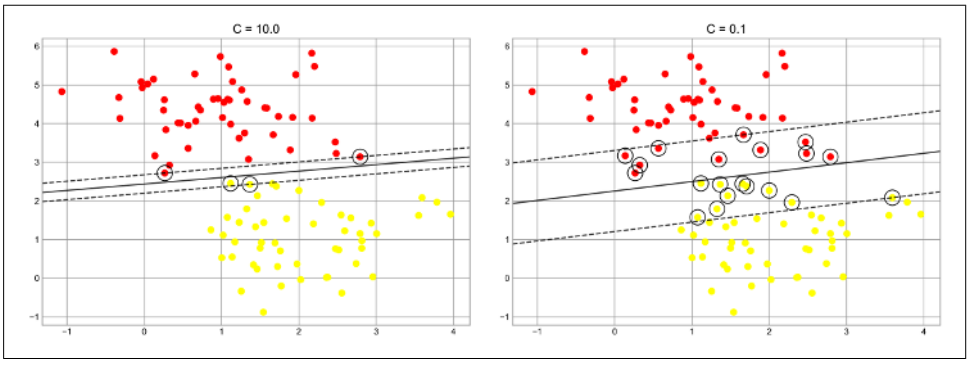

Figure 43-10. The effect of the C parameter on the support vector fit

The optimal value of C will depend on your dataset, and you should tune this parameter using cross-validation or a similar procedure (refer back to Chapter 39).

Example: Face Recognition

As an example of support vector machines in action, let's take a look at the facial recognition problem. We will use the Labeled Faces in the Wild dataset, which consists of several thousand collated photos of various public figures. A fetcher for the dataset is built into Scikit-Learn:

```
In [18]: from sklearn.datasets import fetch_lfw_people
         faces = fetch_lfw_people(min_faces_per_person=60)
         print(faces.target_names)
         print(faces.images.shape)
Out[18]: ['Ariel Sharon' 'Colin Powell' 'Donald Rumsfeld' 'George W Bush'
          'Gerhard Schroeder' 'Hugo Chavez' 'Junichiro Koizumi' 'Tony Blair']
         (1348, 62, 47)
```

Let's plot a few of these faces to see what we're working with (see Figure 43-11).

```
In [19]: fig, ax = plt.subplots(3, 5, figsize=(8, 6))
         for i, axi in enumerate(ax.flat):
             axi.imshow(faces.images[i], cmap='bone')
             axi.set(xticks=[], yticks=[],
                     xlabel=faces.target_names[faces.target[i]])
```

Figure 43-11. Examples from the Labeled Faces in the Wild dataset

Each image contains 62 × 47, or around 3,000, pixels. We could proceed by simply using each pixel value as a feature, but often it is more effective to use some sort of preprocessor to extract more meaningful features; here we will use principal component analysis (see Chapter 45) to extract 150 fundamental components to feed into our support vector machine classifier. We can do this most straightforwardly by packaging the preprocessor and the classifier into a single pipeline:

```
In [20]: from sklearn.svm import SVC
         from sklearn.decomposition import PCA
         from sklearn.pipeline import make_pipeline

         pca = PCA(n_components=150, whiten=True,
                   svd_solver='randomized', random_state=42)
         svc = SVC(kernel='rbf', class_weight='balanced')
         model = make_pipeline(pca, svc)
```

For the sake of testing our classifier output, we will split the data into a training set and a testing set:

```
In [21]: from sklearn.model_selection import train_test_split
         Xtrain, Xtest, ytrain, ytest = train_test_split(faces.data, faces.target,
                                                         random_state=42)
```

Finally, we can use grid search cross-validation to explore combinations of parameters. Here we will adjust `C` (which controls the margin hardness) and `gamma` (which controls the size of the radial basis function kernel), and determine the best model:

```
In [22]: from sklearn.model_selection import GridSearchCV
         param_grid = {'svc__C': [1, 5, 10, 50],
                       'svc__gamma': [0.0001, 0.0005, 0.001, 0.005]}
         grid = GridSearchCV(model, param_grid)

         %time grid.fit(Xtrain, ytrain)
         print(grid.best_params_)
Out[22]: CPU times: user 1min 19s, sys: 8.56 s, total: 1min 27s
         Wall time: 36.2 s
         {'svc__C': 10, 'svc__gamma': 0.001}
```

The optimal values fall toward the middle of our grid; if they fell at the edges, we would want to expand the grid to make sure we have found the true optimum.

Now with this cross-validated model we can predict the labels for the test data, which the model has not yet seen:

```
In [23]: model = grid.best_estimator_
         yfit = model.predict(Xtest)
```

Let's take a look at a few of the test images along with their predicted values (see Figure 43-12).

```
In [24]: fig, ax = plt.subplots(4, 6)
         for i, axi in enumerate(ax.flat):
             axi.imshow(Xtest[i].reshape(62, 47), cmap='bone')
             axi.set(xticks=[], yticks=[])
             axi.set_ylabel(faces.target_names[yfit[i]].split()[-1],
                            color='black' if yfit[i] == ytest[i] else 'red')
         fig.suptitle('Predicted Names; Incorrect Labels in Red', size=14);
```

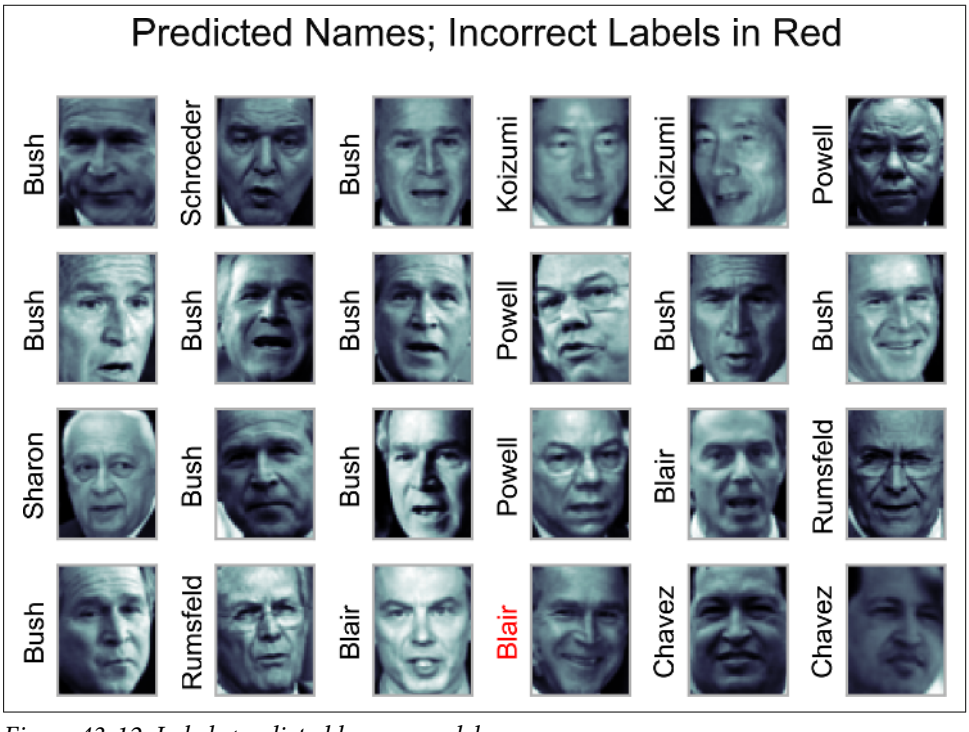

Figure 43-12. Labels predicted by our model

Out of this small sample, our optimal estimator mislabeled only a single face (Bush's face in the bottom row was mislabeled as Blair). We can get a better sense of our estimator's performance using the classification report, which lists recovery statistics label by label:

```
In [25]: from sklearn.metrics import classification_report
         print(classification_report(ytest, yfit,
                                     target_names=faces.target_names))
Out[25]:                    precision    recall  f1-score   support

             Ariel Sharon       0.65      0.73      0.69        15
             Colin Powell       0.80      0.87      0.83        68
          Donald Rumsfeld       0.74      0.84      0.79        31
            George W Bush       0.92      0.83      0.88       126
        Gerhard Schroeder       0.86      0.83      0.84        23
              Hugo Chavez       0.93      0.70      0.80        20
        Junichiro Koizumi       0.92      1.00      0.96        12
               Tony Blair       0.85      0.95      0.90        42

                 accuracy                           0.85       337
                macro avg       0.83      0.84      0.84       337
             weighted avg       0.86      0.85      0.85       337
```

We might also display the confusion matrix between these classes (see Figure 43-13).

```
In [26]: from sklearn.metrics import confusion_matrix
         import seaborn as sns
         mat = confusion_matrix(ytest, yfit)
         sns.heatmap(mat.T, square=True, annot=True, fmt='d',
                     cbar=False, cmap='Blues',
                     xticklabels=faces.target_names,
                     yticklabels=faces.target_names)
         plt.xlabel('true label')
         plt.ylabel('predicted label');
```

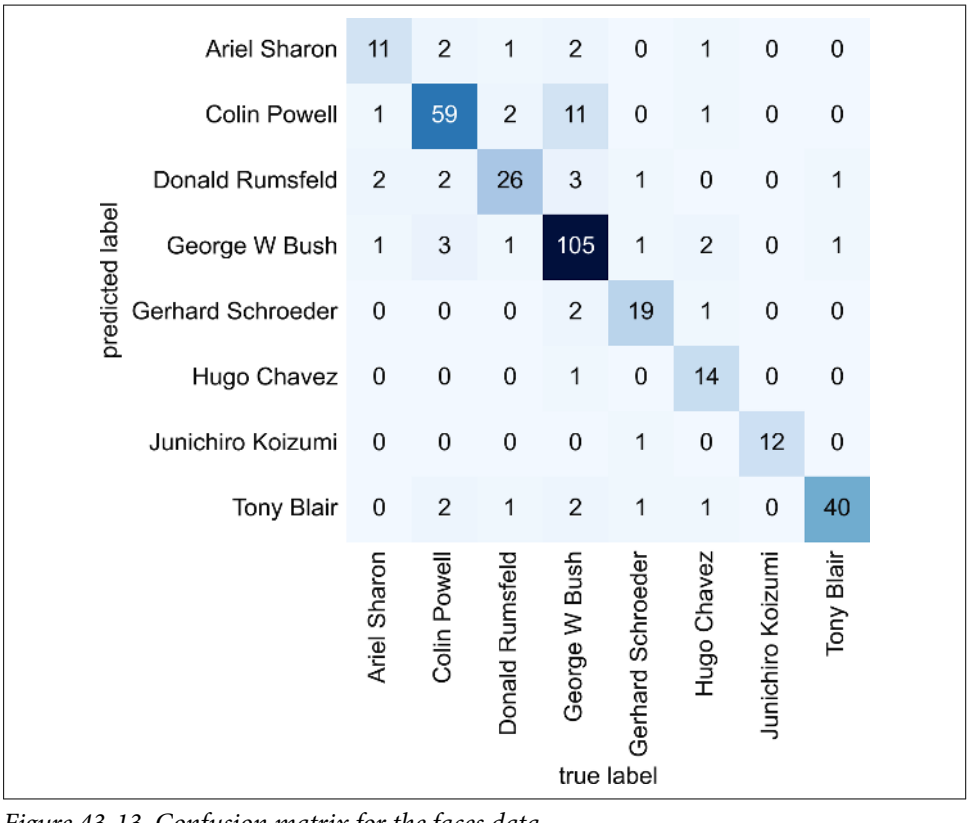

Figure 43-13. Confusion matrix for the faces data

This helps us get a sense of which labels are likely to be confused by the estimator.

For a real-world facial recognition task, in which the photos do not come pre-cropped into nice grids, the only difference in the facial classification scheme is the feature selection: you would need to use a more sophisticated algorithm to find the faces, and extract features that are independent of the pixellation. For this kind of application, one good option is to make use of OpenCV, which, among other things,

includes pretrained implementations of state-of-the-art feature extraction tools for images in general and faces in particular.

Summary

This has been a brief intuitive introduction to the principles behind support vector machines. These models are a powerful classification method for a number of reasons:

- Their dependence on relatively few support vectors means that they are compact and take up very little memory.
- Once the model is trained, the prediction phase is very fast.
- Because they are affected only by points near the margin, they work well with high-dimensional data—even data with more dimensions than samples, which is challenging for other algorithms.
- Their integration with kernel methods makes them very versatile, able to adapt to many types of data.

However, SVMs have several disadvantages as well:

- The scaling with the number of samples N is $\mathcal{O}\left[N^3\right]$ at worst, or $\mathcal{O}\left[N^2\right]$ for efficient implementations. For large numbers of training samples, this computational cost can be prohibitive.
- The results are strongly dependent on a suitable choice for the softening parameter `C`. This must be carefully chosen via cross-validation, which can be expensive as datasets grow in size.
- The results do not have a direct probabilistic interpretation. This can be estimated via an internal cross-validation (see the `probability` parameter of `SVC`), but this extra estimation is costly.

With those traits in mind, I generally only turn to SVMs once other simpler, faster, and less tuning-intensive methods have been shown to be insufficient for my needs. Nevertheless, if you have the CPU cycles to commit to training and cross-validating an SVM on your data, the method can lead to excellent results.