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 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:

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

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; sns.set()
```

Gaussian Naive Bayes

Perhaps the easiest naive Bayes classifier to understand is Gaussian naive Bayes. In this classifier, the assumption is that *data from each label is drawn from a simple Gaussian distribution*. Imagine that you have the following data (Figure 5-38):

```
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');
```

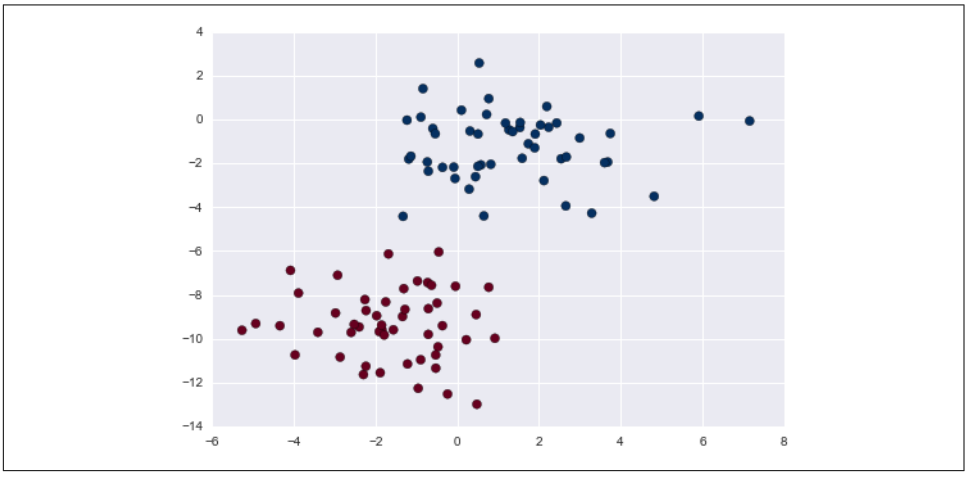

Figure 5-38. Data for Gaussian naive Bayes classification

One extremely fast way to create a simple model is to assume that the data is described by a Gaussian distribution with no covariance between dimensions. We can fit this model by simply finding the mean and standard deviation of the points within each label, which is all you need to define such a distribution. The result of this naive Gaussian assumption is shown in Figure 5-39.

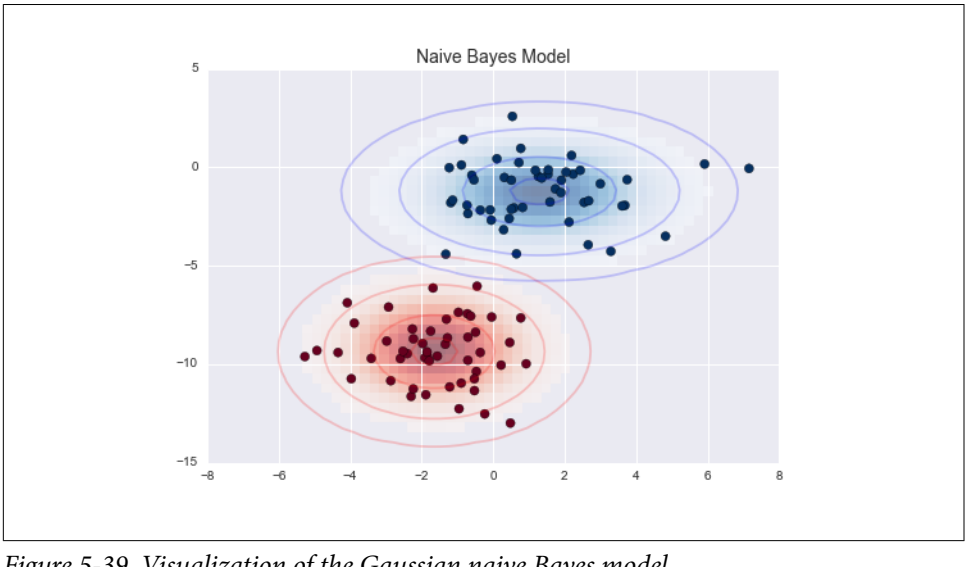

Figure 5-39. Visualization of the Gaussian naive Bayes model

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);
```

Now 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 (Figure 5-40):

```
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);
```

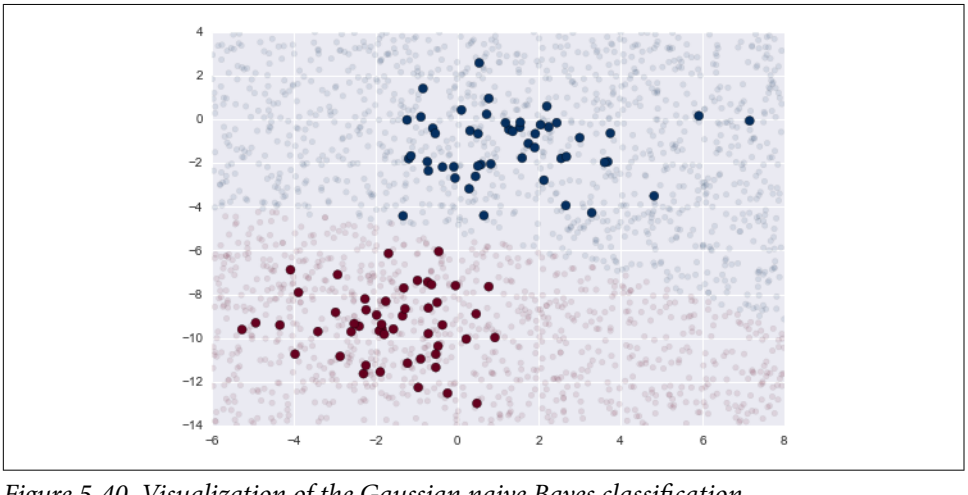

Figure 5-40. Visualization of the Gaussian naive Bayes classification

We see a slightly curved boundary in the classifications—in general, the boundary in Gaussian naive Bayes is quadratic.

A nice piece 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 label, respectively. If you are looking for estimates of uncertainty in your classification, Bayesian approaches like this can be a useful approach.

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 useful 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 the data distribution 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 "Feature Engineering" on page 375; here we will use the sparse word count features from the 20 Newsgroups corpus 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, we will select just a few of these categories, and download the training and testing set:

```
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])
```

```
From: dmcgee@uluhe.soest.hawaii.edu (Don McGee)
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 (discussed in "Feature Engineering" on page 375), 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, here is the confusion matrix between the true and predicted labels for the test data (Figure 5-41):

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

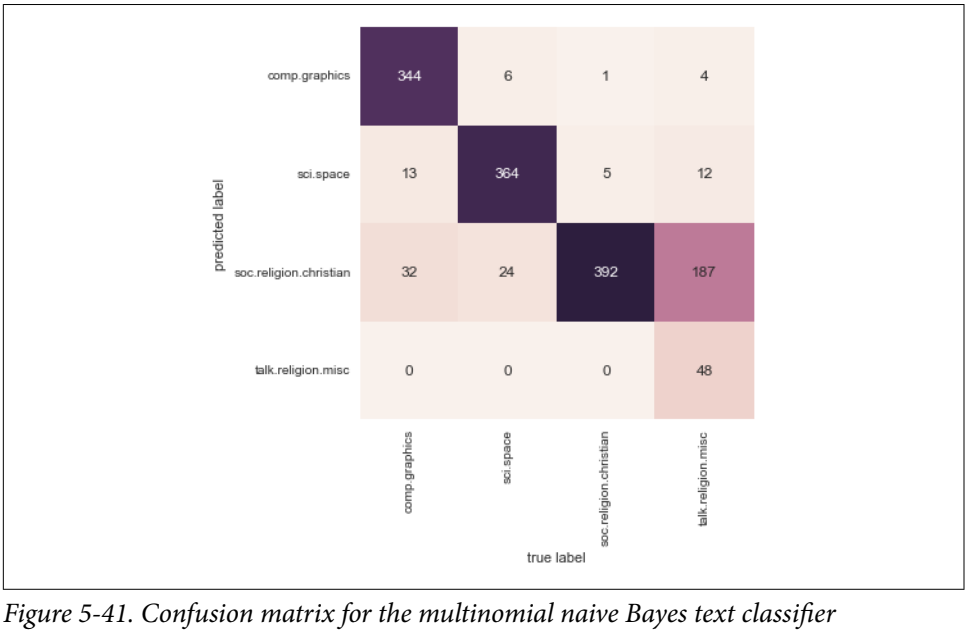

Figure 5-41. Confusion matrix for the multinomial naive Bayes text classifier

Evidently, even this very simple classifier can successfully separate space talk from computer talk, but it gets confused between talk about religion and talk about Christianity. This is perhaps an expected area of confusion!

The very 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 quick 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 islam vs atheism')
Out[15]: 'soc.religion.christian'

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

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 Bayesian classifiers make such stringent assumptions about data, they will generally not perform as well as a more complicated model. That said, they have several advantages:

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

These advantages mean a naive Bayesian 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 one of 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 dimension 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 naive Bayes 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.

In Depth: Linear Regression

Just as naive Bayes (discussed earlier in "In Depth: Naive Bayes Classification" on page 382) is a good starting point for classification tasks, linear regression models are a good starting point for regression tasks. Such models are popular because they can be fit very quickly, and are very interpretable. You are probably familiar with the simplest form of a linear regression model (i.e., fitting a straight line to data), but such models can be extended to model more complicated data behavior.

In this section we will start with a quick intuitive walk-through of the mathematics behind this well-known problem, before moving on to see how linear models can be generalized to account for more complicated patterns in data. We begin with the standard imports:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
       import numpy as np
```

Simple Linear Regression

We will start with the most familiar linear regression, a straight-line fit to data. A straight-line fit is a model of the form $y = ax + b$ where a is commonly known as the *slope*, and b is commonly known as the *intercept*.

Consider the following data, which is scattered about a line with a slope of 2 and an intercept of –5 (Figure 5-42):

```
In[2]: rng = np.random.RandomState(1)
       x = 10 * rng.rand(50)
       y = 2 * x - 5 + rng.randn(50)
       plt.scatter(x, y);
```

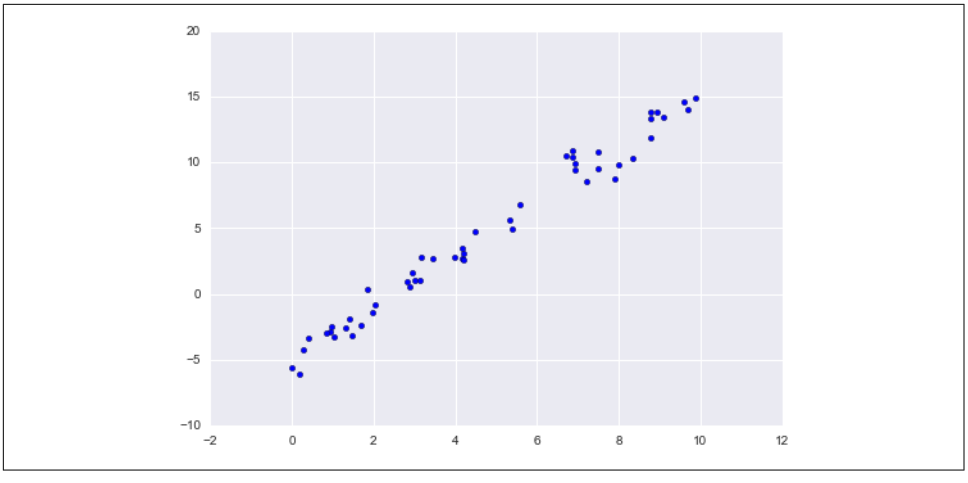

Figure 5-42. Data for linear regression

We can use Scikit-Learn's `LinearRegression` estimator to fit this data and construct the best-fit line (Figure 5-43):

```
In[3]: from sklearn.linear_model import LinearRegression
       model = LinearRegression(fit_intercept=True)

       model.fit(x[:, np.newaxis], y)

       xfit = np.linspace(0, 10, 1000)
       yfit = model.predict(xfit[:, np.newaxis])

       plt.scatter(x, y)
       plt.plot(xfit, yfit);
```

Figure 5-43. A linear regression model

The slope and intercept of the data are contained in the model's fit parameters, which in Scikit-Learn are always marked by a trailing underscore. Here the relevant parameters are `coef_` and `intercept_`:

```
In[4]: print("Model slope:    ", model.coef_[0])
       print("Model intercept:", model.intercept_)
```

```
Model slope:     2.02720881036
Model intercept: -4.99857708555
```

We see that the results are very close to the inputs, as we might hope.

The `LinearRegression` estimator is much more capable than this, however—in addition to simple straight-line fits, it can also handle multidimensional linear models of the form:

$$y = a_0 + a_1 x_1 + a_2 x_2 + \cdots$$

where there are multiple x values. Geometrically, this is akin to fitting a plane to points in three dimensions, or fitting a hyper-plane to points in higher dimensions.

The multidimensional nature of such regressions makes them more difficult to visualize, but we can see one of these fits in action by building some example data, using NumPy's matrix multiplication operator:

```
In[5]: rng = np.random.RandomState(1)
       X = 10 * rng.rand(100, 3)
       y = 0.5 + np.dot(X, [1.5, -2., 1.])

       model.fit(X, y)
       print(model.intercept_)
       print(model.coef_)
```

```
0.5
[ 1.5 -2.   1. ]
```

Here the y data is constructed from three random x values, and the linear regression recovers the coefficients used to construct the data.

In this way, we can use the single `LinearRegression` estimator to fit lines, planes, or hyperplanes to our data. It still appears that this approach would be limited to strictly linear relationships between variables, but it turns out we can relax this as well.

Basis Function Regression

One trick you can use to adapt linear regression to nonlinear relationships between variables is to transform the data according to *basis functions*. We have seen one version of this before, in the `PolynomialRegression` pipeline used in "Hyperparameters

and Model Validation" on page 359 and "Feature Engineering" on page 375. The idea is to take our multidimensional linear model:

$$y = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_3 + \cdots$$

and build the x_1, x_2, x_3, and so on from our single-dimensional input x. That is, we let $x_n = f_n(x)$, where $f_n()$ is some function that transforms our data.

For example, if $f_n(x) = x^n$, our model becomes a polynomial regression:

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots$$

Notice that this is *still a linear model*—the linearity refers to the fact that the coefficients a_n never multiply or divide each other. What we have effectively done is taken our one-dimensional x values and projected them into a higher dimension, so that a linear fit can fit more complicated relationships between x and y.

Polynomial basis functions

This polynomial projection is useful enough that it is built into Scikit-Learn, using the `PolynomialFeatures` transformer:

```
In[6]: from sklearn.preprocessing import PolynomialFeatures
       x = np.array([2, 3, 4])
       poly = PolynomialFeatures(3, include_bias=False)
       poly.fit_transform(x[:, None])
```

```
Out[6]: array([[  2.,   4.,   8.],
               [  3.,   9.,  27.],
               [  4.,  16.,  64.]])
```

We see here that the transformer has converted our one-dimensional array into a three-dimensional array by taking the exponent of each value. This new, higher-dimensional data representation can then be plugged into a linear regression.

As we saw in "Feature Engineering" on page 375, the cleanest way to accomplish this is to use a pipeline. Let's make a 7th-degree polynomial model in this way:

```
In[7]: from sklearn.pipeline import make_pipeline
       poly_model = make_pipeline(PolynomialFeatures(7),
                                  LinearRegression())
```

With this transform in place, we can use the linear model to fit much more complicated relationships between x and y. For example, here is a sine wave with noise (Figure 5-44):

```
In[8]: rng = np.random.RandomState(1)
       x = 10 * rng.rand(50)
       y = np.sin(x) + 0.1 * rng.randn(50)

       poly_model.fit(x[:, np.newaxis], y)
       yfit = poly_model.predict(xfit[:, np.newaxis])

       plt.scatter(x, y)
       plt.plot(xfit, yfit);
```

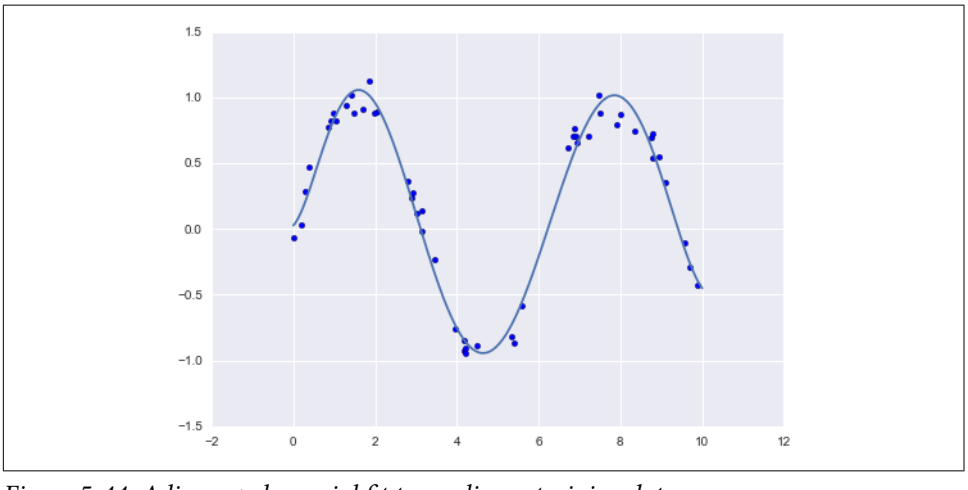

Figure 5-44. A linear polynomial fit to nonlinear training data

Our linear model, through the use of 7th-order polynomial basis functions, can provide an excellent fit to this nonlinear data!

Gaussian basis functions

Of course, other basis functions are possible. For example, one useful pattern is to fit a model that is not a sum of polynomial bases, but a sum of Gaussian bases. The result might look something like Figure 5-45.

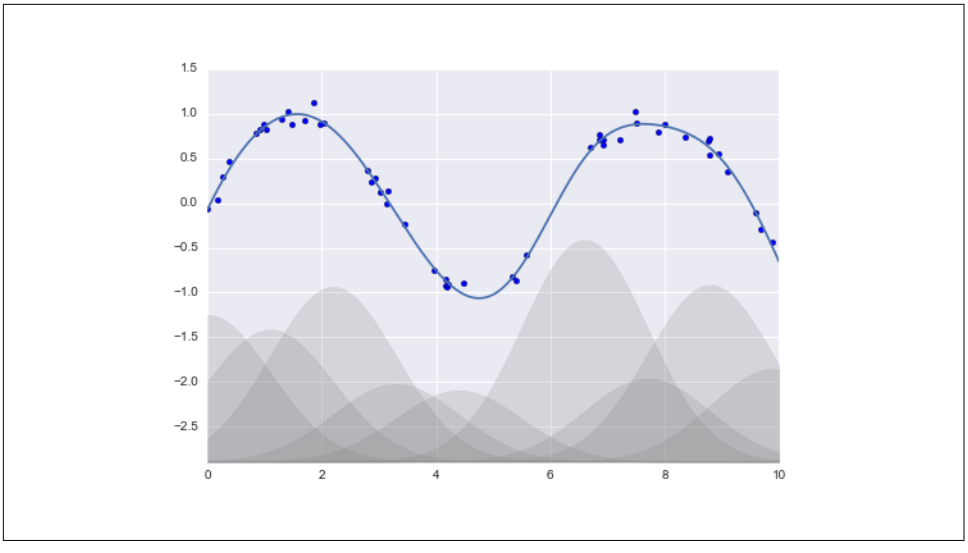

Figure 5-45. A Gaussian basis function fit to nonlinear data

The shaded regions in the plot shown in Figure 5-45 are the scaled basis functions, and when added together they reproduce the smooth curve through the data. These Gaussian basis functions are not built into Scikit-Learn, but we can write a custom transformer that will create them, as shown here and illustrated in Figure 5-46 (Scikit-Learn transformers are implemented as Python classes; reading Scikit-Learn's source is a good way to see how they can be created):

```
In[9]:
from sklearn.base import BaseEstimator, TransformerMixin

class GaussianFeatures(BaseEstimator, TransformerMixin):
    """Uniformly spaced Gaussian features for one-dimensional input"""

    def __init__(self, N, width_factor=2.0):
        self.N = N
        self.width_factor = width_factor

    @staticmethod
    def _gauss_basis(x, y, width, axis=None):
        arg = (x - y) / width
        return np.exp(-0.5 * np.sum(arg ** 2, axis))

    def fit(self, X, y=None):
        # create N centers spread along the data range
        self.centers_ = np.linspace(X.min(), X.max(), self.N)
        self.width_ = self.width_factor * (self.centers_[1] - self.centers_[0])
        return self

    def transform(self, X):
```

```
            return self._gauss_basis(X[:, :, np.newaxis], self.centers_,
                                     self.width_, axis=1)

gauss_model = make_pipeline(GaussianFeatures(20),
                            LinearRegression())
gauss_model.fit(x[:, np.newaxis], y)
yfit = gauss_model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)
plt.plot(xfit, yfit)
plt.xlim(0, 10);
```

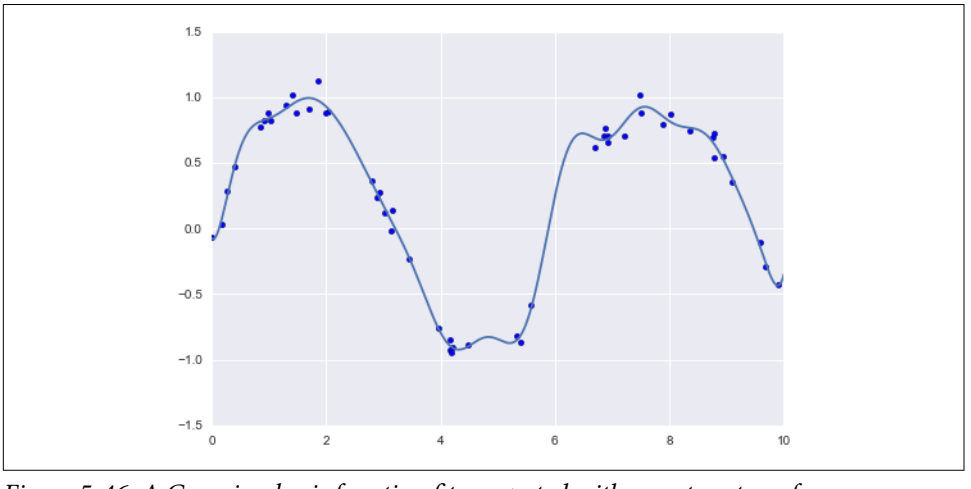

Figure 5-46. A Gaussian basis function fit computed with a custom transformer

We put this example here just to make clear that there is nothing magic about polynomial basis functions: if you have some sort of intuition into the generating process of your data that makes you think one basis or another might be appropriate, you can use them as well.

Regularization

The introduction of basis functions into our linear regression makes the model much more flexible, but it also can very quickly lead to overfitting (refer back to "Hyperparameters and Model Validation" on page 359 for a discussion of this). For example, if we choose too many Gaussian basis functions, we end up with results that don't look so good (Figure 5-47):

```
In[10]: model = make_pipeline(GaussianFeatures(30),
                              LinearRegression())
        model.fit(x[:, np.newaxis], y)

        plt.scatter(x, y)
        plt.plot(xfit, model.predict(xfit[:, np.newaxis]))
```

```
plt.xlim(0, 10)
plt.ylim(-1.5, 1.5);
```

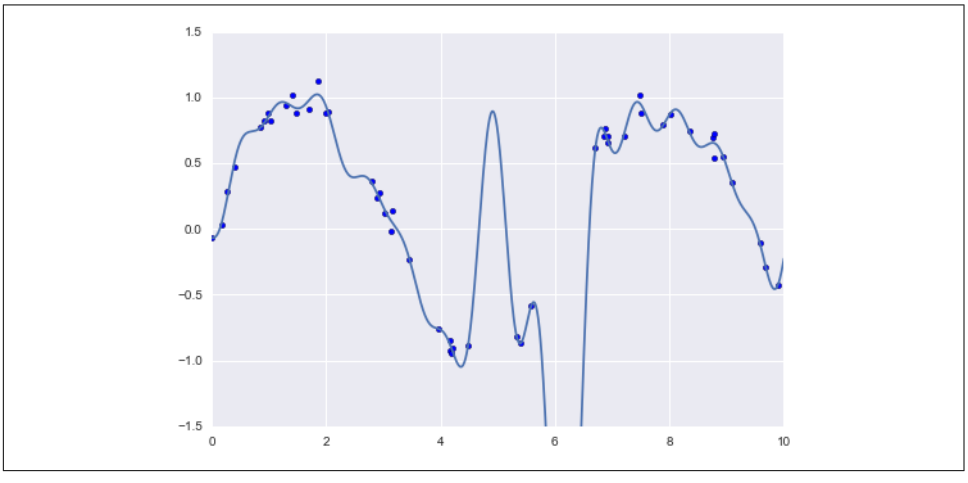

Figure 5-47. An overly complex basis function model that overfits the data

With the data projected to the 30-dimensional basis, the model has far too much flexibility and goes to extreme values between locations where it is constrained by data. We can see the reason for this if we plot the coefficients of the Gaussian bases with respect to their locations (Figure 5-48):

```
In[11]: def basis_plot(model, title=None):
            fig, ax = plt.subplots(2, sharex=True)
            model.fit(x[:, np.newaxis], y)
            ax[0].scatter(x, y)
            ax[0].plot(xfit, model.predict(xfit[:, np.newaxis]))
            ax[0].set(xlabel='x', ylabel='y', ylim=(-1.5, 1.5))

            if title:
                ax[0].set_title(title)

            ax[1].plot(model.steps[0][1].centers_,
                       model.steps[1][1].coef_)
            ax[1].set(xlabel='basis location',
                      ylabel='coefficient',
                      xlim=(0, 10))

        model = make_pipeline(GaussianFeatures(30), LinearRegression())
        basis_plot(model)
```

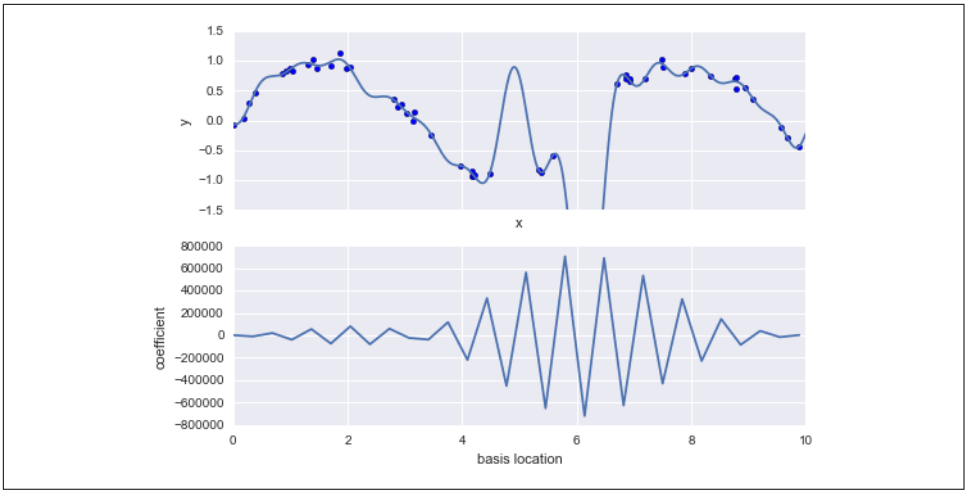

Figure 5-48. The coefficients of the Gaussian bases in the overly complex model

The lower panel in Figure 5-48 shows the amplitude of the basis function at each location. This is typical overfitting behavior when basis functions overlap: the coefficients of adjacent basis functions blow up and cancel each other out. We know that such behavior is problematic, and it would be nice if we could limit such spikes explicitly in the model by penalizing large values of the model parameters. Such a penalty is known as *regularization*, and comes in several forms.

Ridge regression (L_2 regularization)

Perhaps the most common form of regularization is known as *ridge regression* or L_2 *regularization*, sometimes also called *Tikhonov regularization*. This proceeds by penalizing the sum of squares (2-norms) of the model coefficients; in this case, the penalty on the model fit would be:

$$P = \alpha \sum_{n=1}^{N} \theta_n^2$$

where α is a free parameter that controls the strength of the penalty. This type of penalized model is built into Scikit-Learn with the `Ridge` estimator (Figure 5-49):

```
In[12]: from sklearn.linear_model import Ridge
        model = make_pipeline(GaussianFeatures(30), Ridge(alpha=0.1))
        basis_plot(model, title='Ridge Regression')
```

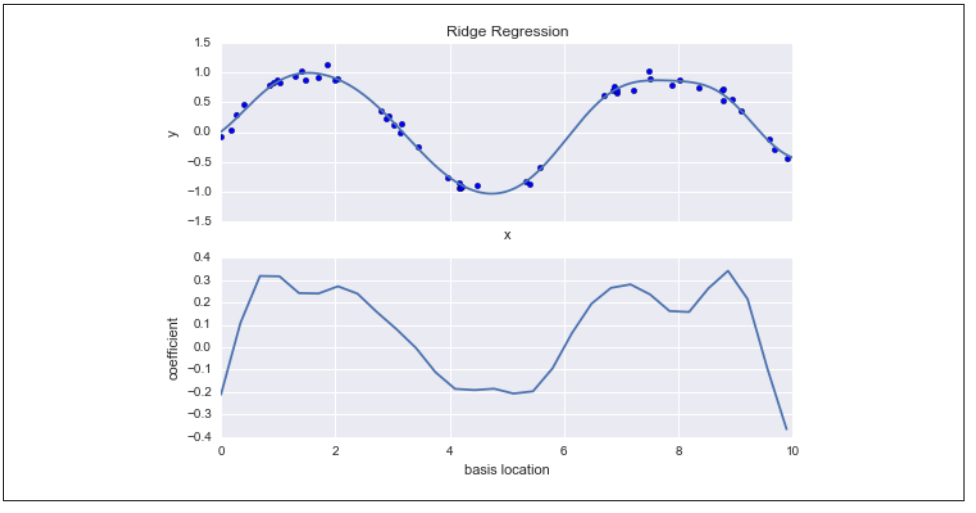

Figure 5-49. Ridge (L_2) regularization applied to the overly complex model (compare to Figure 5-48)

The α parameter is essentially a knob controlling the complexity of the resulting model. In the limit $\alpha \to 0$, we recover the standard linear regression result; in the limit $\alpha \to \infty$, all model responses will be suppressed. One advantage of ridge regression in particular is that it can be computed very efficiently—at hardly more computational cost than the original linear regression model.

Lasso regularization (L_1)

Another very common type of regularization is known as lasso, and involves penalizing the sum of absolute values (1-norms) of regression coefficients:

$$P = \alpha \Sigma_{n=1}^{N} |\theta_n|$$

Though this is conceptually very similar to ridge regression, the results can differ surprisingly: for example, due to geometric reasons lasso regression tends to favor *sparse models* where possible; that is, it preferentially sets model coefficients to exactly zero.

We can see this behavior in duplicating the plot shown in Figure 5-49, but using L1-normalized coefficients (Figure 5-50):

```
In[13]: from sklearn.linear_model import Lasso
        model = make_pipeline(GaussianFeatures(30), Lasso(alpha=0.001))
        basis_plot(model, title='Lasso Regression')
```

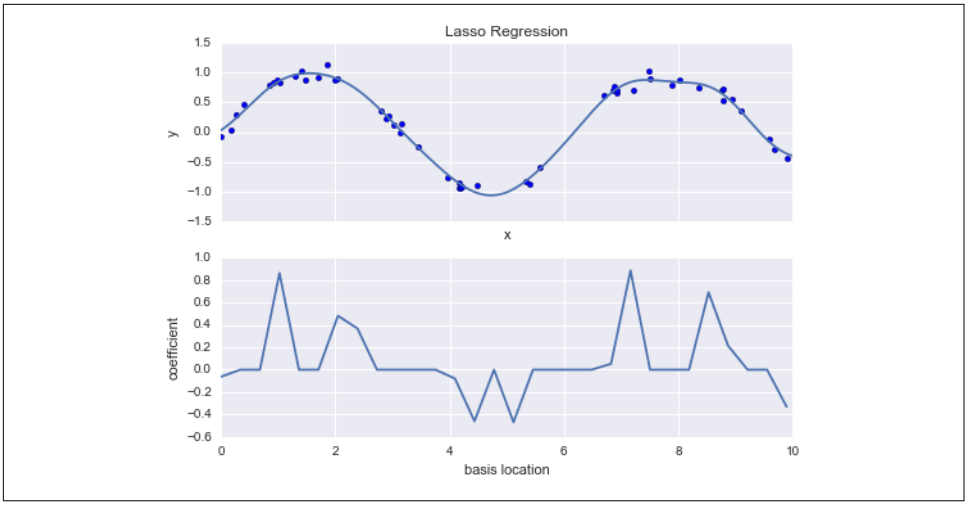

Figure 5-50. Lasso (L_1) regularization applied to the overly complex model (compare to Figure 5-48)

With the lasso regression penalty, the majority of the coefficients are exactly zero, with the functional behavior being modeled by a small subset of the available basis functions. As with ridge regularization, the α parameter tunes the strength of the penalty, and should be determined via, for example, cross-validation (refer back to "Hyperparameters and Model Validation" on page 359 for a discussion of this).

Example: Predicting Bicycle Traffic

As an example, let's take a look at whether we can predict the number of bicycle trips across Seattle's Fremont Bridge based on weather, season, and other factors. We have seen this data already in "Working with Time Series" on page 188.

In this section, we will join the bike data with another dataset, and try to determine the extent to which weather and seasonal factors—temperature, precipitation, and daylight hours—affect the volume of bicycle traffic through this corridor. Fortunately, the NOAA makes available their daily weather station data (I used station ID USW00024233) and we can easily use Pandas to join the two data sources. We will perform a simple linear regression to relate weather and other information to bicycle counts, in order to estimate how a change in any one of these parameters affects the number of riders on a given day.

In particular, this is an example of how the tools of Scikit-Learn can be used in a statistical modeling framework, in which the parameters of the model are assumed to have interpretable meaning. As discussed previously, this is not a standard approach within machine learning, but such interpretation is possible for some models.

Let's start by loading the two datasets, indexing by date:

```
In[14]:
import pandas as pd
counts = pd.read_csv('fremont_hourly.csv', index_col='Date', parse_dates=True)
weather = pd.read_csv('599021.csv', index_col='DATE', parse_dates=True)
```

Next we will compute the total daily bicycle traffic, and put this in its own `DataFrame`:

```
In[15]: daily = counts.resample('d', how='sum')
        daily['Total'] = daily.sum(axis=1)
        daily = daily[['Total']] # remove other columns
```

We saw previously that the patterns of use generally vary from day to day; let's account for this in our data by adding binary columns that indicate the day of the week:

```
In[16]: days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
        for i in range(7):
            daily[days[i]] = (daily.index.dayofweek == i).astype(float)
```

Similarly, we might expect riders to behave differently on holidays; let's add an indicator of this as well:

```
In[17]: from pandas.tseries.holiday import USFederalHolidayCalendar
        cal = USFederalHolidayCalendar()
        holidays = cal.holidays('2012', '2016')
        daily = daily.join(pd.Series(1, index=holidays, name='holiday'))
        daily['holiday'].fillna(0, inplace=True)
```

We also might suspect that the hours of daylight would affect how many people ride; let's use the standard astronomical calculation to add this information (Figure 5-51):

```
In[18]: def hours_of_daylight(date, axis=23.44, latitude=47.61):
            """Compute the hours of daylight for the given date"""
            days = (date - pd.datetime(2000, 12, 21)).days
            m = (1. - np.tan(np.radians(latitude))
                 * np.tan(np.radians(axis) * np.cos(days * 2 * np.pi / 365.25)))
            return 24. * np.degrees(np.arccos(1 - np.clip(m, 0, 2))) / 180.

        daily['daylight_hrs'] = list(map(hours_of_daylight, daily.index))
        daily[['daylight_hrs']].plot();
```

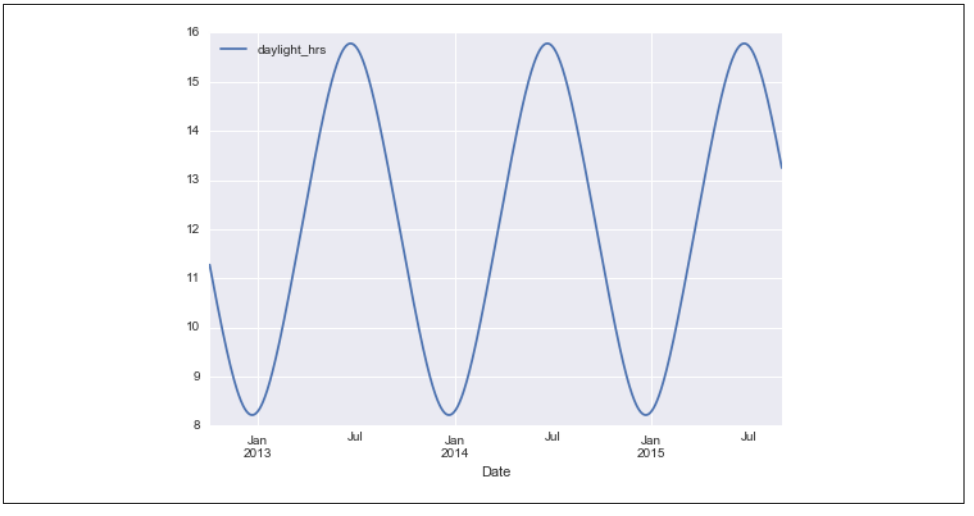

Figure 5-51. Visualization of hours of daylight in Seattle

We can also add the average temperature and total precipitation to the data. In addition to the inches of precipitation, let's add a flag that indicates whether a day is dry (has zero precipitation):

```
In[19]: # temperatures are in 1/10 deg C; convert to C
        weather['TMIN'] /= 10
        weather['TMAX'] /= 10
        weather['Temp (C)'] = 0.5 * (weather['TMIN'] + weather['TMAX'])

        # precip is in 1/10 mm; convert to inches
        weather['PRCP'] /= 254
        weather['dry day'] = (weather['PRCP'] == 0).astype(int)

        daily = daily.join(weather[['PRCP', 'Temp (C)', 'dry day']])
```

Finally, let's add a counter that increases from day 1, and measures how many years have passed. This will let us measure any observed annual increase or decrease in daily crossings:

```
In[20]: daily['annual'] = (daily.index - daily.index[0]).days / 365.
```

Now our data is in order, and we can take a look at it:

```
In[21]: daily.head()
```

```
Out[21]:
            Total  Mon  Tue  Wed  Thu  Fri  Sat  Sun  holiday  daylight_hrs  \\
Date
2012-10-03   3521    0    0    1    0    0    0    0        0     11.277359
2012-10-04   3475    0    0    0    1    0    0    0        0     11.219142
2012-10-05   3148    0    0    0    0    1    0    0        0     11.161038
2012-10-06   2006    0    0    0    0    0    1    0        0     11.103056
```

```
2012-10-07   2142    0    0    0    0    0    0    1        0     11.045208
```

```
            PRCP  Temp (C)  dry day    annual
Date
2012-10-03     0     13.35        1  0.000000
2012-10-04     0     13.60        1  0.002740
2012-10-05     0     15.30        1  0.005479
2012-10-06     0     15.85        1  0.008219
2012-10-07     0     15.85        1  0.010959
```

With this in place, we can choose the columns to use, and fit a linear regression model to our data. We will set `fit_intercept = False`, because the daily flags essentially operate as their own day-specific intercepts:

```
In[22]:
column_names = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun', 'holiday',
                'daylight_hrs', 'PRCP', 'dry day', 'Temp (C)', 'annual']
X = daily[column_names]
y = daily['Total']

model = LinearRegression(fit_intercept=False)
model.fit(X, y)
daily['predicted'] = model.predict(X)
```

Finally, we can compare the total and predicted bicycle traffic visually (Figure 5-52):

```
In[23]: daily[['Total', 'predicted']].plot(alpha=0.5);
```

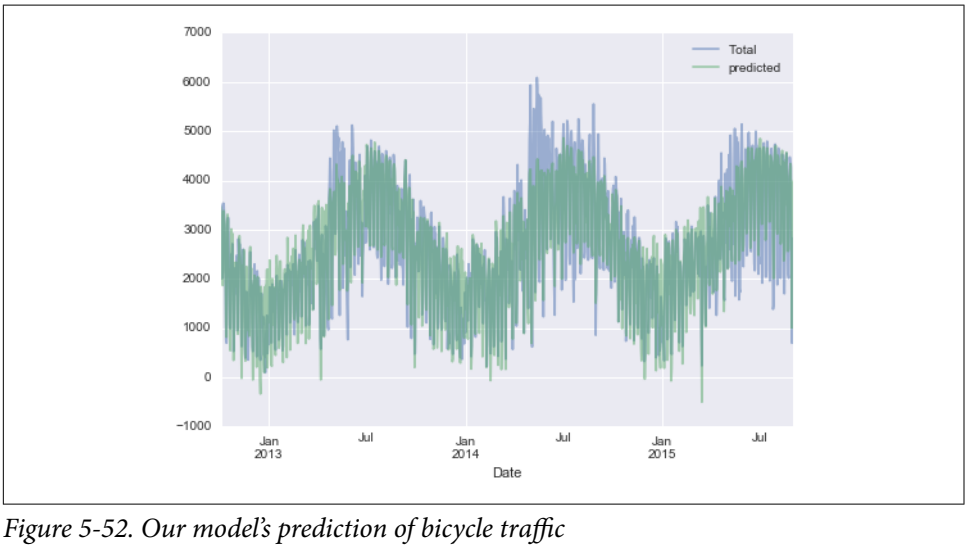

Figure 5-52. Our model's prediction of bicycle traffic

It is evident that we have missed some key features, especially during the summer time. Either our features are not complete (i.e., people decide whether to ride to work based on more than just these) or there are some nonlinear relationships that we have

failed to take into account (e.g., perhaps people ride less at both high and low temperatures). Nevertheless, our rough approximation is enough to give us some insights, and we can take a look at the coefficients of the linear model to estimate how much each feature contributes to the daily bicycle count:

```
In[24]: params = pd.Series(model.coef_, index=X.columns)
        params

Out[24]: Mon              503.797330
         Tue              612.088879
         Wed              591.611292
         Thu              481.250377
         Fri              176.838999
         Sat            -1104.321406
         Sun            -1134.610322
         holiday        -1187.212688
         daylight_hrs     128.873251
         PRCP            -665.185105
         dry day          546.185613
         Temp (C)          65.194390
         annual            27.865349
         dtype: float64
```

These numbers are difficult to interpret without some measure of their uncertainty. We can compute these uncertainties quickly using bootstrap resamplings of the data:

```
In[25]: from sklearn.utils import resample
        np.random.seed(1)
        err = np.std([model.fit(*resample(X, y)).coef_
                      for i in range(1000)], 0)
```

With these errors estimated, let's again look at the results:

```
In[26]: print(pd.DataFrame({'effect': params.round(0),
                            'error': err.round(0)}))

              effect  error
Mon              504     85
Tue              612     82
Wed              592     82
Thu              481     85
Fri              177     81
Sat            -1104     79
Sun            -1135     82
holiday        -1187    164
daylight_hrs     129      9
PRCP            -665     62
dry day          546     33
Temp (C)          65      4
annual            28     18
```

We first see that there is a relatively stable trend in the weekly baseline: there are many more riders on weekdays than on weekends and holidays. We see that for each

additional hour of daylight, 129 ± 9 more people choose to ride; a temperature increase of one degree Celsius encourages 65 ± 4 people to grab their bicycle; a dry day means an average of 546 ± 33 more riders; and each inch of precipitation means 665 ± 62 more people leave their bike at home. Once all these effects are accounted for, we see a modest increase of 28 ± 18 new daily riders each year.

Our model is almost certainly missing some relevant information. For example, nonlinear effects (such as effects of precipitation *and* cold temperature) and nonlinear trends within each variable (such as disinclination to ride at very cold and very hot temperatures) cannot be accounted for in this model. Additionally, we have thrown away some of the finer-grained information (such as the difference between a rainy morning and a rainy afternoon), and we have ignored correlations between days (such as the possible effect of a rainy Tuesday on Wednesday's numbers, or the effect of an unexpected sunny day after a streak of rainy days). These are all potentially interesting effects, and you now have the tools to begin exploring them if you wish!

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 section, we will develop the intuition behind support vector machines 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
       from scipy import stats

       # use Seaborn plotting defaults
       import seaborn as sns; sns.set()
```

Motivating Support Vector Machines

As part of our discussion of Bayesian classification (see "In Depth: Naive Bayes Classification" on page 382), we learned a simple model describing the distribution of each underlying class, and used these generative models to probabilistically determine labels for new points. That was an example of *generative classification*; here we will consider instead *discriminative classification*: rather than modeling each class, we 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 (Figure 5-53):

```
In[2]: from sklearn.datasets.samples_generator 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');
```

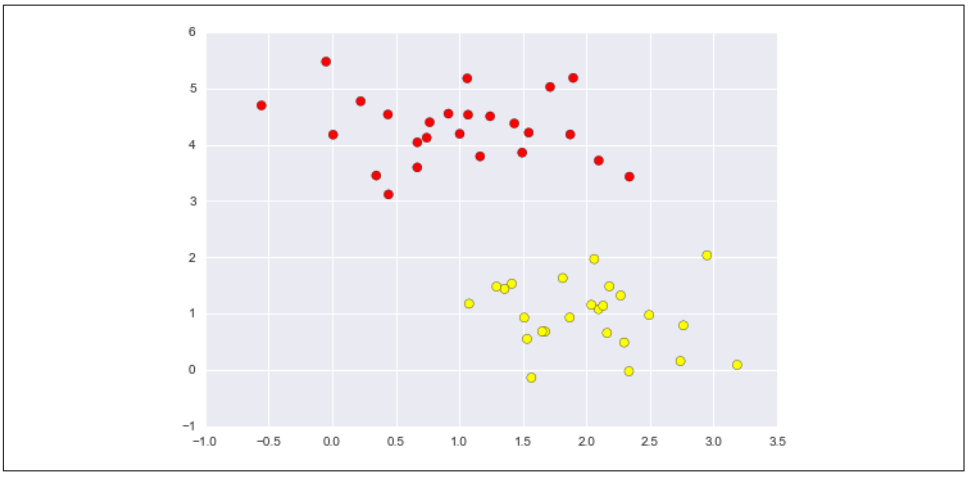

Figure 5-53. Simple data for classification

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 them as follows (Figure 5-54):

```
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);
```

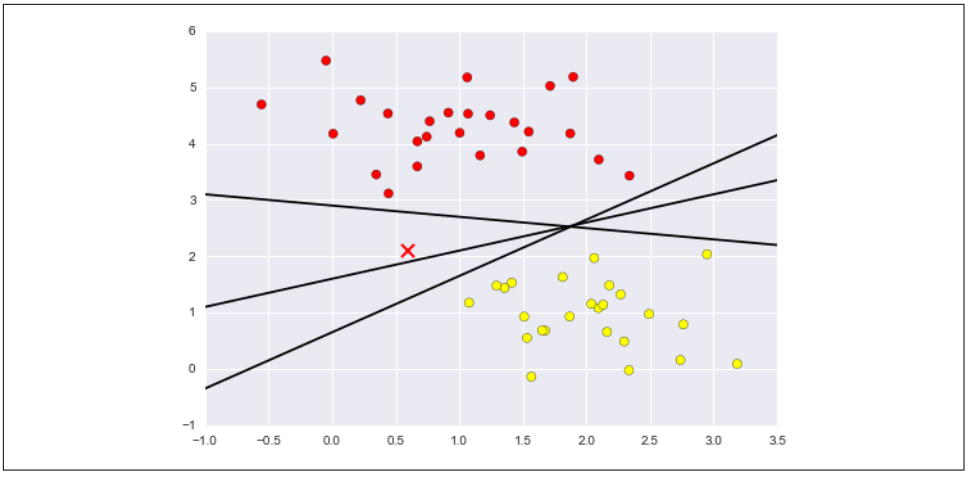

Figure 5-54. Three perfect linear discriminative classifiers for our data

These are three *very* different separators that, nevertheless, perfectly discriminate between these samples. Depending on which you choose, a new data point (e.g., the one marked by the "X" in Figure 5-54) will be assigned a different label! Evidently our simple intuition of "drawing a line between classes" is not enough, and we need to think a bit deeper.

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 5-55):

```
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='#AAAAAA',
                     alpha=0.4)

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

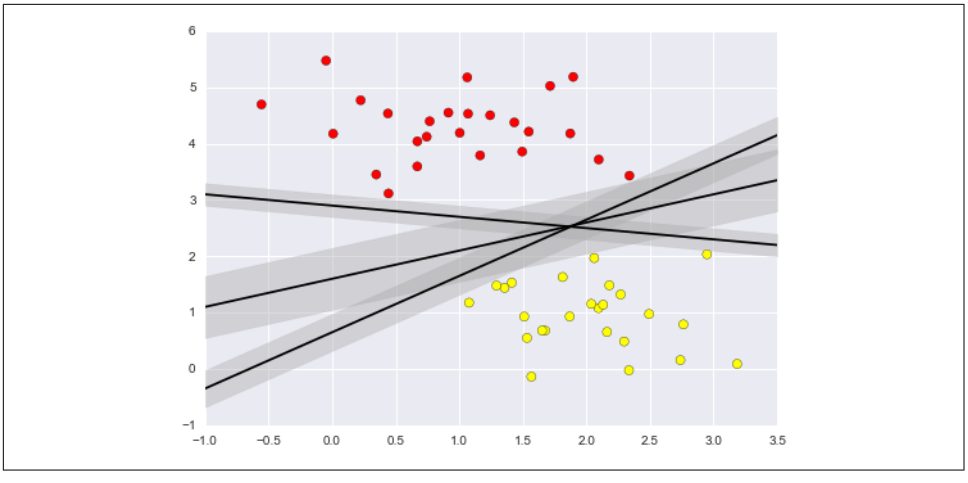

Figure 5-55. Visualization of "margins" within discriminative classifiers

In support vector machines, the line that maximizes this margin is the one we will choose as the optimal model. Support vector machines are an example of such a *maximum margin* estimator.

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 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, cache_size=200, class_weight=None, coef0=0.0,
          decision_function_shape=None, degree=3, gamma='auto', kernel='linear',
          max_iter=-1, probability=False, random_state=None, shrinking=True,
          tol=0.001, verbose=False)
```

To better visualize what's happening here, let's create a quick convenience function that will plot SVM decision boundaries for us (Figure 5-56):

```
In[6]: def plot_svc_decision_function(model, ax=None, plot_support=True):
           """Plot the decision function for a two-dimensional 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, 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);
```

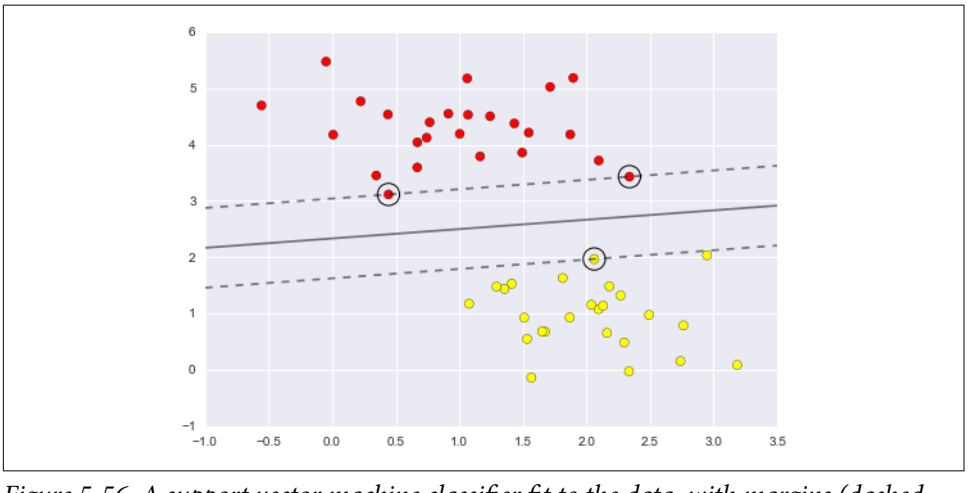

Figure 5-56. A support vector machine classifier fit to the data, with margins (dashed lines) and support vectors (circles) shown

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 indicated by the black circles in Figure 5-56. These points are the pivotal elements of this fit, and are known as the *support vectors*, and give the algorithm its name. In Scikit-Learn, the identity of these points is 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 position of the support vectors matters; 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 5-57):

```
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))
```

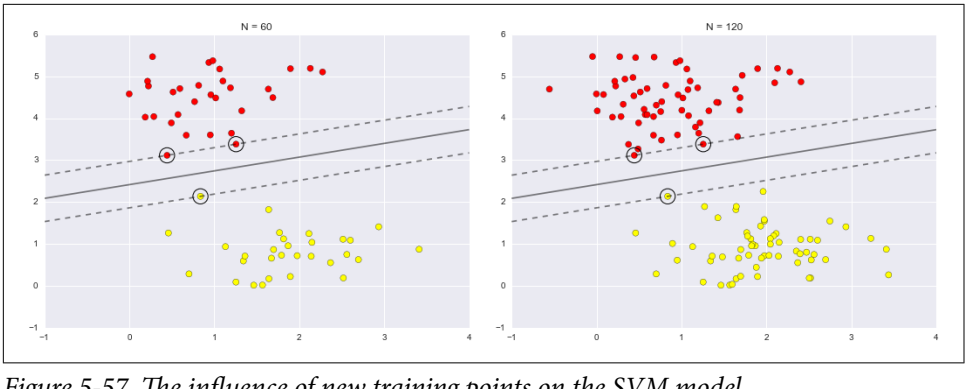

Figure 5-57. The influence of new training points on the SVM model

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 from the left panel are still the support vectors from 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 (Figure 5-58):

```
In[10]: from ipywidgets import interact, fixed
        interact(plot_svm, N=[10, 200], ax=fixed(None));
```

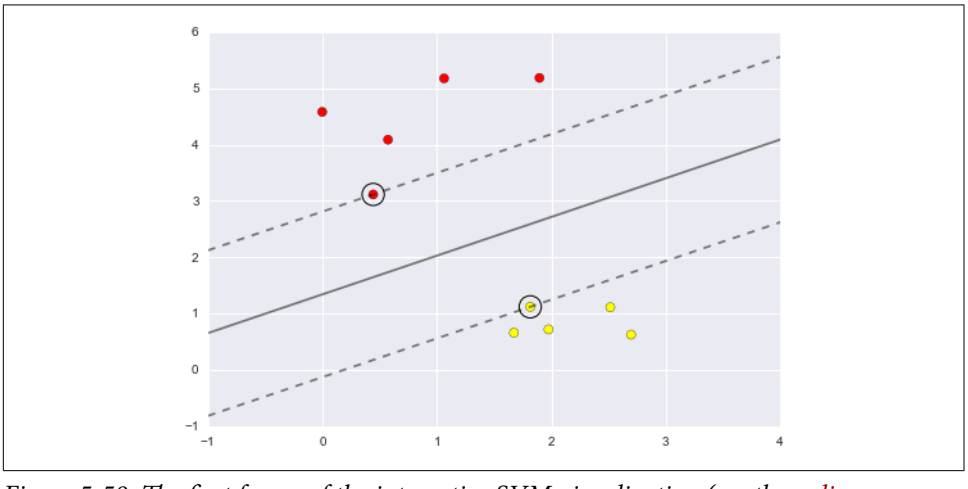

Figure 5-58. The first frame of the interactive SVM visualization (see the online appendix for the full version)

Beyond linear boundaries: Kernel SVM

Where SVM becomes extremely powerful is when it is combined with *kernels*. We have seen a version of kernels before, in the basis function regressions of "In Depth: Linear Regression" on page 390. There we projected our data into 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 5-59):

```
In[11]: from sklearn.datasets.samples_generator 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);
```

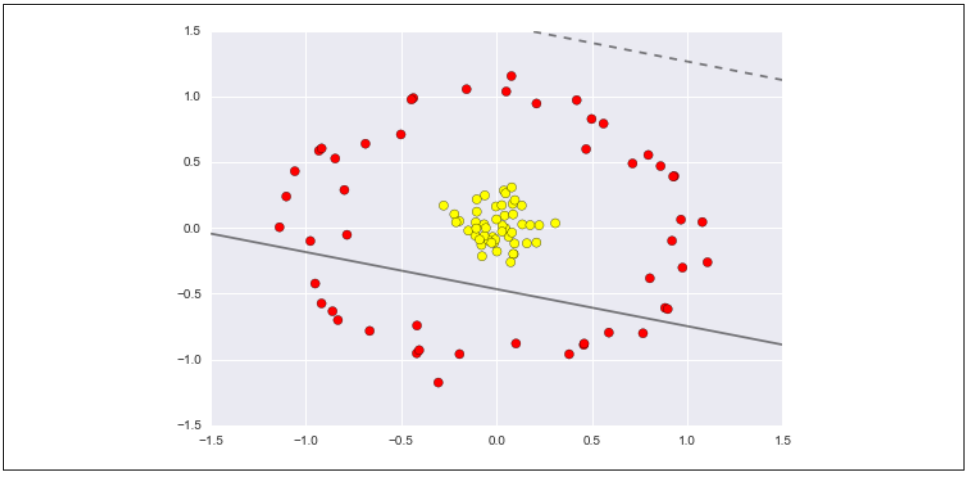

Figure 5-59. A linear classifier performs poorly for nonlinear boundaries

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 "In Depth: Linear Regression" on page 390, 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* 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—if you are running this notebook live, you will be able to use the sliders to rotate the plot (Figure 5-60):

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

        def plot_3D(elev=30, azim=30, X=X, y=y):
            ax = plt.subplot(projection='3d')
            ax.scatter3D(X[:, 0], X[:, 1], r, c=y, s=50, cmap='autumn')
            ax.view_init(elev=elev, azim=azim)
            ax.set_xlabel('x')
            ax.set_ylabel('y')
            ax.set_zlabel('r')

        interact(plot_3D, elev=[-90, 90], azip=(-180, 180),
                 X=fixed(X), y=fixed(y));
```

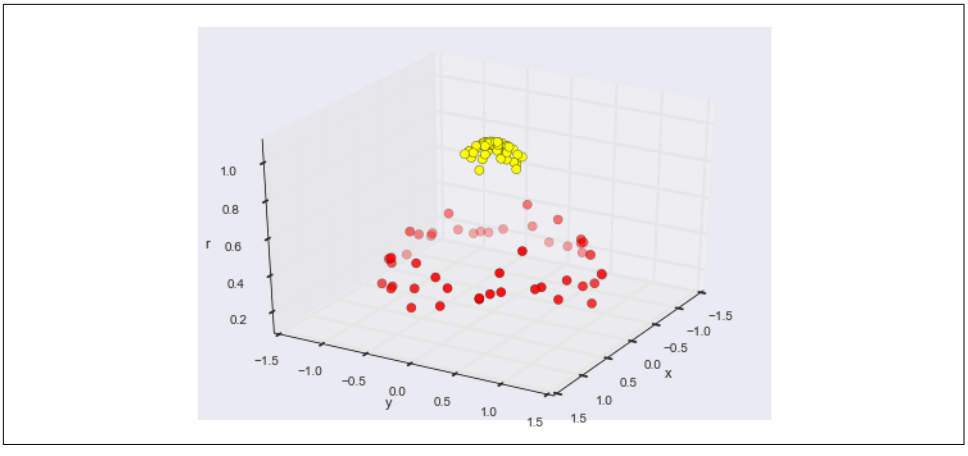

Figure 5-60. A third dimension added to the data allows for linear separation

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

Here 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 (radial basis function) kernel, using the `kernel` model hyperparameter (Figure 5-61):

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

```
Out[14]: SVC(C=1000000.0, cache_size=200, class_weight=None, coef0=0.0,
           decision_function_shape=None, degree=3, gamma='auto', kernel='rbf',
           max_iter=-1, probability=False, random_state=None, shrinking=True,
           tol=0.001, verbose=False)
```

```
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');
```

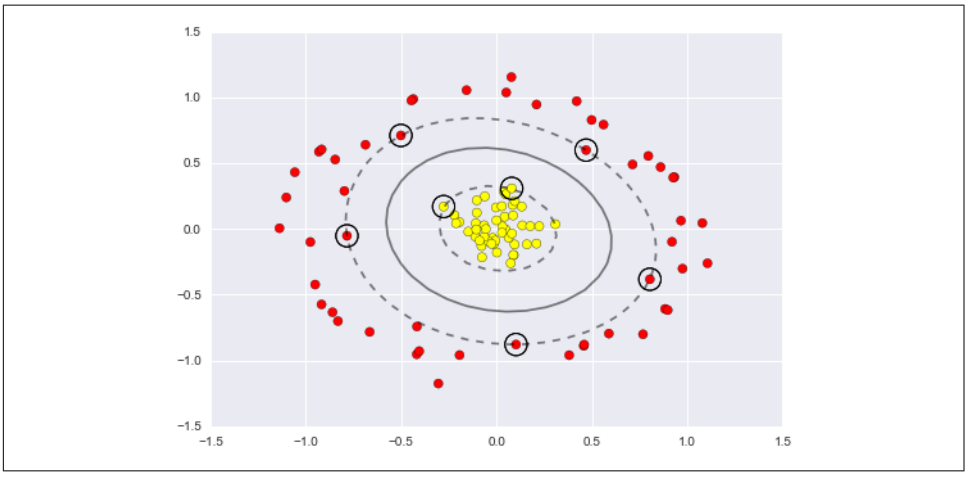

Figure 5-61. Kernel SVM fit to the data

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 so far has centered on 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 (Figure 5-62):

```
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');
```

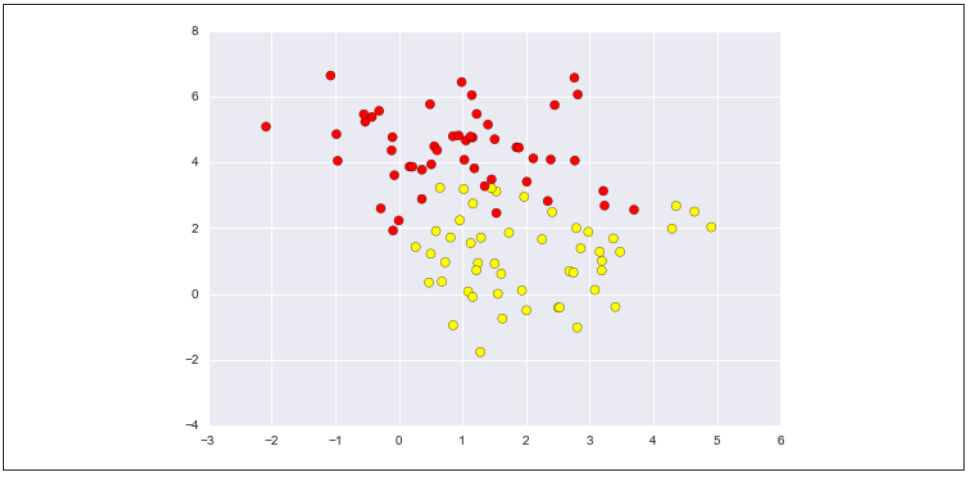

Figure 5-62. Data with some level of overlap

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 very large *C*, the margin is hard, and points cannot lie in it. For smaller *C*, the margin is softer, and can grow to encompass some points.

The plot shown in Figure 5-63 gives a visual picture of how a changing *C* parameter 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)
```

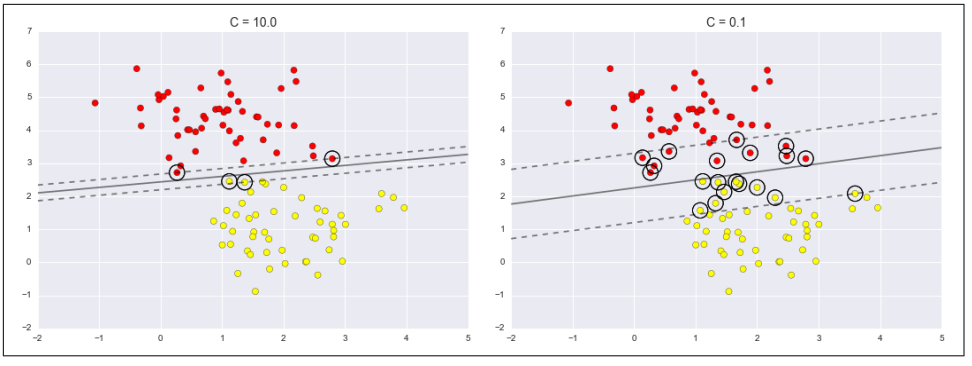

Figure 5-63. The effect of the C parameter on the support vector fit

The optimal value of the *C* parameter will depend on your dataset, and should be tuned via cross-validation or a similar procedure (refer back to "Hyperparameters and Model Validation" on page 359 for further information).

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)
```

```
['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 (Figure 5-64):

```
In[19]: fig, ax = plt.subplots(3, 5)
        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]])
```

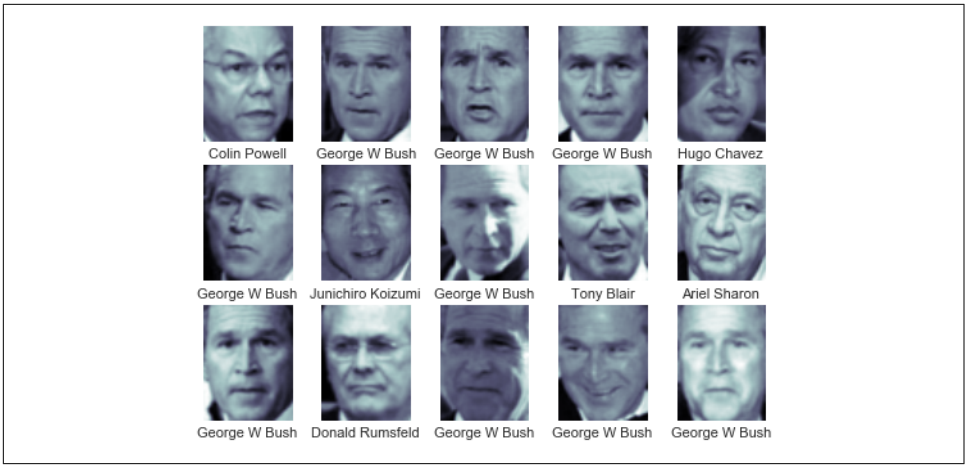

Figure 5-64. Examples from the Labeled Faces in the Wild dataset

Each image contains [62×47] or nearly 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 a principal component analysis (see "In Depth: Principal Component Analysis" on page 433) 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 RandomizedPCA
        from sklearn.pipeline import make_pipeline

        pca = RandomizedPCA(n_components=150, whiten=True, 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 and testing set:

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

Finally, we can use a 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.grid_search 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_)
```

```
CPU times: user 47.8 s, sys: 4.08 s, total: 51.8 s
Wall time: 26 s
{'svc__gamma': 0.001, 'svc__C': 10}
```

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 (Figure 5-65):

```
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);
```

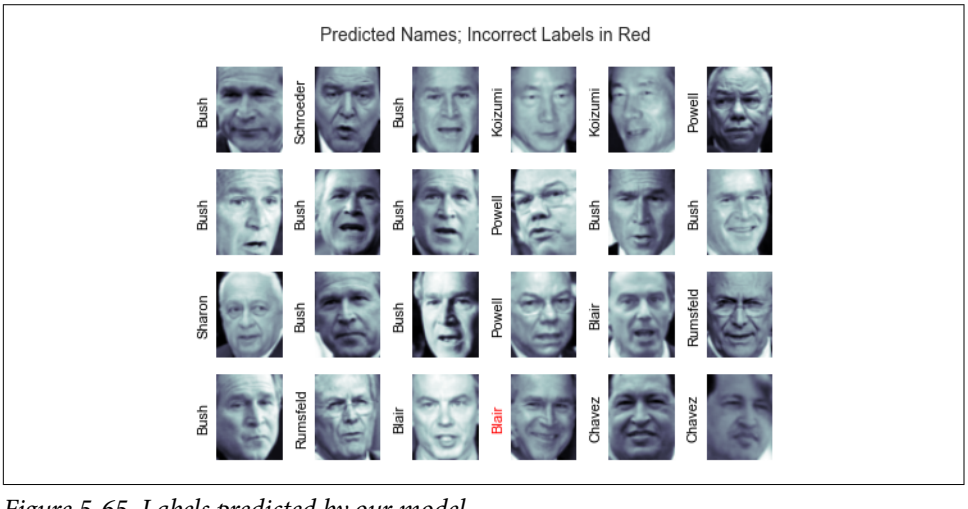

Figure 5-65. Labels predicted by our model

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))
```

```
                   precision    recall  f1-score   support

     Ariel Sharon       0.65      0.73      0.69        15
     Colin Powell       0.81      0.87      0.84        68
  Donald Rumsfeld       0.75      0.87      0.81        31
    George W Bush       0.93      0.83      0.88       126
Gerhard Schroeder       0.86      0.78      0.82        23
      Hugo Chavez       0.93      0.70      0.80        20
Junichiro Koizumi       0.80      1.00      0.89        12
       Tony Blair       0.83      0.93      0.88        42

      avg / total       0.85      0.85      0.85       337
```

We might also display the confusion matrix between these classes (Figure 5-66):

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

Figure 5-66. 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 precropped 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.

Support Vector Machine Summary

We have seen here a brief intuitive introduction to the principals behind support vector machines. These methods are a powerful classification method for a number of reasons:

- Their dependence on relatively few support vectors means that they are very compact models, 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 a challenging regime 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.

In-Depth: Decision Trees and Random Forests

Previously we have looked in depth at a simple generative classifier (naive Bayes; see "In Depth: Naive Bayes Classification" on page 382) and a powerful discriminative classifier (support vector machines; see "In-Depth: Support Vector Machines" on page 405). Here we'll take a look at motivating another powerful algorithm—a non-parametric algorithm called *random forests*. Random forests are an example of an *ensemble* method, a method that relies on aggregating the results of an ensemble of simpler estimators. The somewhat surprising result with such ensemble methods is that the sum can be greater than the parts; that is, a majority vote among a number of estimators can end up being better than 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
       import seaborn as sns; sns.set()
```

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 an animal you come across while on a hike, you might construct the one shown in Figure 5-67.

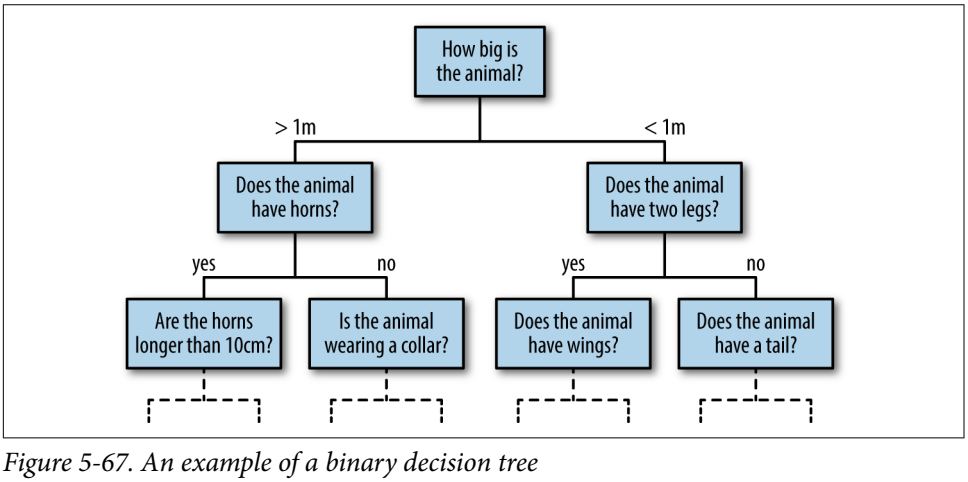

Figure 5-67. An example of a binary decision tree

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 take a look at an example.

Creating a decision tree

Consider the following two-dimensional data, which has one of four class labels (Figure 5-68):

```
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');
```

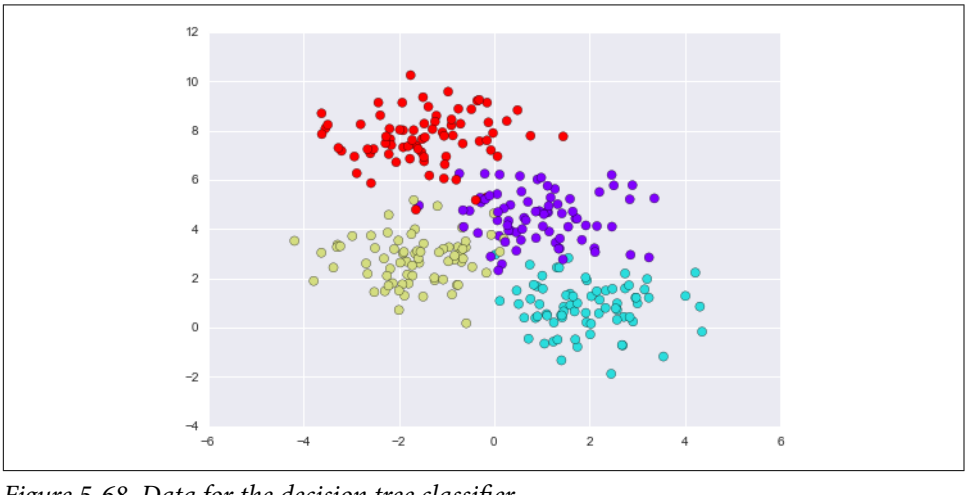

Figure 5-68. Data for the decision tree classifier

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 5-69 presents a visualization of the first four levels of a decision tree classifier for this data.

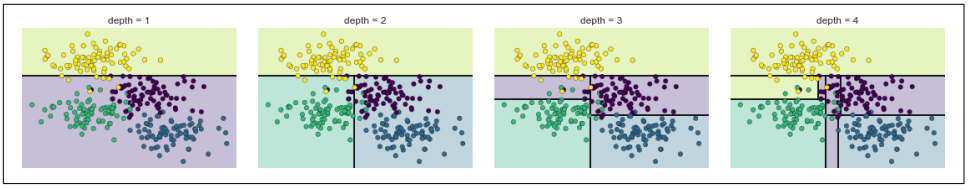

Figure 5-69. Visualization of how the decision tree splits the data

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)
```

Let's write a quick 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, clim=(y.min(), y.max()),
                                  zorder=1)

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

Now we can examine what the decision tree classification looks like (Figure 5-70):

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

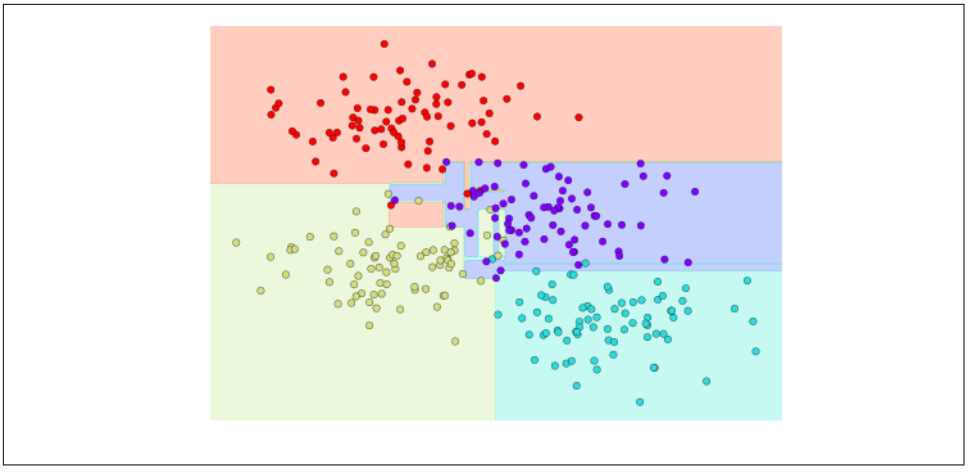

Figure 5-70. Visualization of a decision tree classification

If you're running this notebook live, you can use the helpers script included in the online appendix to bring up an interactive visualization of the decision tree building process (Figure 5-71):

```
In[6]: # helpers_05_08 is found in the online appendix
       # (https://github.com/jakevdp/PythonDataScienceHandbook)
       import helpers_05_08
       helpers_05_08.plot_tree_interactive(X, y);
```

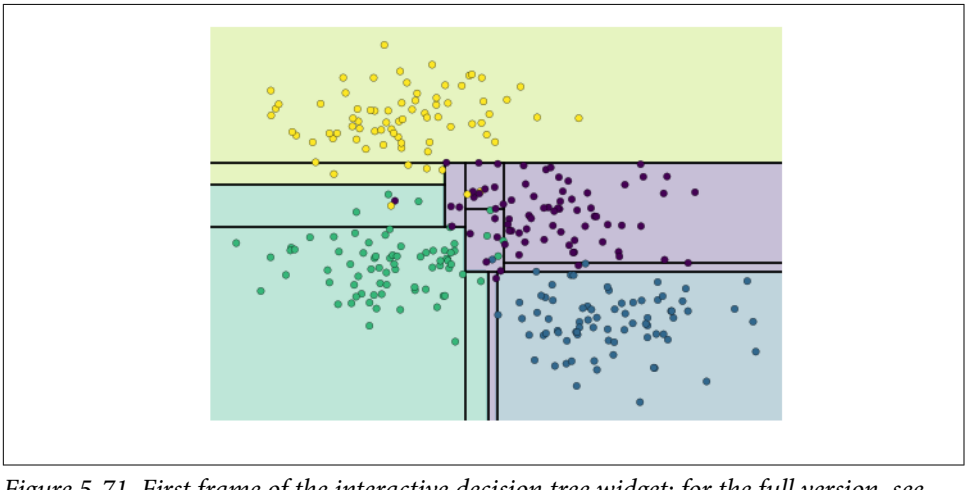

Figure 5-71. First frame of the interactive decision tree widget; for the full version, see the online appendix

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 they are drawn from. Another way to see this overfitting is to look at models trained on different subsets of the data—for example, in Figure 5-72 we train two different trees, each on half of the original data.

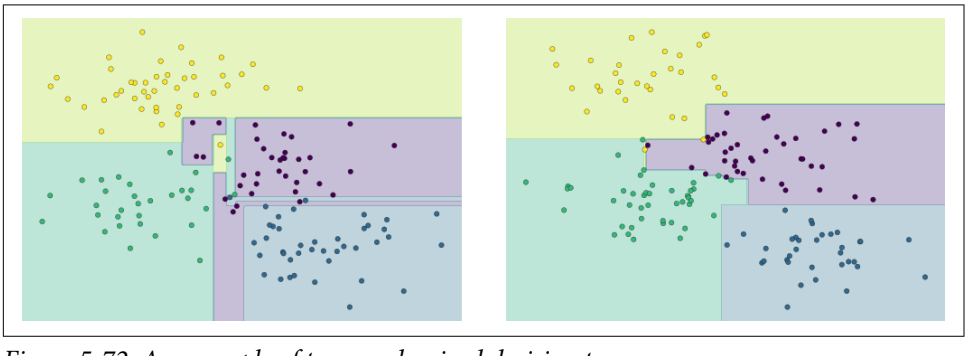

Figure 5-72. An example of two randomized decision trees

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!

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 (Figure 5-73):

```
In[7]: # helpers_05_08 is found in the online appendix
       # (https://github.com/jakevdp/PythonDataScienceHandbook)
       import helpers_05_08
       helpers_05_08.randomized_tree_interactive(X, y)
```

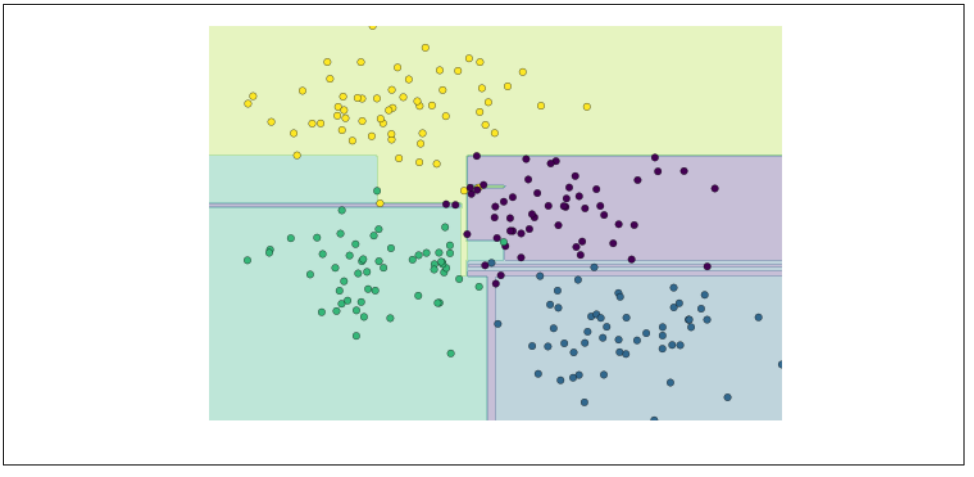

Figure 5-73. First frame of the interactive randomized decision tree widget; for the full version, see the online appendix

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*.

We can do this type of bagging classification manually using Scikit-Learn's `Bagging Classifier` meta-estimator as shown here (Figure 5-74):

```
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)
```

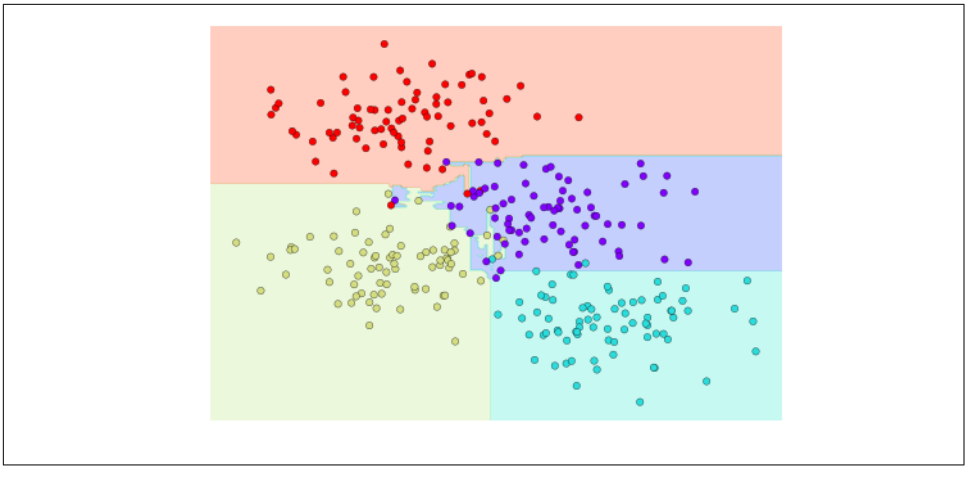

Figure 5-74. Decision boundaries for an ensemble of random decision trees

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 when some stochasticity is injected 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.

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 (Figure 5-75):

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

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

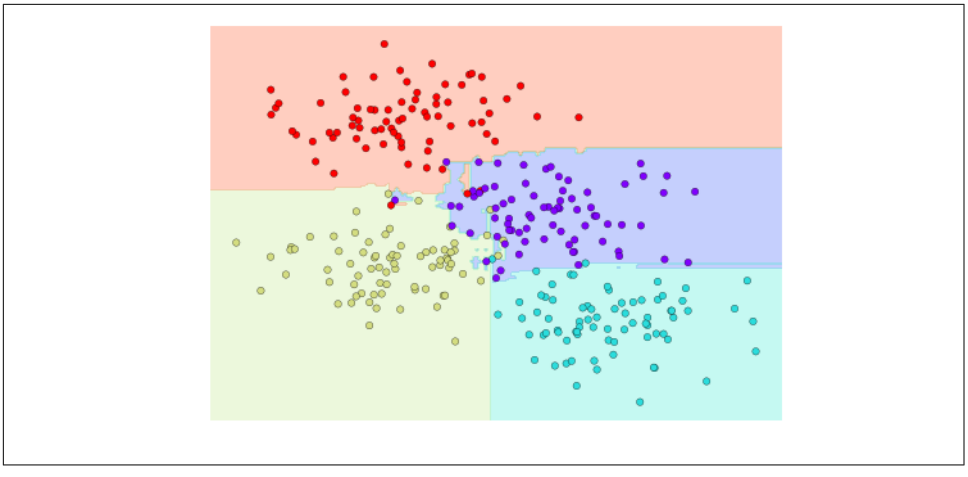

Figure 5-75. Decision boundaries for a random forest, which is an optimized ensemble of decision trees

We see that by averaging over 100 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, 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 (Figure 5-76):

```
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');
```

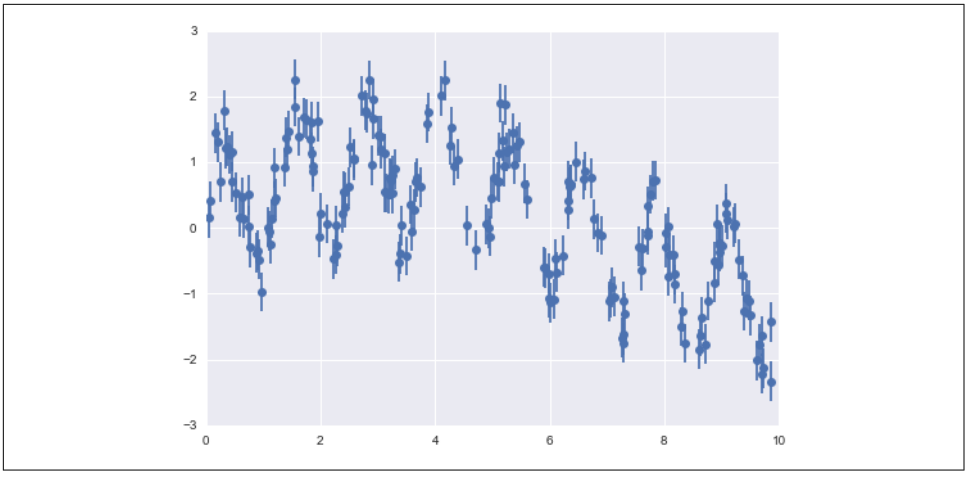

Figure 5-76. Data for random forest regression

Using the random forest regressor, we can find the best-fit curve as follows (Figure 5-77):

```
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);
```

Here the true model is shown by the smooth curve, while the random forest model is shown by the jagged curve. As you can see, the nonparametric random forest model is flexible enough to fit the multiperiod data, without us needing to specify a multiperiod model!

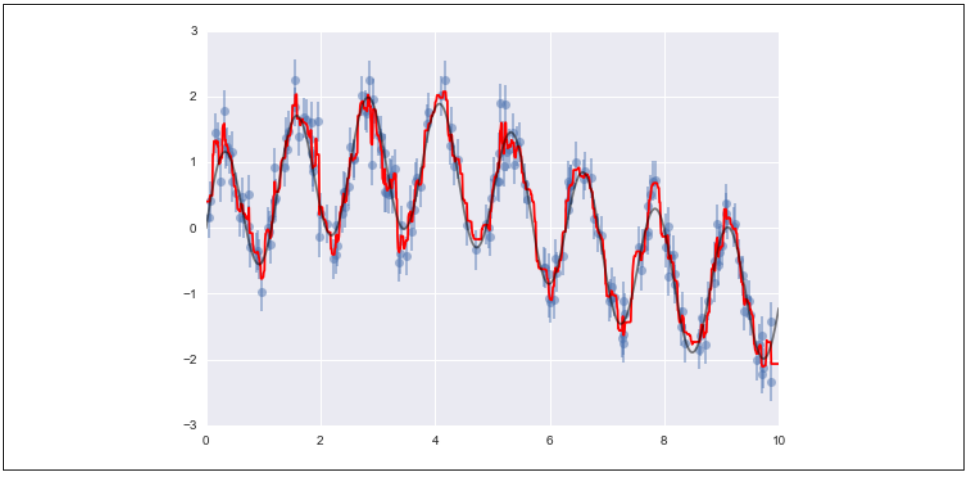

Figure 5-77. Random forest model fit to the data

Example: Random Forest for Classifying Digits

Earlier we took a quick look at the handwritten digits data (see "Introducing Scikit-Learn" on page 343). Let's use that again here to see how the random forest classifier can be used in this context.

```
In[12]: from sklearn.datasets import load_digits
        digits = load_digits()
        digits.keys()
```

```
Out[12]: dict_keys(['target', 'data', 'target_names', 'DESCR', 'images'])
```

To remind us what we're looking at, we'll visualize the first few data points (Figure 5-78):

```
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]))
```

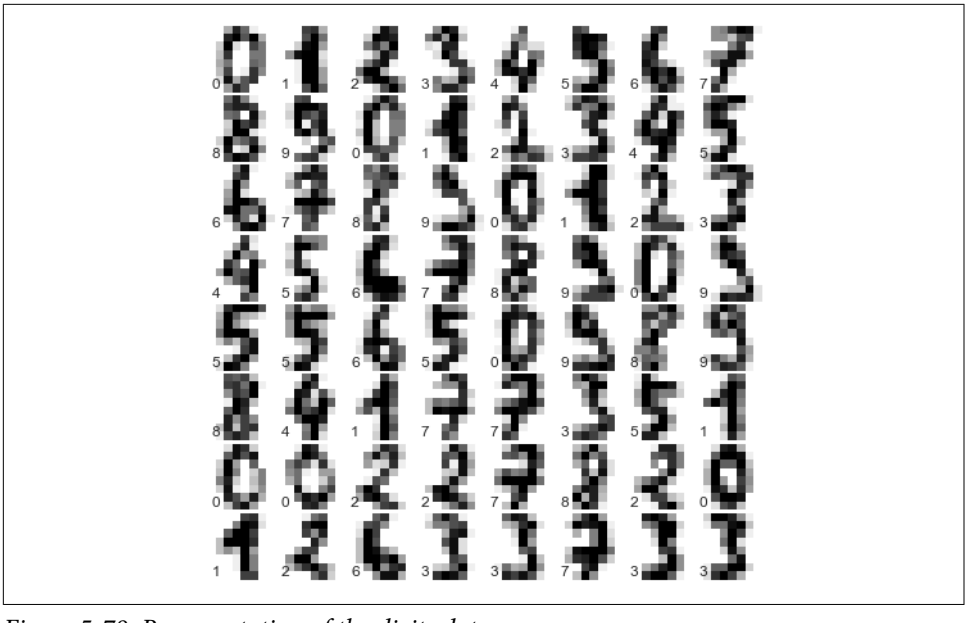

Figure 5-78. Representation of the digits data

We can quickly classify the digits using a random forest as follows (Figure 5-79):

```
In[14]:
from sklearn.cross_validation 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)
```

We can take a look at the classification report for this classifier:

```
In[15]: from sklearn import metrics
        print(metrics.classification_report(ypred, ytest))
```

```
             precision    recall  f1-score   support

          0       1.00      0.97      0.99        38
          1       1.00      0.98      0.99        44
          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       0.96      0.98      0.97        46

avg / total       0.98      0.98      0.98       450
```

And for good measure, plot the confusion matrix (Figure 5-79):

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

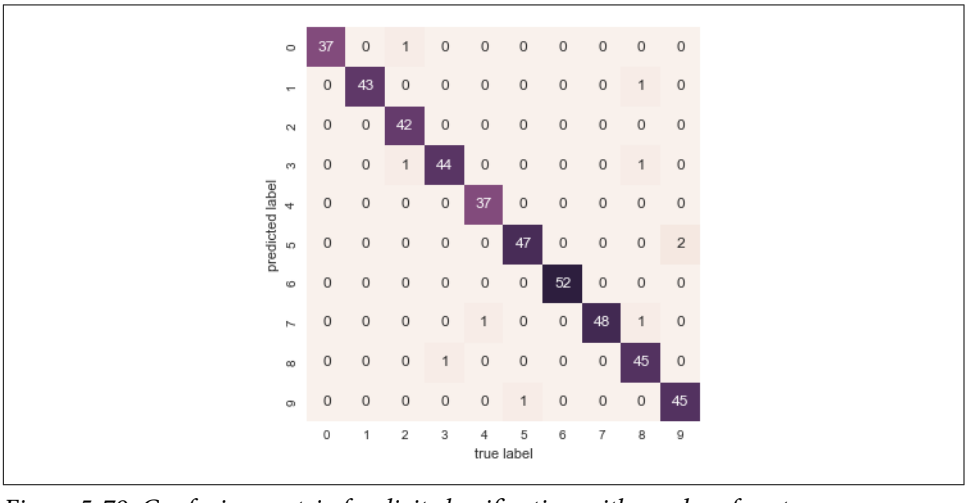

Figure 5-79. Confusion matrix for digit classification with random forests

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

Summary of Random Forests

This section contained a brief introduction to the concept of *ensemble estimators*, and in particular the random forest model—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.

In Depth: Principal Component Analysis

Up until now, we have been looking in depth at supervised learning estimators: those estimators that predict labels based on labeled training data. Here we begin looking at several unsupervised estimators, which can highlight interesting aspects of the data without reference to any known labels.

In this section, we explore what is perhaps one of the most broadly used of unsupervised algorithms, principal component analysis (PCA). PCA is fundamentally a dimensionality reduction algorithm, but it can also be useful as a tool for visualization, for noise filtering, for feature extraction and engineering, and much more. After a brief conceptual discussion of the PCA algorithm, we will see a couple examples of these further applications. We begin with the standard imports:

```
In[1]: %matplotlib inline
       import numpy as np
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
```

Introducing Principal Component Analysis

Principal component analysis is a fast and flexible unsupervised method for dimensionality reduction in data, which we saw briefly in "Introducing Scikit-Learn" on page 343. Its behavior is easiest to visualize by looking at a two-dimensional dataset. Consider the following 200 points (Figure 5-80):

```
In[2]: rng = np.random.RandomState(1)
       X = np.dot(rng.rand(2, 2), rng.randn(2, 200)).T
       plt.scatter(X[:, 0], X[:, 1])
       plt.axis('equal');
```

By eye, it is clear that there is a nearly linear relationship between the x and y variables. This is reminiscent of the linear regression data we explored in "In Depth: Linear Regression" on page 390, but the problem setting here is slightly different: rather than attempting to *predict* the *y* values from the *x* values, the unsupervised learning problem attempts to learn about the *relationship* between the *x* and *y* values.

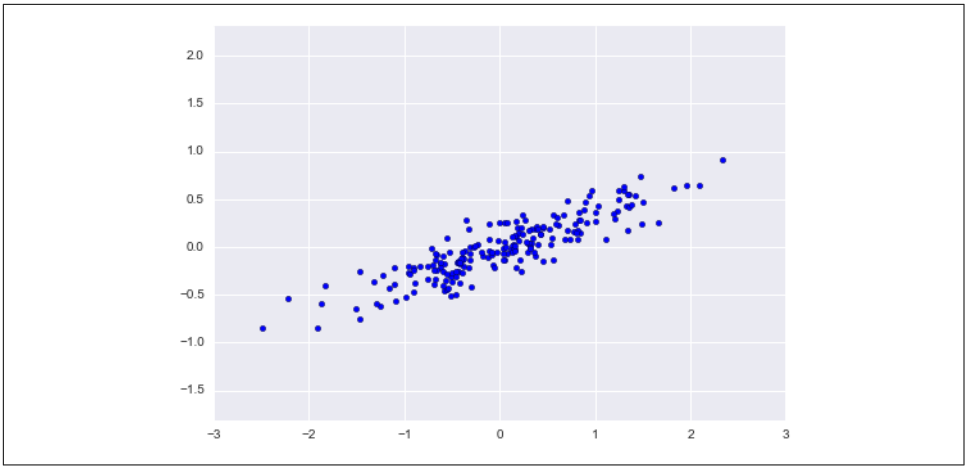

Figure 5-80. Data for demonstration of PCA

In principal component analysis, one quantifies this relationship by finding a list of the *principal axes* in the data, and using those axes to describe the dataset. Using Scikit-Learn's PCA estimator, we can compute this as follows:

```
In[3]: from sklearn.decomposition import PCA
       pca = PCA(n_components=2)
       pca.fit(X)

Out[3]: PCA(copy=True, n_components=2, whiten=False)
```

The fit learns some quantities from the data, most importantly the "components" and "explained variance":

```
In[4]: print(pca.components_)

[[ 0.94446029  0.32862557]
 [ 0.32862557 -0.94446029]]

In[5]: print(pca.explained_variance_)

[ 0.75871884  0.01838551]
```

To see what these numbers mean, let's visualize them as vectors over the input data, using the "components" to define the direction of the vector, and the "explained variance" to define the squared-length of the vector (Figure 5-81):

```
In[6]: def draw_vector(v0, v1, ax=None):
           ax = ax or plt.gca()
           arrowprops=dict(arrowstyle='->',
                           linewidth=2,
                           shrinkA=0, shrinkB=0)
           ax.annotate('', v1, v0, arrowprops=arrowprops)

       # plot data
```

```
plt.scatter(X[:, 0], X[:, 1], alpha=0.2)
for length, vector in zip(pca.explained_variance_, pca.components_):
    v = vector * 3 * np.sqrt(length)
    draw_vector(pca.mean_, pca.mean_ + v)
plt.axis('equal');
```

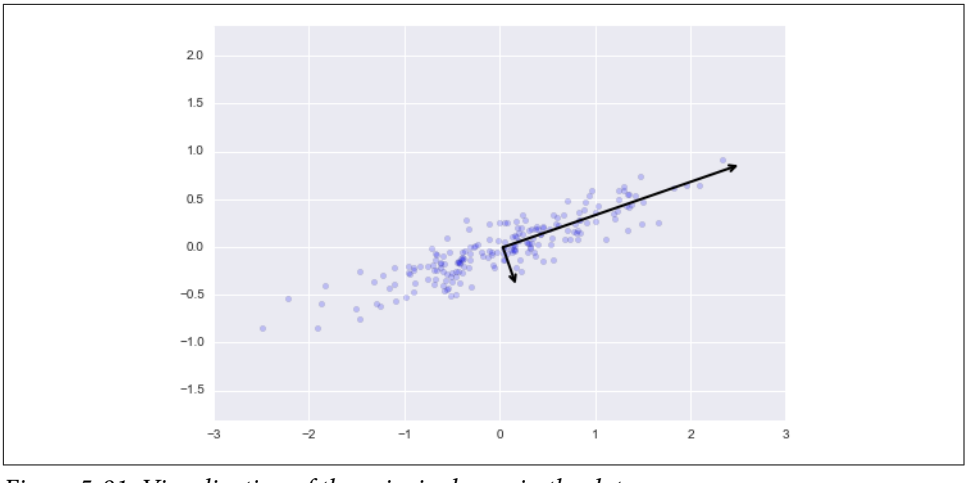

Figure 5-81. Visualization of the principal axes in the data

These vectors represent the *principal axes* of the data, and the length shown in Figure 5-81 is an indication of how "important" that axis is in describing the distribution of the data—more precisely, it is a measure of the variance of the data when projected onto that axis. The projection of each data point onto the principal axes are the "principal components" of the data.

If we plot these principal components beside the original data, we see the plots shown in Figure 5-82.

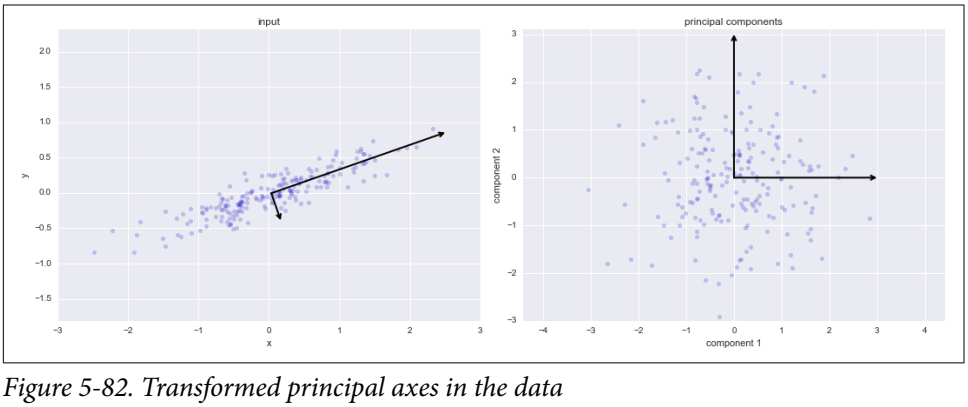

Figure 5-82. Transformed principal axes in the data

This transformation from data axes to principal axes is as an *affine transformation*, which basically means it is composed of a translation, rotation, and uniform scaling.

While this algorithm to find principal components may seem like just a mathematical curiosity, it turns out to have very far-reaching applications in the world of machine learning and data exploration.

PCA as dimensionality reduction

Using PCA for dimensionality reduction involves zeroing out one or more of the smallest principal components, resulting in a lower-dimensional projection of the data that preserves the maximal data variance.

Here is an example of using PCA as a dimensionality reduction transform:

```
In[7]: pca = PCA(n_components=1)
       pca.fit(X)
       X_pca = pca.transform(X)
       print("original shape:   ", X.shape)
       print("transformed shape:", X_pca.shape)
```

```
original shape:    (200, 2)
transformed shape: (200, 1)
```

The transformed data has been reduced to a single dimension. To understand the effect of this dimensionality reduction, we can perform the inverse transform of this reduced data and plot it along with the original data (Figure 5-83):

```
In[8]: X_new = pca.inverse_transform(X_pca)
       plt.scatter(X[:, 0], X[:, 1], alpha=0.2)
       plt.scatter(X_new[:, 0], X_new[:, 1], alpha=0.8)
       plt.axis('equal');
```

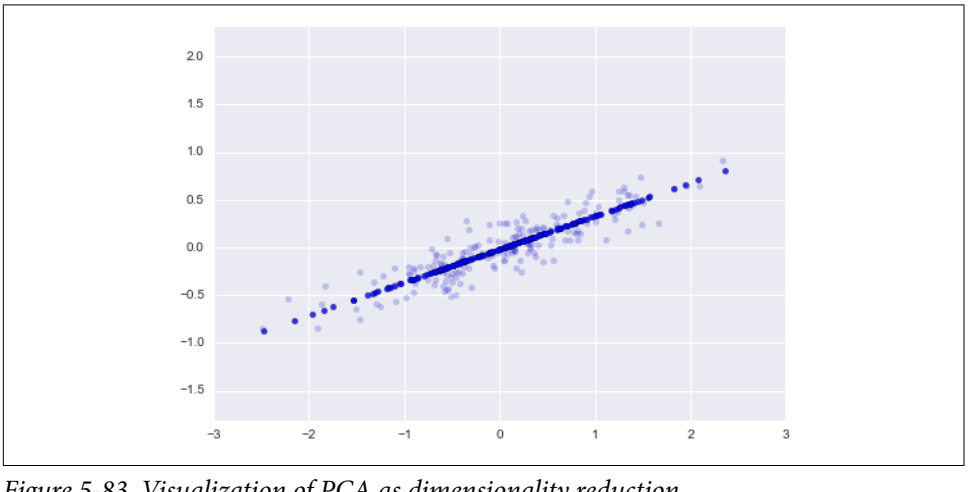

Figure 5-83. Visualization of PCA as dimensionality reduction

The light points are the original data, while the dark points are the projected version. This makes clear what a PCA dimensionality reduction means: the information along the least important principal axis or axes is removed, leaving only the component(s) of the data with the highest variance. The fraction of variance that is cut out (proportional to the spread of points about the line formed in Figure 5-83) is roughly a measure of how much "information" is discarded in this reduction of dimensionality.

This reduced-dimension dataset is in some senses "good enough" to encode the most important relationships between the points: despite reducing the dimension of the data by 50%, the overall relationship between the data points is mostly preserved.

PCA for visualization: Handwritten digits

The usefulness of the dimensionality reduction may not be entirely apparent in only two dimensions, but becomes much more clear when we look at high-dimensional data. To see this, let's take a quick look at the application of PCA to the digits data we saw in "In-Depth: Decision Trees and Random Forests" on page 421.

We start by loading the data:

```
In[9]: from sklearn.datasets import load_digits
       digits = load_digits()
       digits.data.shape
```

```
Out[9]:
(1797, 64)
```

Recall that the data consists of 8×8 pixel images, meaning that they are 64-dimensional. To gain some intuition into the relationships between these points, we can use PCA to project them to a more manageable number of dimensions, say two:

```
In[10]: pca = PCA(2)  # project from 64 to 2 dimensions
        projected = pca.fit_transform(digits.data)
        print(digits.data.shape)
        print(projected.shape)
```

```
(1797, 64)
(1797, 2)
```

We can now plot the first two principal components of each point to learn about the data (Figure 5-84):

```
In[11]: plt.scatter(projected[:, 0], projected[:, 1],
                    c=digits.target, edgecolor='none', alpha=0.5,
                    cmap=plt.cm.get_cmap('spectral', 10))
        plt.xlabel('component 1')
        plt.ylabel('component 2')
        plt.colorbar();
```

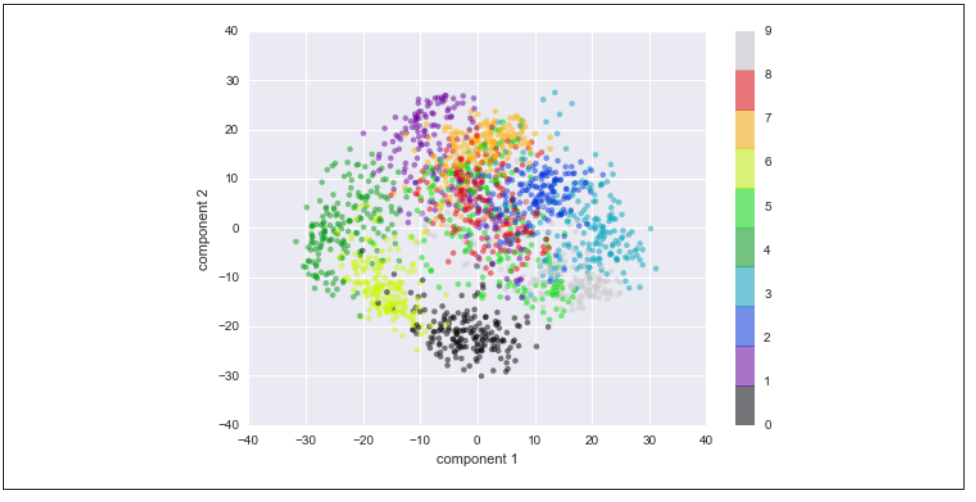

Figure 5-84. PCA applied to the handwritten digits data

Recall what these components mean: the full data is a 64-dimensional point cloud, and these points are the projection of each data point along the directions with the largest variance. Essentially, we have found the optimal stretch and rotation in 64-dimensional space that allows us to see the layout of the digits in two dimensions, and have done this in an unsupervised manner—that is, without reference to the labels.

What do the components mean?

We can go a bit further here, and begin to ask what the reduced dimensions *mean*. This meaning can be understood in terms of combinations of basis vectors. For example, each image in the training set is defined by a collection of 64 pixel values, which we will call the vector x:

$$x = [x_1, x_2, x_3 \cdots x_{64}]$$

One way we can think about this is in terms of a pixel basis. That is, to construct the image, we multiply each element of the vector by the pixel it describes, and then add the results together to build the image:

$$\text{image}(x) = x_1 \cdot (\text{pixel } 1) + x_2 \cdot (\text{pixel } 2) + x_3 \cdot (\text{pixel } 3) \cdots x_{64} \cdot (\text{pixel } 64)$$

One way we might imagine reducing the dimension of this data is to zero out all but a few of these basis vectors. For example, if we use only the first eight pixels, we get an eight-dimensional projection of the data (Figure 5-85), but it is not very reflective of the whole image: we've thrown out nearly 90% of the pixels!

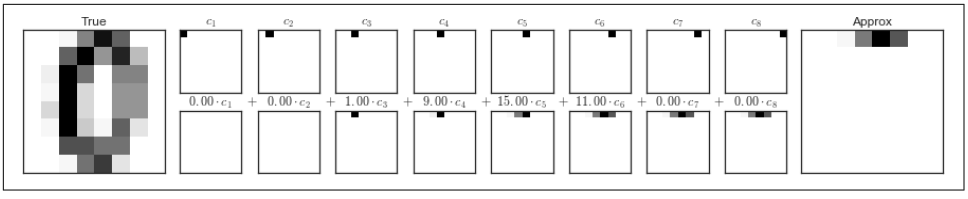

Figure 5-85. A naive dimensionality reduction achieved by discarding pixels

The upper row of panels shows the individual pixels, and the lower row shows the cumulative contribution of these pixels to the construction of the image. Using only eight of the pixel-basis components, we can only construct a small portion of the 64-pixel image. Were we to continue this sequence and use all 64 pixels, we would recover the original image.

But the pixel-wise representation is not the only choice of basis. We can also use other basis functions, which each contain some predefined contribution from each pixel, and write something like:

$$image(x) = \text{mean} + x_1 \cdot (\text{basis } 1) + x_2 \cdot (\text{basis } 2) + x_3 \cdot (\text{basis } 3) \cdots$$

PCA can be thought of as a process of choosing optimal basis functions, such that adding together just the first few of them is enough to suitably reconstruct the bulk of the elements in the dataset. The principal components, which act as the low-dimensional representation of our data, are simply the coefficients that multiply each of the elements in this series. Figure 5-86 is a similar depiction of reconstructing this digit using the mean plus the first eight PCA basis functions.

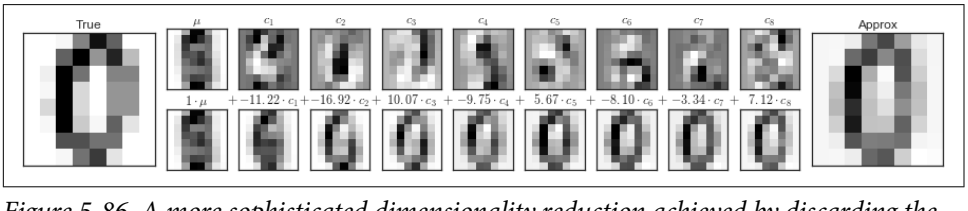

Figure 5-86. A more sophisticated dimensionality reduction achieved by discarding the least important principal components (compare to Figure 5-85)

Unlike the pixel basis, the PCA basis allows us to recover the salient features of the input image with just a mean plus eight components! The amount of each pixel in each component is the corollary of the orientation of the vector in our two-dimensional example. This is the sense in which PCA provides a low-dimensional representation of the data: it discovers a set of basis functions that are more efficient than the native pixel-basis of the input data.

Choosing the number of components

A vital part of using PCA in practice is the ability to estimate how many components are needed to describe the data. We can determine this by looking at the cumulative *explained variance ratio* as a function of the number of components (Figure 5-87):

```
In[12]: pca = PCA().fit(digits.data)
        plt.plot(np.cumsum(pca.explained_variance_ratio_))
        plt.xlabel('number of components')
        plt.ylabel('cumulative explained variance');
```

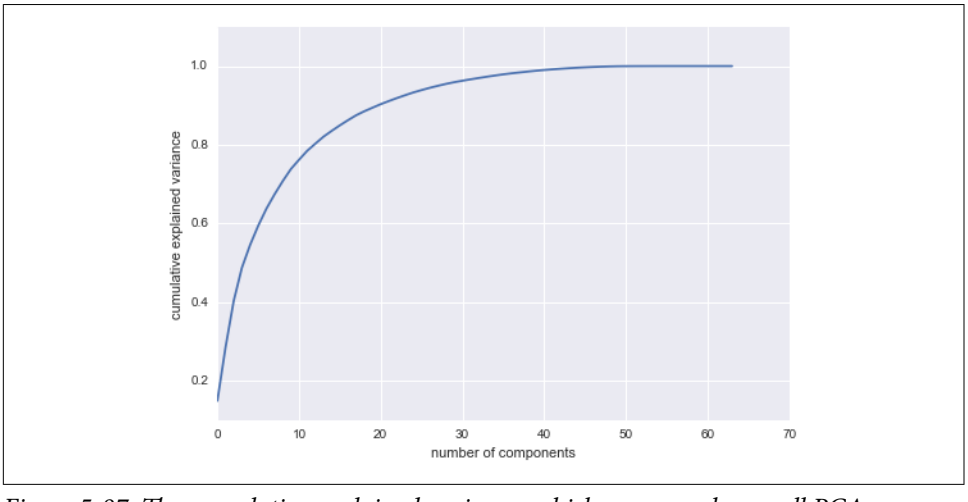

Figure 5-87. The cumulative explained variance, which measures how well PCA preserves the content of the data

This curve quantifies how much of the total, 64-dimensional variance is contained within the first *N* components. For example, we see that with the digits the first 10 components contain approximately 75% of the variance, while you need around 50 components to describe close to 100% of the variance.

Here we see that our two-dimensional projection loses a lot of information (as measured by the explained variance) and that we'd need about 20 components to retain 90% of the variance. Looking at this plot for a high-dimensional dataset can help you understand the level of redundancy present in multiple observations.

PCA as Noise Filtering

PCA can also be used as a filtering approach for noisy data. The idea is this: any components with variance much larger than the effect of the noise should be relatively unaffected by the noise. So if you reconstruct the data using just the largest subset of principal components, you should be preferentially keeping the signal and throwing out the noise.

Let's see how this looks with the digits data. First we will plot several of the input noise-free data (Figure 5-88):

```
In[13]: def plot_digits(data):
            fig, axes = plt.subplots(4, 10, figsize=(10, 4),
                                     subplot_kw={'xticks':[], 'yticks':[]},
                                     gridspec_kw=dict(hspace=0.1, wspace=0.1))
            for i, ax in enumerate(axes.flat):
                ax.imshow(data[i].reshape(8, 8),
                          cmap='binary', interpolation='nearest',
                          clim=(0, 16))
        plot_digits(digits.data)
```

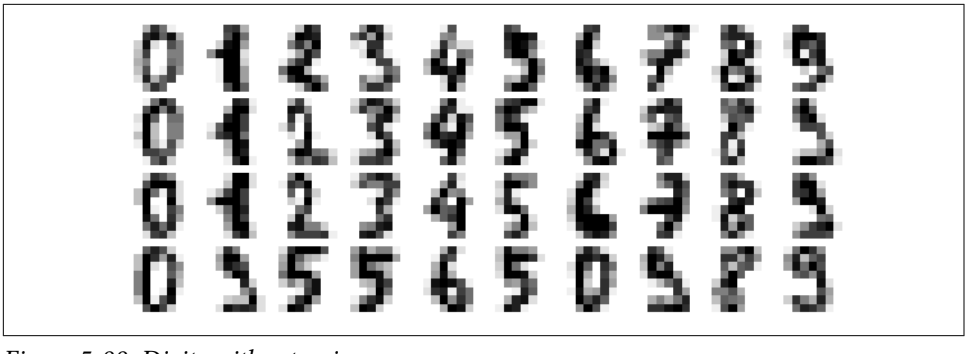

Figure 5-88. Digits without noise

Now let's add some random noise to create a noisy dataset, and replot it (Figure 5-89):

```
In[14]: np.random.seed(42)
        noisy = np.random.normal(digits.data, 4)
        plot_digits(noisy)
```

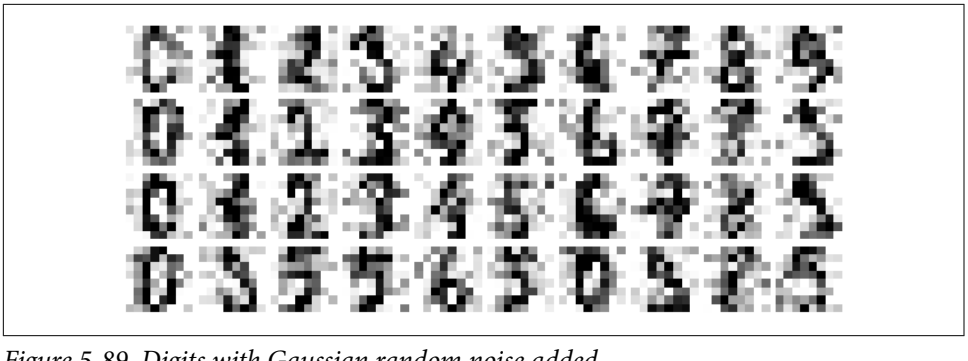

Figure 5-89. Digits with Gaussian random noise added

It's clear by eye that the images are noisy, and contain spurious pixels. Let's train a PCA on the noisy data, requesting that the projection preserve 50% of the variance:

```
In[15]: pca = PCA(0.50).fit(noisy)
        pca.n_components_
```

```
Out[15]: 12
```

Here 50% of the variance amounts to 12 principal components. Now we compute these components, and then use the inverse of the transform to reconstruct the filtered digits (Figure 5-90):

```
In[16]: components = pca.transform(noisy)
        filtered = pca.inverse_transform(components)
        plot_digits(filtered)
```

Figure 5-90. Digits "denoised" using PCA

This signal preserving/noise filtering property makes PCA a very useful feature selection routine—for example, rather than training a classifier on very high-dimensional data, you might instead train the classifier on the lower-dimensional representation, which will automatically serve to filter out random noise in the inputs.

Example: Eigenfaces

Earlier we explored an example of using a PCA projection as a feature selector for facial recognition with a support vector machine ("In-Depth: Support Vector Machines" on page 405). Here we will take a look back and explore a bit more of what went into that. Recall that we were using the Labeled Faces in the Wild dataset made available through Scikit-Learn:

```
In[17]: from sklearn.datasets import fetch_lfw_people
        faces = fetch_lfw_people(min_faces_per_person=60)
        print(faces.target_names)
        print(faces.images.shape)
```

```
['Ariel Sharon' 'Colin Powell' 'Donald Rumsfeld' 'George W Bush'
 'Gerhard Schroeder' 'Hugo Chavez' 'Junichiro Koizumi' 'Tony Blair']
(1348, 62, 47)
```

Let's take a look at the principal axes that span this dataset. Because this is a large dataset, we will use `RandomizedPCA`—it contains a randomized method to approxi-

mate the first *N* principal components much more quickly than the standard `PCA` estimator, and thus is very useful for high-dimensional data (here, a dimensionality of nearly 3,000). We will take a look at the first 150 components:

```
In[18]: from sklearn.decomposition import RandomizedPCA
        pca = RandomizedPCA(150)
        pca.fit(faces.data)

Out[18]: RandomizedPCA(copy=True, iterated_power=3, n_components=150,
                       random_state=None, whiten=False)
```

In this case, it can be interesting to visualize the images associated with the first several principal components (these components are technically known as "eigenvectors," so these types of images are often called "eigenfaces"). As you can see in Figure 5-91, they are as creepy as they sound:

```
In[19]: fig, axes = plt.subplots(3, 8, figsize=(9, 4),
                                 subplot_kw={'xticks':[], 'yticks':[]},
                                 gridspec_kw=dict(hspace=0.1, wspace=0.1))
        for i, ax in enumerate(axes.flat):
            ax.imshow(pca.components_[i].reshape(62, 47), cmap='bone')
```

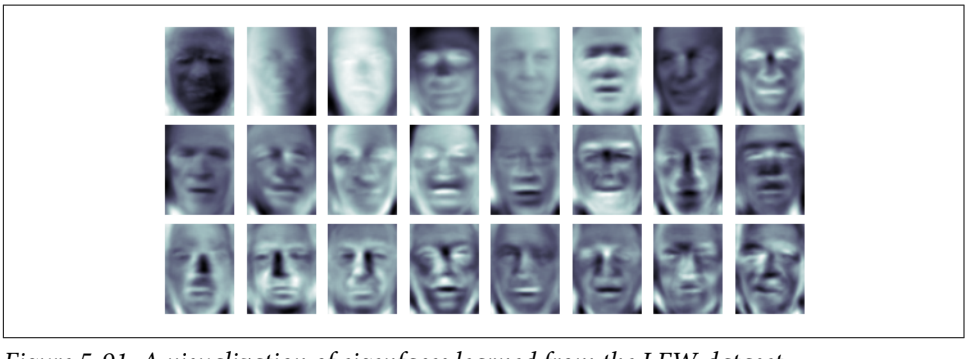

Figure 5-91. A visualization of eigenfaces learned from the LFW dataset

The results are very interesting, and give us insight into how the images vary: for example, the first few eigenfaces (from the top left) seem to be associated with the angle of lighting on the face, and later principal vectors seem to be picking out certain features, such as eyes, noses, and lips. Let's take a look at the cumulative variance of these components to see how much of the data information the projection is preserving (Figure 5-92):

```
In[20]: plt.plot(np.cumsum(pca.explained_variance_ratio_))
        plt.xlabel('number of components')
        plt.ylabel('cumulative explained variance');
```

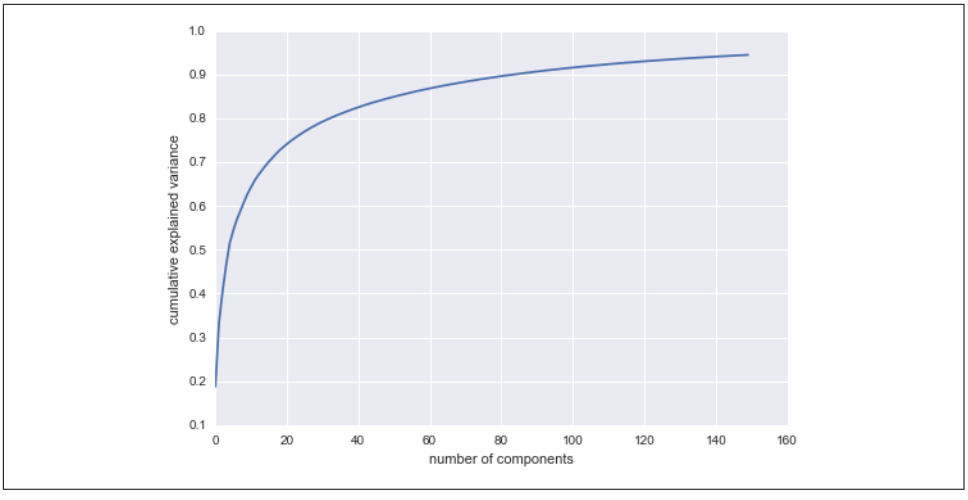

Figure 5-92. Cumulative explained variance for the LFW data

We see that these 150 components account for just over 90% of the variance. That would lead us to believe that using these 150 components, we would recover most of the essential characteristics of the data. To make this more concrete, we can compare the input images with the images reconstructed from these 150 components (Figure 5-93):

```
In[21]: # Compute the components and projected faces
        pca = RandomizedPCA(150).fit(faces.data)
        components = pca.transform(faces.data)
        projected = pca.inverse_transform(components)

In[22]: # Plot the results
        fig, ax = plt.subplots(2, 10, figsize=(10, 2.5),
                               subplot_kw={'xticks':[], 'yticks':[]},
                               gridspec_kw=dict(hspace=0.1, wspace=0.1))
        for i in range(10):
            ax[0, i].imshow(faces.data[i].reshape(62, 47), cmap='binary_r')
            ax[1, i].imshow(projected[i].reshape(62, 47), cmap='binary_r')

        ax[0, 0].set_ylabel('full-dim\ninput')
        ax[1, 0].set_ylabel('150-dim\nreconstruction');
```

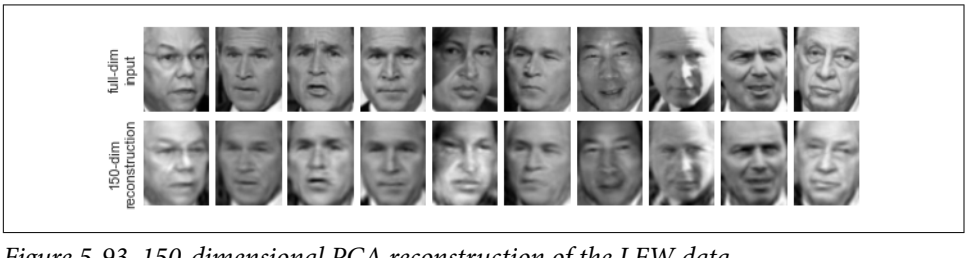

Figure 5-93. 150-dimensional PCA reconstruction of the LFW data

The top row here shows the input images, while the bottom row shows the reconstruction of the images from just 150 of the ~3,000 initial features. This visualization makes clear why the PCA feature selection used in "In-Depth: Support Vector Machines" on page 405 was so successful: although it reduces the dimensionality of the data by nearly a factor of 20, the projected images contain enough information that we might, by eye, recognize the individuals in the image. What this means is that our classification algorithm needs to be trained on 150-dimensional data rather than 3,000-dimensional data, which depending on the particular algorithm we choose, can lead to a much more efficient classification.

Principal Component Analysis Summary

In this section we have discussed the use of principal component analysis for dimensionality reduction, for visualization of high-dimensional data, for noise filtering, and for feature selection within high-dimensional data. Because of the versatility and interpretability of PCA, it has been shown to be effective in a wide variety of contexts and disciplines. Given any high-dimensional dataset, I tend to start with PCA in order to visualize the relationship between points (as we did with the digits), to understand the main variance in the data (as we did with the eigenfaces), and to understand the intrinsic dimensionality (by plotting the explained variance ratio). Certainly PCA is not useful for every high-dimensional dataset, but it offers a straightforward and efficient path to gaining insight into high-dimensional data.

PCA's main weakness is that it tends to be highly affected by outliers in the data. For this reason, many robust variants of PCA have been developed, many of which act to iteratively discard data points that are poorly described by the initial components. Scikit-Learn contains a couple interesting variants on PCA, including `RandomizedPCA` and `SparsePCA`, both also in the `sklearn.decomposition` submodule. `RandomizedPCA`, which we saw earlier, uses a nondeterministic method to quickly approximate the first few principal components in very high-dimensional data, while `SparsePCA` introduces a regularization term (see "In Depth: Linear Regression" on page 390) that serves to enforce sparsity of the components.

In the following sections, we will look at other unsupervised learning methods that build on some of the ideas of PCA.

In-Depth: Manifold Learning

We have seen how principal component analysis can be used in the dimensionality reduction task—reducing the number of features of a dataset while maintaining the essential relationships between the points. While PCA is flexible, fast, and easily interpretable, it does not perform so well when there are *nonlinear* relationships within the data; we will see some examples of these below.

To address this deficiency, we can turn to a class of methods known as *manifold learning*—a class of unsupervised estimators that seeks to describe datasets as low-dimensional manifolds embedded in high-dimensional spaces. When you think of a manifold, I'd suggest imagining a sheet of paper: this is a two-dimensional object that lives in our familiar three-dimensional world, and can be bent or rolled in two dimensions. In the parlance of manifold learning, we can think of this sheet as a two-dimensional manifold embedded in three-dimensional space.

Rotating, reorienting, or stretching the piece of paper in three-dimensional space doesn't change the flat geometry of the paper: such operations are akin to linear embeddings. If you bend, curl, or crumple the paper, it is still a two-dimensional manifold, but the embedding into the three-dimensional space is no longer linear. Manifold learning algorithms would seek to learn about the fundamental two-dimensional nature of the paper, even as it is contorted to fill the three-dimensional space.

Here we will demonstrate a number of manifold methods, going most deeply into a couple techniques: multidimensional scaling (MDS), locally linear embedding (LLE), and isometric mapping (Isomap). We begin with the standard imports:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
       import numpy as np
```

Manifold Learning: "HELLO"

To make these concepts more clear, let's start by generating some two-dimensional data that we can use to define a manifold. Here is a function that will create data in the shape of the word "HELLO":

```
In[2]:
def make_hello(N=1000, rseed=42):
    # Make a plot with "HELLO" text; save as PNG
    fig, ax = plt.subplots(figsize=(4, 1))
    fig.subplots_adjust(left=0, right=1, bottom=0, top=1)
    ax.axis('off')
    ax.text(0.5, 0.4, 'HELLO', va='center', ha='center', weight='bold', size=85)
    fig.savefig('hello.png')
    plt.close(fig)

    # Open this PNG and draw random points from it
    from matplotlib.image import imread
    data = imread('hello.png')[::-1, :, 0].T
    rng = np.random.RandomState(rseed)
    X = rng.rand(4 * N, 2)
    i, j = (X * data.shape).astype(int).T
    mask = (data[i, j] < 1)
    X = X[mask]
```

```
    X[:, 0] *= (data.shape[0] / data.shape[1])
    X = X[:N]
    return X[np.argsort(X[:, 0])]
```

Let's call the function and visualize the resulting data (Figure 5-94):

```
In[3]: X = make_hello(1000)
       colorize = dict(c=X[:, 0], cmap=plt.cm.get_cmap('rainbow', 5))
       plt.scatter(X[:, 0], X[:, 1], **colorize)
       plt.axis('equal');
```

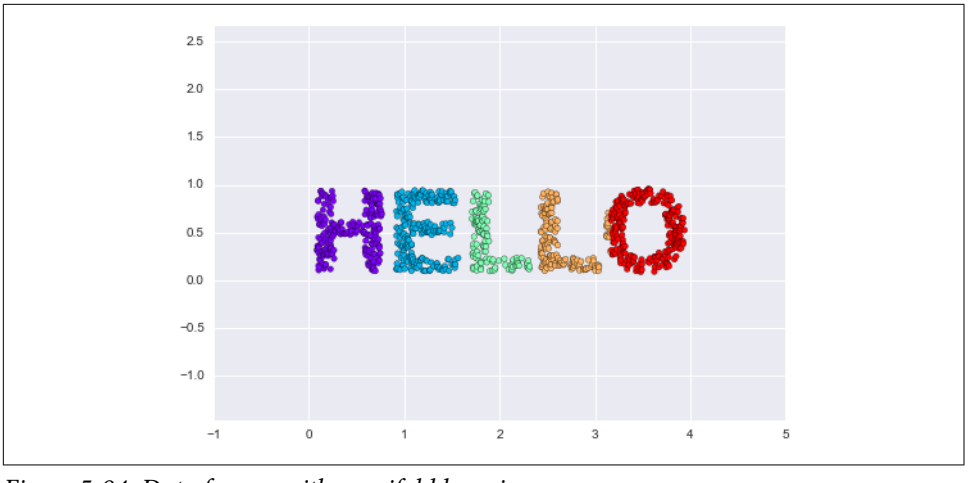

Figure 5-94. Data for use with manifold learning

The output is two dimensional, and consists of points drawn in the shape of the word "HELLO". This data form will help us to see visually what these algorithms are doing.

Multidimensional Scaling (MDS)

Looking at data like this, we can see that the particular choice of *x* and *y* values of the dataset are not the most fundamental description of the data: we can scale, shrink, or rotate the data, and the "HELLO" will still be apparent. For example, if we use a rotation matrix to rotate the data, the *x* and *y* values change, but the data is still fundamentally the same (Figure 5-95):

```
In[4]: def rotate(X, angle):
           theta = np.deg2rad(angle)
           R = [[np.cos(theta), np.sin(theta)],
                [-np.sin(theta), np.cos(theta)]]
           return np.dot(X, R)

       X2 = rotate(X, 20) + 5
       plt.scatter(X2[:, 0], X2[:, 1], **colorize)
       plt.axis('equal');
```

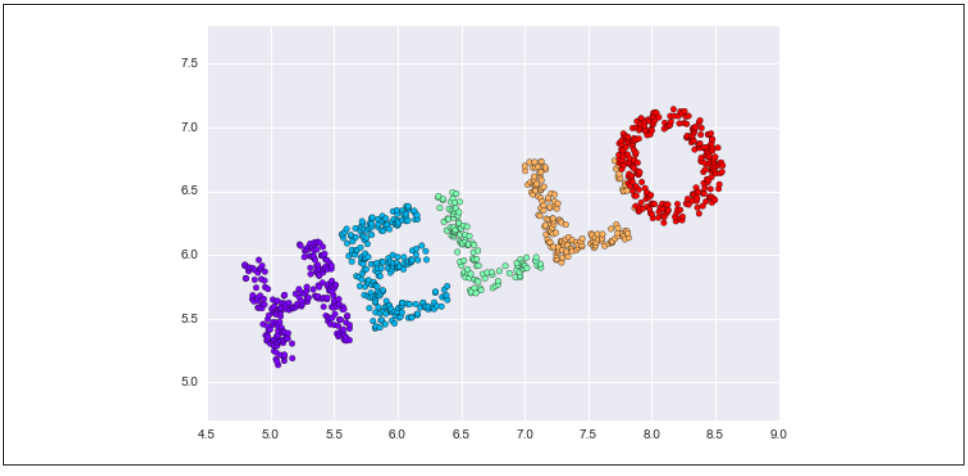

Figure 5-95. Rotated dataset

This tells us that the x and y values are not necessarily fundamental to the relationships in the data. What *is* fundamental, in this case, is the *distance* between each point and the other points in the dataset. A common way to represent this is to use a distance matrix: for N points, we construct an $N \times N$ array such that entry (i, j) contains the distance between point i and point j. Let's use Scikit-Learn's efficient `pairwise_distances` function to do this for our original data:

```
In[5]: from sklearn.metrics import pairwise_distances
       D = pairwise_distances(X)
       D.shape
```

```
Out[5]: (1000, 1000)
```

As promised, for our N=1,000 points, we obtain a 1,000×1,000 matrix, which can be visualized as shown in Figure 5-96:

```
In[6]: plt.imshow(D, zorder=2, cmap='Blues', interpolation='nearest')
       plt.colorbar();
```

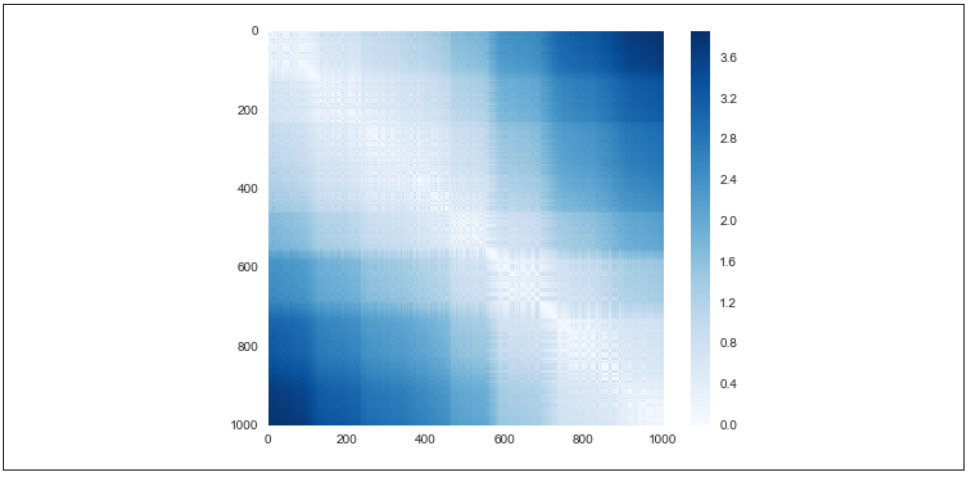

Figure 5-96. Visualization of the pairwise distances between points

If we similarly construct a distance matrix for our rotated and translated data, we see that it is the same:

```
In[7]: D2 = pairwise_distances(X2)
       np.allclose(D, D2)

Out[7]: True
```

This distance matrix gives us a representation of our data that is invariant to rotations and translations, but the visualization of the matrix is not entirely intuitive. In the representation presented in Figure 5-96, we have lost any visible sign of the interesting structure in the data: the "HELLO" that we saw before.

Further, while computing this distance matrix from the (x, y) coordinates is straightforward, transforming the distances back into *x* and *y* coordinates is rather difficult. This is exactly what the multidimensional scaling algorithm aims to do: given a distance matrix between points, it recovers a *D*-dimensional coordinate representation of the data. Let's see how it works for our distance matrix, using the `precomputed` dissimilarity to specify that we are passing a distance matrix (Figure 5-97):

```
In[8]: from sklearn.manifold import MDS
       model = MDS(n_components=2, dissimilarity='precomputed', random_state=1)
       out = model.fit_transform(D)
       plt.scatter(out[:, 0], out[:, 1], **colorize)
       plt.axis('equal');
```

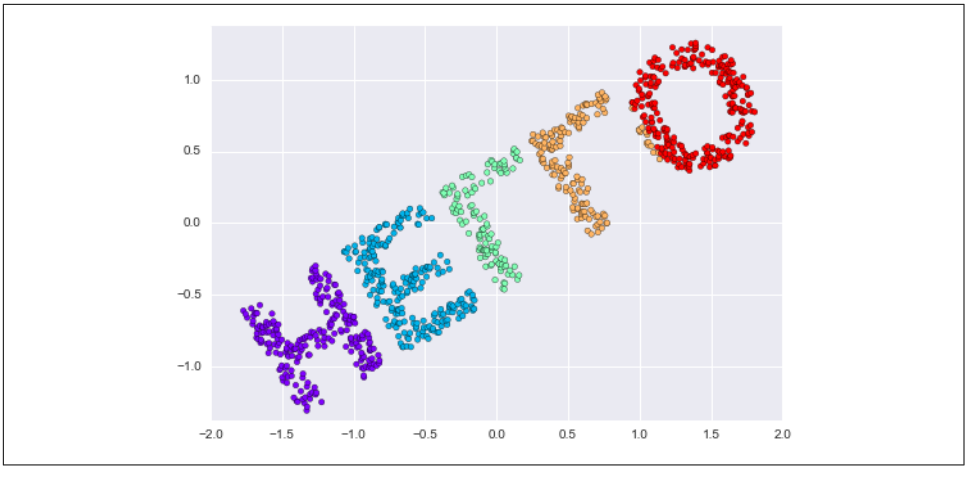

Figure 5-97. An MDS embedding computed from the pairwise distances

The MDS algorithm recovers one of the possible two-dimensional coordinate representations of our data, using *only* the $N \times N$ distance matrix describing the relationship between the data points.

MDS as Manifold Learning

The usefulness of this becomes more apparent when we consider the fact that distance matrices can be computed from data in *any* dimension. So, for example, instead of simply rotating the data in the two-dimensional plane, we can project it into three dimensions using the following function (essentially a three-dimensional generalization of the rotation matrix used earlier):

```
In[9]: def random_projection(X, dimension=3, rseed=42):
           assert dimension >= X.shape[1]
           rng = np.random.RandomState(rseed)
           C = rng.randn(dimension, dimension)
           e, V = np.linalg.eigh(np.dot(C, C.T))
           return np.dot(X, V[:X.shape[1]])

       X3 = random_projection(X, 3)
       X3.shape

Out[9]: (1000, 3)
```

Let's visualize these points to see what we're working with (Figure 5-98):

```
In[10]: from mpl_toolkits import mplot3d
        ax = plt.axes(projection='3d')
        ax.scatter3D(X3[:, 0], X3[:, 1], X3[:, 2],
                     **colorize)
        ax.view_init(azim=70, elev=50)
```

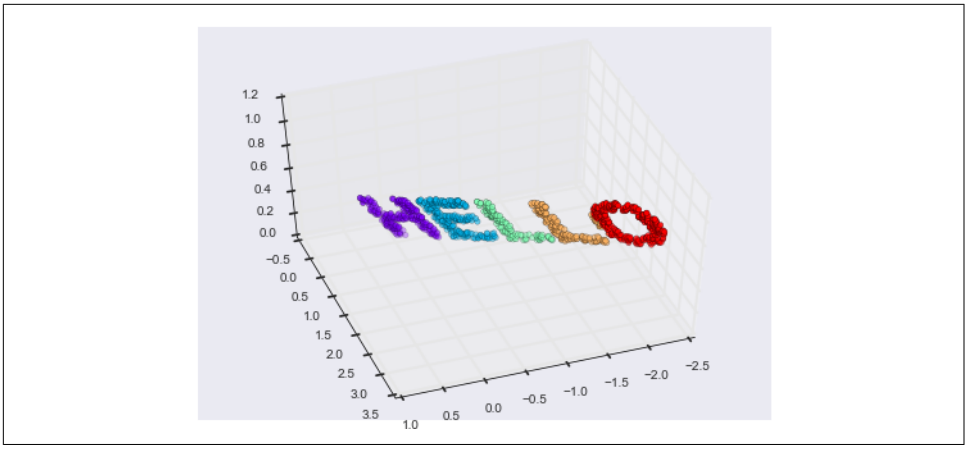

Figure 5-98. Data embedded linearly into three dimensions

We can now ask the `MDS` estimator to input this three-dimensional data, compute the distance matrix, and then determine the optimal two-dimensional embedding for this distance matrix. The result recovers a representation of the original data (Figure 5-99):

```
In[11]: model = MDS(n_components=2, random_state=1)
        out3 = model.fit_transform(X3)
        plt.scatter(out3[:, 0], out3[:, 1], **colorize)
        plt.axis('equal');
```

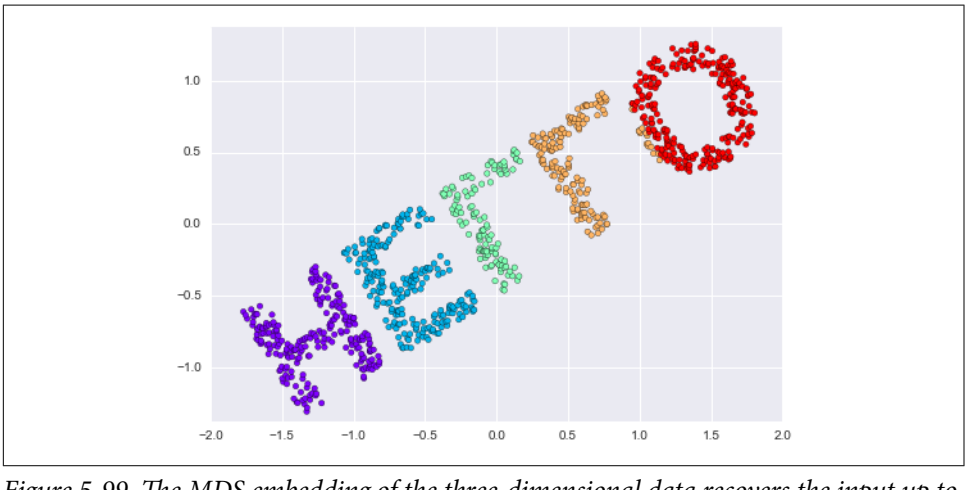

Figure 5-99. The MDS embedding of the three-dimensional data recovers the input up to a rotation and reflection

This is essentially the goal of a manifold learning estimator: given high-dimensional embedded data, it seeks a low-dimensional representation of the data that preserves

certain relationships within the data. In the case of MDS, the quantity preserved is the distance between every pair of points.

Nonlinear Embeddings: Where MDS Fails

Our discussion so far has considered *linear* embeddings, which essentially consist of rotations, translations, and scalings of data into higher-dimensional spaces. Where MDS breaks down is when the embedding is nonlinear—that is, when it goes beyond this simple set of operations. Consider the following embedding, which takes the input and contorts it into an "S" shape in three dimensions:

```
In[12]: def make_hello_s_curve(X):
            t = (X[:, 0] - 2) * 0.75 * np.pi
            x = np.sin(t)
            y = X[:, 1]
            z = np.sign(t) * (np.cos(t) - 1)
            return np.vstack((x, y, z)).T

        XS = make_hello_s_curve(X)
```

This is again three-dimensional data, but we can see that the embedding is much more complicated (Figure 5-100):

```
In[13]: from mpl_toolkits import mplot3d
        ax = plt.axes(projection='3d')
        ax.scatter3D(XS[:, 0], XS[:, 1], XS[:, 2],
                     **colorize);
```

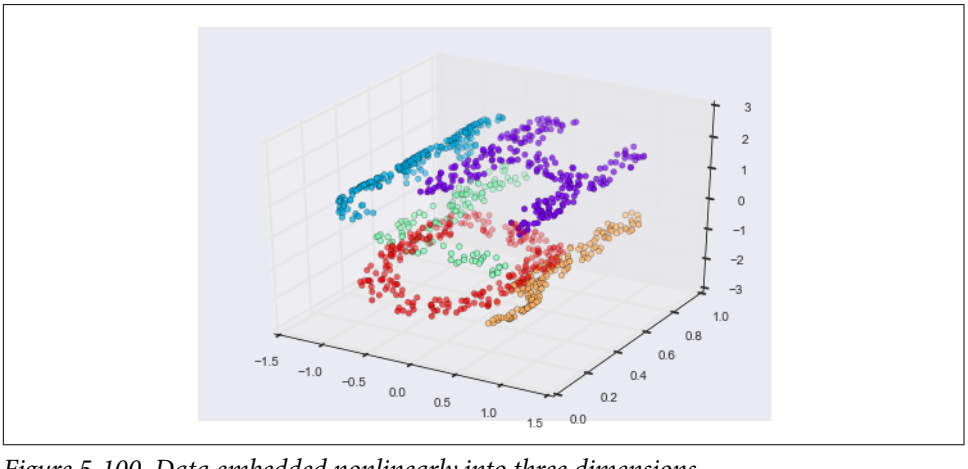

Figure 5-100. Data embedded nonlinearly into three dimensions

The fundamental relationships between the data points are still there, but this time the data has been transformed in a nonlinear way: it has been wrapped up into the shape of an "S."

If we try a simple MDS algorithm on this data, it is not able to "unwrap" this nonlinear embedding, and we lose track of the fundamental relationships in the embedded manifold (Figure 5-101):

```
In[14]: from sklearn.manifold import MDS
        model = MDS(n_components=2, random_state=2)
        outS = model.fit_transform(XS)
        plt.scatter(outS[:, 0], outS[:, 1], **colorize)
        plt.axis('equal');
```

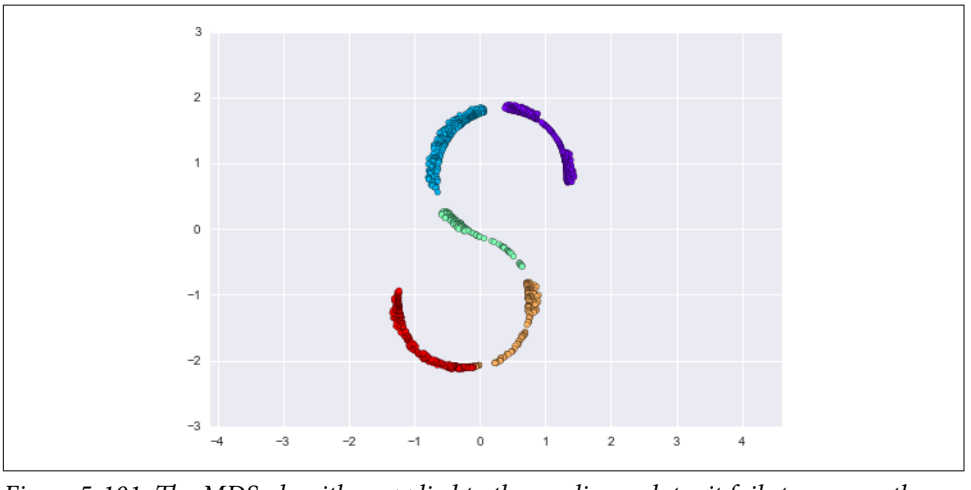

Figure 5-101. The MDS algorithm applied to the nonlinear data; it fails to recover the underlying structure

The best two-dimensional *linear* embedding does not unwrap the S-curve, but instead throws out the original y-axis.

Nonlinear Manifolds: Locally Linear Embedding

How can we move forward here? Stepping back, we can see that the source of the problem is that MDS tries to preserve distances between faraway points when constructing the embedding. But what if we instead modified the algorithm such that it only preserves distances between nearby points? The resulting embedding would be closer to what we want.

Visually, we can think of it as illustrated in Figure 5-102.

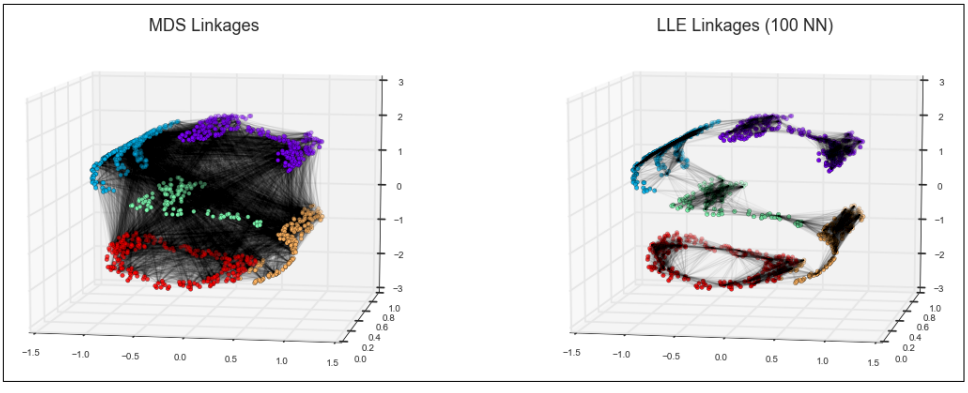

Figure 5-102. Representation of linkages between points within MDS and LLE

Here each faint line represents a distance that should be preserved in the embedding. On the left is a representation of the model used by MDS: it tries to preserve the distances between each pair of points in the dataset. On the right is a representation of the model used by a manifold learning algorithm called locally linear embedding (LLE): rather than preserving *all* distances, it instead tries to preserve only the distances between *neighboring points*: in this case, the nearest 100 neighbors of each point.

Thinking about the left panel, we can see why MDS fails: there is no way to flatten this data while adequately preserving the length of every line drawn between the two points. For the right panel, on the other hand, things look a bit more optimistic. We could imagine unrolling the data in a way that keeps the lengths of the lines approximately the same. This is precisely what LLE does, through a global optimization of a cost function reflecting this logic.

LLE comes in a number of flavors; here we will use the *modified LLE* algorithm to recover the embedded two-dimensional manifold. In general, modified LLE does better than other flavors of the algorithm at recovering well-defined manifolds with very little distortion (Figure 5-103):

```
In[15]:
from sklearn.manifold import LocallyLinearEmbedding
model = LocallyLinearEmbedding(n_neighbors=100, n_components=2, method='modified',
                               eigen_solver='dense')
out = model.fit_transform(XS)

fig, ax = plt.subplots()
ax.scatter(out[:, 0], out[:, 1], **colorize)
ax.set_ylim(0.15, -0.15);
```

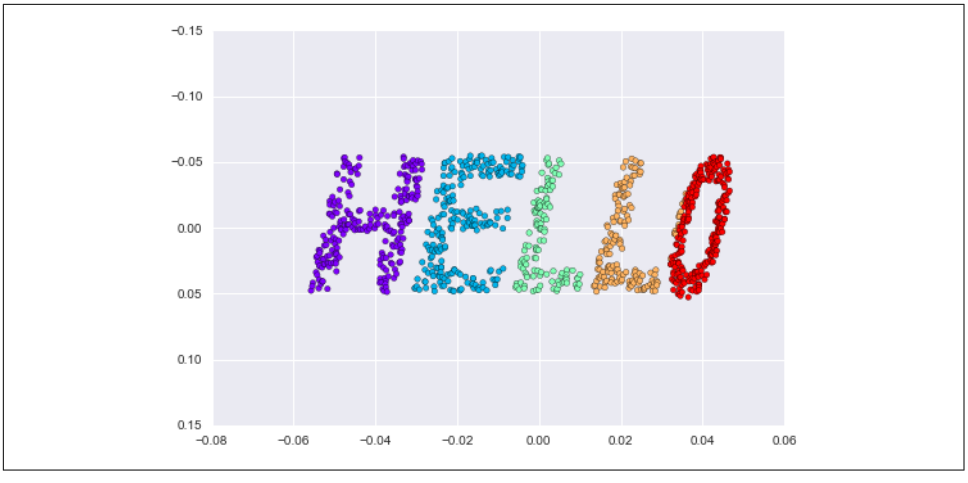

Figure 5-103. Locally linear embedding can recover the underlying data from a nonlinearly embedded input

The result remains somewhat distorted compared to our original manifold, but captures the essential relationships in the data!

Some Thoughts on Manifold Methods

Though this story and motivation is compelling, in practice manifold learning techniques tend to be finicky enough that they are rarely used for anything more than simple qualitative visualization of high-dimensional data.

The following are some of the particular challenges of manifold learning, which all contrast poorly with PCA:

- In manifold learning, there is no good framework for handling missing data. In contrast, there are straightforward iterative approaches for missing data in PCA.
- In manifold learning, the presence of noise in the data can "short-circuit" the manifold and drastically change the embedding. In contrast, PCA naturally filters noise from the most important components.
- The manifold embedding result is generally highly dependent on the number of neighbors chosen, and there is generally no solid quantitative way to choose an optimal number of neighbors. In contrast, PCA does not involve such a choice.
- In manifold learning, the globally optimal number of output dimensions is difficult to determine. In contrast, PCA lets you find the output dimension based on the explained variance.
- In manifold learning, the meaning of the embedded dimensions is not always clear. In PCA, the principal components have a very clear meaning.

- In manifold learning the computational expense of manifold methods scales as $O[N^2]$ or $O[N^3]$. For PCA, there exist randomized approaches that are generally much faster (though see the megaman package for some more scalable implementations of manifold learning).

With all that on the table, the only clear advantage of manifold learning methods over PCA is their ability to preserve nonlinear relationships in the data; for that reason I tend to explore data with manifold methods only after first exploring them with PCA.

Scikit-Learn implements several common variants of manifold learning beyond Isomap and LLE: the Scikit-Learn documentation has a nice discussion and comparison of them. Based on my own experience, I would give the following recommendations:

- For toy problems such as the S-curve we saw before, locally linear embedding (LLE) and its variants (especially *modified LLE*), perform very well. This is implemented in `sklearn.manifold.LocallyLinearEmbedding`.
- For high-dimensional data from real-world sources, LLE often produces poor results, and isometric mapping (Isomap) seems to generally lead to more meaningful embeddings. This is implemented in `sklearn.manifold.Isomap`.
- For data that is highly clustered, *t-distributed stochastic neighbor embedding* (t-SNE) seems to work very well, though can be very slow compared to other methods. This is implemented in `sklearn.manifold.TSNE`.

If you're interested in getting a feel for how these work, I'd suggest running each of the methods on the data in this section.

Example: Isomap on Faces

One place manifold learning is often used is in understanding the relationship between high-dimensional data points. A common case of high-dimensional data is images; for example, a set of images with 1,000 pixels each can be thought of as collection of points in 1,000 dimensions—the brightness of each pixel in each image defines the coordinate in that dimension.

Here let's apply Isomap on some faces data. We will use the Labeled Faces in the Wild dataset, which we previously saw in "In-Depth: Support Vector Machines" on page 405 and "In Depth: Principal Component Analysis" on page 433. Running this command will download the data and cache it in your home directory for later use:

```
In[16]: from sklearn.datasets import fetch_lfw_people
        faces = fetch_lfw_people(min_faces_per_person=30)
        faces.data.shape

Out[16]: (2370, 2914)
```

We have 2,370 images, each with 2,914 pixels. In other words, the images can be thought of as data points in a 2,914-dimensional space!

Let's quickly visualize several of these images to see what we're working with (Figure 5-104):

```
In[17]: fig, ax = plt.subplots(4, 8, subplot_kw=dict(xticks=[], yticks=[]))
        for i, axi in enumerate(ax.flat):
            axi.imshow(faces.images[i], cmap='gray')
```

Figure 5-104. Examples of the input faces

We would like to plot a low-dimensional embedding of the 2,914-dimensional data to learn the fundamental relationships between the images. One useful way to start is to compute a PCA, and examine the explained variance ratio, which will give us an idea of how many linear features are required to describe the data (Figure 5-105):

```
In[18]: from sklearn.decomposition import RandomizedPCA
        model = RandomizedPCA(100).fit(faces.data)
        plt.plot(np.cumsum(model.explained_variance_ratio_))
        plt.xlabel('n components')
        plt.ylabel('cumulative variance');
```

We see that for this data, nearly 100 components are required to preserve 90% of the variance. This tells us that the data is intrinsically very high dimensional—it can't be described linearly with just a few components.

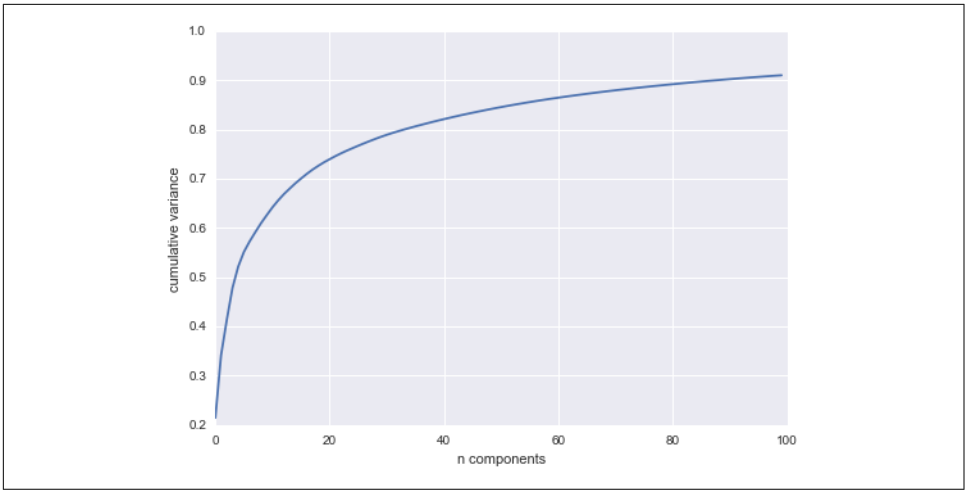

Figure 5-105. Cumulative variance from the PCA projection

When this is the case, nonlinear manifold embeddings like LLE and Isomap can be helpful. We can compute an Isomap embedding on these faces using the same pattern shown before:

```
In[19]: from sklearn.manifold import Isomap
        model = Isomap(n_components=2)
        proj = model.fit_transform(faces.data)
        proj.shape
```

```
Out[19]: (2370, 2)
```

The output is a two-dimensional projection of all the input images. To get a better idea of what the projection tells us, let's define a function that will output image thumbnails at the locations of the projections:

```
In[20]: from matplotlib import offsetbox

        def plot_components(data, model, images=None, ax=None,
                            thumb_frac=0.05, cmap='gray'):
            ax = ax or plt.gca()

            proj = model.fit_transform(data)
            ax.plot(proj[:, 0], proj[:, 1], '.k')

            if images is not None:
                min_dist_2 = (thumb_frac * max(proj.max(0) - proj.min(0))) ** 2
                shown_images = np.array([2 * proj.max(0)])
                for i in range(data.shape[0]):
                    dist = np.sum((proj[i] - shown_images) ** 2, 1)
                    if np.min(dist) < min_dist_2:
                        # don't show points that are too close
                        continue
```

```
                shown_images = np.vstack([shown_images, proj[i]])
                imagebox = offsetbox.AnnotationBbox(
                    offsetbox.OffsetImage(images[i], cmap=cmap),
                                          proj[i])
                ax.add_artist(imagebox)
```

Calling this function now, we see the result (Figure 5-106):

```
In[21]: fig, ax = plt.subplots(figsize=(10, 10))
        plot_components(faces.data,
                        model=Isomap(n_components=2),
                        images=faces.images[:, ::2, ::2])
```

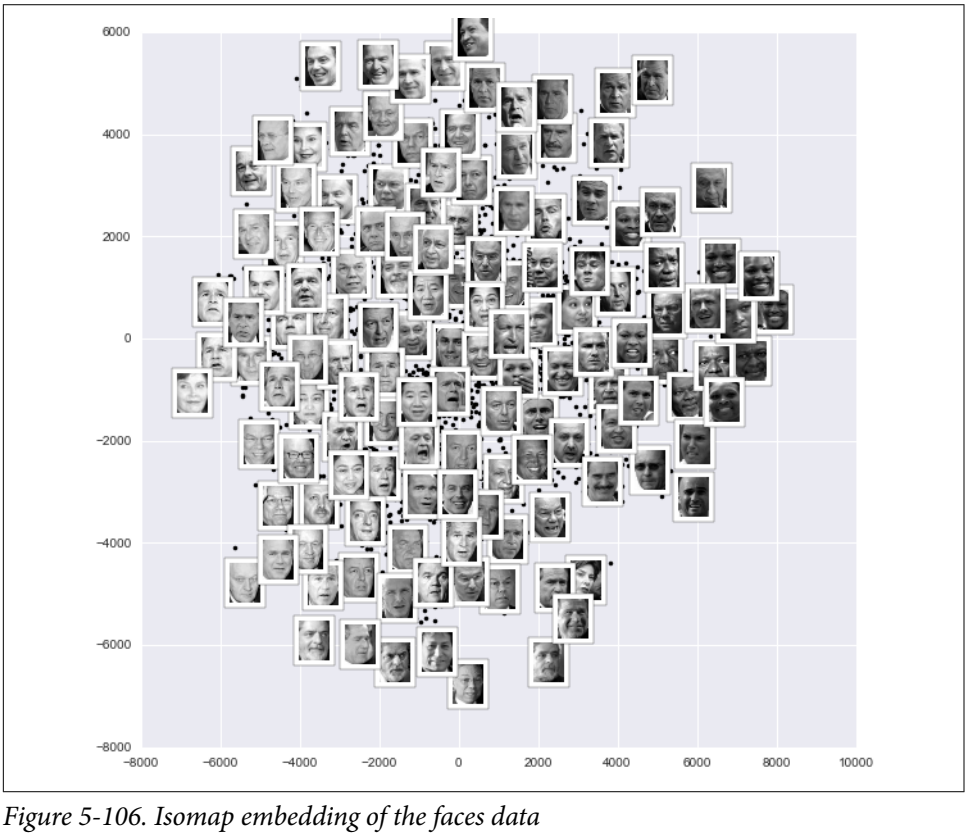

Figure 5-106. Isomap embedding of the faces data

The result is interesting: the first two Isomap dimensions seem to describe global image features: the overall darkness or lightness of the image from left to right, and the general orientation of the face from bottom to top. This gives us a nice visual indication of some of the fundamental features in our data.

We could then go on to classify this data, perhaps using manifold features as inputs to the classification algorithm as we did in "In-Depth: Support Vector Machines" on page 405.

Example: Visualizing Structure in Digits

As another example of using manifold learning for visualization, let's take a look at the MNIST handwritten digits set. This data is similar to the digits we saw in "In-Depth: Decision Trees and Random Forests" on page 421, but with many more pixels per image. It can be downloaded from *http://mldata.org/* with the Scikit-Learn utility:

```
In[22]: from sklearn.datasets import fetch_mldata
        mnist = fetch_mldata('MNIST original')
        mnist.data.shape

Out[22]: (70000, 784)
```

This consists of 70,000 images, each with 784 pixels (i.e., the images are 28×28). As before, we can take a look at the first few images (Figure 5-107):

```
In[23]: fig, ax = plt.subplots(6, 8, subplot_kw=dict(xticks=[], yticks=[]))
        for i, axi in enumerate(ax.flat):
            axi.imshow(mnist.data[1250 * i].reshape(28, 28), cmap='gray_r')
```

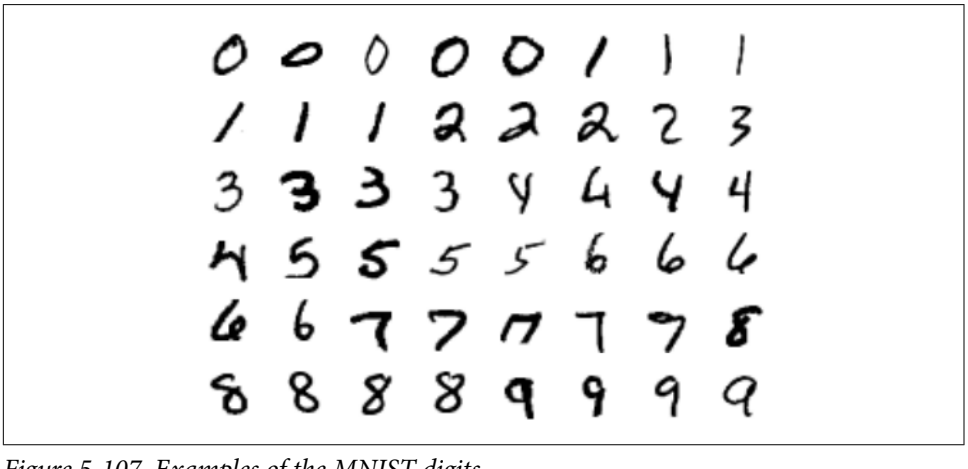

Figure 5-107. Examples of the MNIST digits

This gives us an idea of the variety of handwriting styles in the dataset.

Let's compute a manifold learning projection across the data, illustrated in Figure 5-108. For speed here, we'll only use 1/30 of the data, which is about ~2,000 points (because of the relatively poor scaling of manifold learning, I find that a few thousand samples is a good number to start with for relatively quick exploration before moving to a full calculation):

```
In[24]:
# use only 1/30 of the data: full dataset takes a long time!
data = mnist.data[::30]
target = mnist.target[::30]

model = Isomap(n_components=2)
proj = model.fit_transform(data)
plt.scatter(proj[:, 0], proj[:, 1], c=target, cmap=plt.cm.get_cmap('jet', 10))
plt.colorbar(ticks=range(10))
plt.clim(-0.5, 9.5);
```

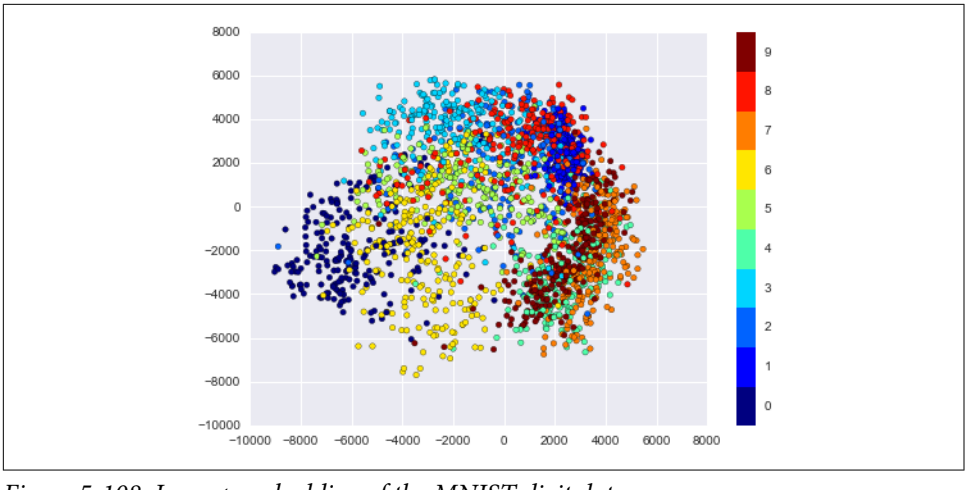

Figure 5-108. Isomap embedding of the MNIST digit data

The resulting scatter plot shows some of the relationships between the data points, but is a bit crowded. We can gain more insight by looking at just a single number at a time (Figure 5-109):

```
In[25]: from sklearn.manifold import Isomap

        # Choose 1/4 of the "1" digits to project
        data = mnist.data[mnist.target == 1][::4]

        fig, ax = plt.subplots(figsize=(10, 10))
        model = Isomap(n_neighbors=5, n_components=2, eigen_solver='dense')
        plot_components(data, model, images=data.reshape((-1, 28, 28)),
                        ax=ax, thumb_frac=0.05, cmap='gray_r')
```

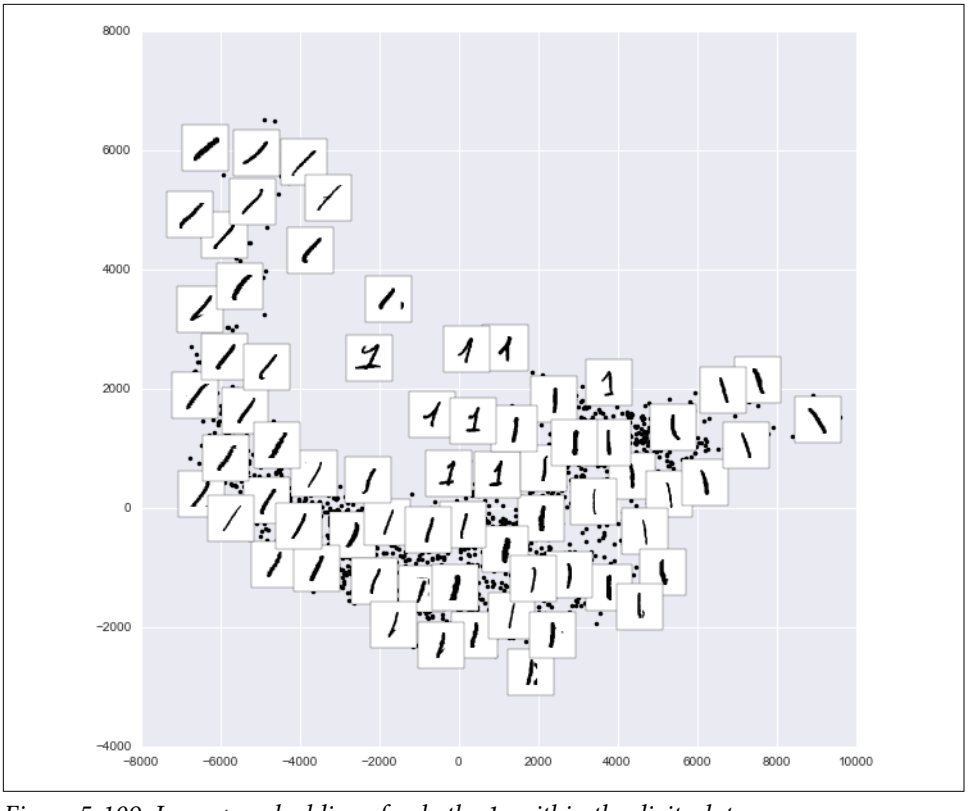

Figure 5-109. Isomap embedding of only the 1s within the digits data

The result gives you an idea of the variety of forms that the number "1" can take within the dataset. The data lies along a broad curve in the projected space, which appears to trace the orientation of the digit. As you move up the plot, you find ones that have hats and/or bases, though these are very sparse within the dataset. The projection lets us identify outliers that have data issues (i.e., pieces of the neighboring digits that snuck into the extracted images).

Now, this in itself may not be useful for the task of classifying digits, but it does help us get an understanding of the data, and may give us ideas about how to move forward, such as how we might want to preprocess the data before building a classification pipeline.

In Depth: k-Means Clustering

In the previous few sections, we have explored one category of unsupervised machine learning models: dimensionality reduction. Here we will move on to another class of unsupervised machine learning models: clustering algorithms. Clustering algorithms

seek to learn, from the properties of the data, an optimal division or discrete labeling of groups of points.

Many clustering algorithms are available in Scikit-Learn and elsewhere, but perhaps the simplest to understand is an algorithm known as *k-means clustering*, which is implemented in `sklearn.cluster.KMeans`. We begin with the standard imports:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()  # for plot styling
       import numpy as np
```

Introducing k-Means

The *k*-means algorithm searches for a predetermined number of clusters within an unlabeled multidimensional dataset. It accomplishes this using a simple conception of what the optimal clustering looks like:

- The "cluster center" is the arithmetic mean of all the points belonging to the cluster.
- Each point is closer to its own cluster center than to other cluster centers.

Those two assumptions are the basis of the *k*-means model. We will soon dive into exactly *how* the algorithm reaches this solution, but for now let's take a look at a simple dataset and see the *k*-means result.

First, let's generate a two-dimensional dataset containing four distinct blobs. To emphasize that this is an unsupervised algorithm, we will leave the labels out of the visualization (Figure 5-110):

```
In[2]: from sklearn.datasets.samples_generator import make_blobs
       X, y_true = make_blobs(n_samples=300, centers=4,
                              cluster_std=0.60, random_state=0)
       plt.scatter(X[:, 0], X[:, 1], s=50);
```

By eye, it is relatively easy to pick out the four clusters. The *k*-means algorithm does this automatically, and in Scikit-Learn uses the typical estimator API:

```
In[3]: from sklearn.cluster import KMeans
       kmeans = KMeans(n_clusters=4)
       kmeans.fit(X)
       y_kmeans = kmeans.predict(X)
```

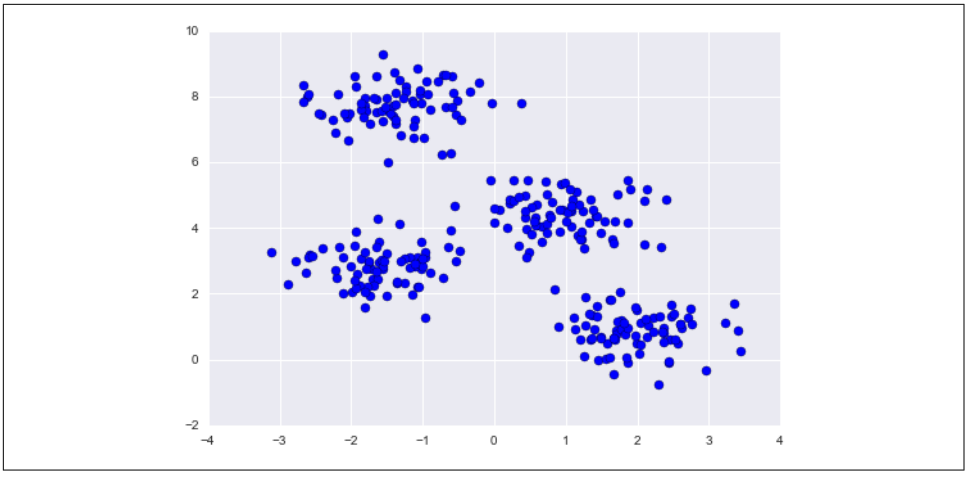

Figure 5-110. Data for demonstration of clustering

Let's visualize the results by plotting the data colored by these labels. We will also plot the cluster centers as determined by the *k*-means estimator (Figure 5-111):

```
In[4]: plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')

       centers = kmeans.cluster_centers_
       plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5);
```

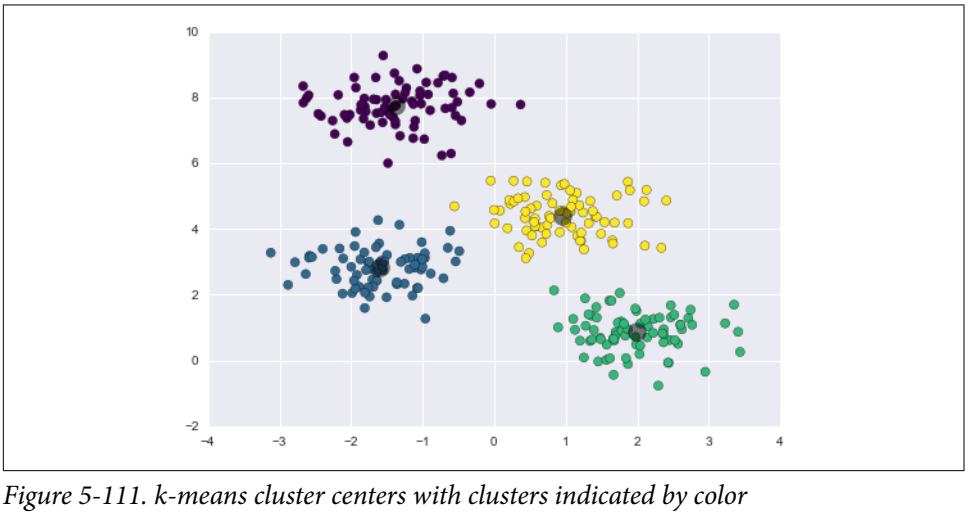

Figure 5-111. k-means cluster centers with clusters indicated by color

The good news is that the *k*-means algorithm (at least in this simple case) assigns the points to clusters very similarly to how we might assign them by eye. But you might wonder how this algorithm finds these clusters so quickly! After all, the number of possible combinations of cluster assignments is exponential in the number of data

points—an exhaustive search would be very, very costly. Fortunately for us, such an exhaustive search is not necessary; instead, the typical approach to *k*-means involves an intuitive iterative approach known as *expectation–maximization*.

k-Means Algorithm: Expectation–Maximization

Expectation–maximization (E–M) is a powerful algorithm that comes up in a variety of contexts within data science. *k*-means is a particularly simple and easy-to-understand application of the algorithm, and we will walk through it briefly here. In short, the expectation–maximization approach consists of the following procedure:

1. Guess some cluster centers
2. Repeat until converged
   a. *E-Step*: assign points to the nearest cluster center
   b. *M-Step*: set the cluster centers to the mean

Here the "E-step" or "Expectation step" is so named because it involves updating our expectation of which cluster each point belongs to. The "M-step" or "Maximization step" is so named because it involves maximizing some fitness function that defines the location of the cluster centers—in this case, that maximization is accomplished by taking a simple mean of the data in each cluster.

The literature about this algorithm is vast, but can be summarized as follows: under typical circumstances, each repetition of the E-step and M-step will always result in a better estimate of the cluster characteristics.

We can visualize the algorithm as shown in Figure 5-112.

For the particular initialization shown here, the clusters converge in just three iterations. For an interactive version of this figure, refer to the code in the online appendix.

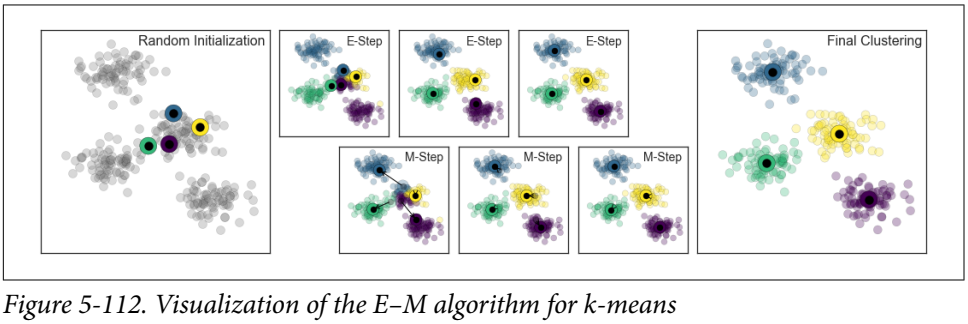

Figure 5-112. Visualization of the E–M algorithm for k-means

The *k*-means algorithm is simple enough that we can write it in a few lines of code. The following is a very basic implementation (Figure 5-113):

```
In[5]: from sklearn.metrics import pairwise_distances_argmin

       def find_clusters(X, n_clusters, rseed=2):
           # 1. Randomly choose clusters
           rng = np.random.RandomState(rseed)
           i = rng.permutation(X.shape[0])[:n_clusters]
           centers = X[i]

           while True:
               # 2a. Assign labels based on closest center
               labels = pairwise_distances_argmin(X, centers)

               # 2b. Find new centers from means of points
               new_centers = np.array([X[labels == i].mean(0)
                                       for i in range(n_clusters)])

               # 2c. Check for convergence
               if np.all(centers == new_centers):
                   break
               centers = new_centers

           return centers, labels

       centers, labels = find_clusters(X, 4)
       plt.scatter(X[:, 0], X[:, 1], c=labels,
                   s=50, cmap='viridis');
```

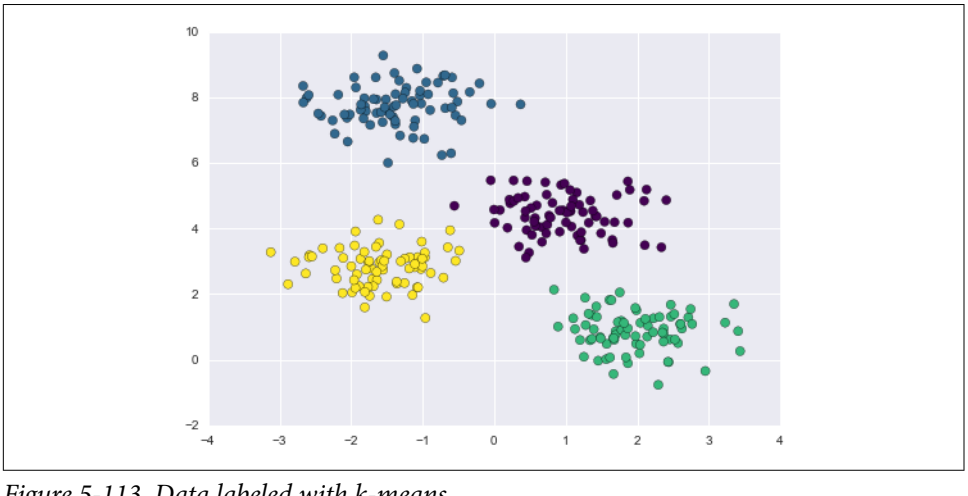

Figure 5-113. Data labeled with k-means

Most well-tested implementations will do a bit more than this under the hood, but the preceding function gives the gist of the expectation–maximization approach.

Caveats of expectation–maximization

There are a few issues to be aware of when using the expectation–maximization algorithm.

The globally optimal result may not be achieved

First, although the E–M procedure is guaranteed to improve the result in each step, there is no assurance that it will lead to the *global* best solution. For example, if we use a different random seed in our simple procedure, the particular starting guesses lead to poor results (Figure 5-114):

```
In[6]: centers, labels = find_clusters(X, 4, rseed=0)
       plt.scatter(X[:, 0], X[:, 1], c=labels,
                   s=50, cmap='viridis');
```

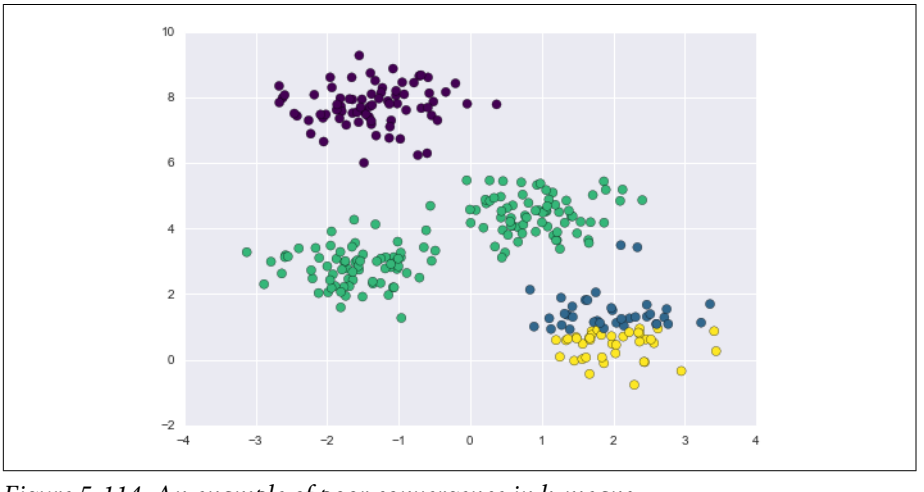

Figure 5-114. An example of poor convergence in k-means

Here the E–M approach has converged, but has not converged to a globally optimal configuration. For this reason, it is common for the algorithm to be run for multiple starting guesses, as indeed Scikit-Learn does by default (set by the `n_init` parameter, which defaults to 10).

The number of clusters must be selected beforehand

Another common challenge with *k*-means is that you must tell it how many clusters you expect: it cannot learn the number of clusters from the data. For example, if we ask the algorithm to identify six clusters, it will happily proceed and find the best six clusters (Figure 5-115):

```
In[7]: labels = KMeans(6, random_state=0).fit_predict(X)
       plt.scatter(X[:, 0], X[:, 1], c=labels,
                   s=50, cmap='viridis');
```

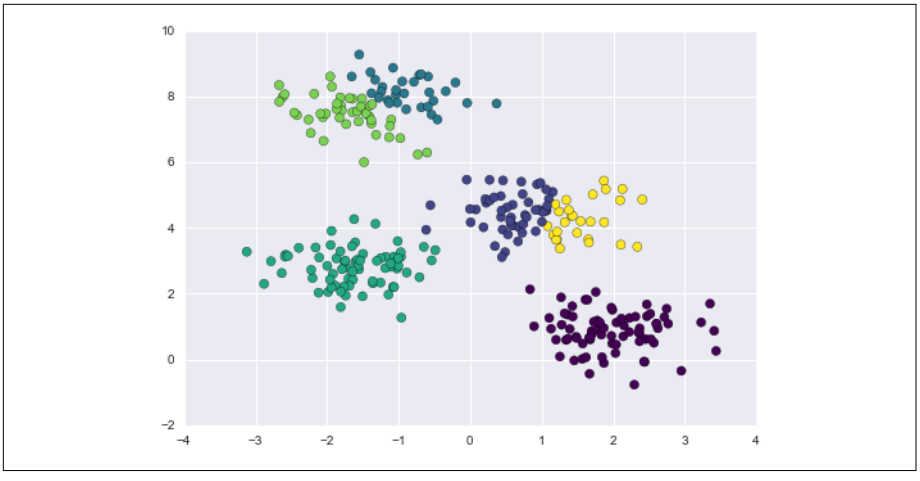

Figure 5-115. An example where the number of clusters is chosen poorly

Whether the result is meaningful is a question that is difficult to answer definitively; one approach that is rather intuitive, but that we won't discuss further here, is called silhouette analysis.

Alternatively, you might use a more complicated clustering algorithm which has a better quantitative measure of the fitness per number of clusters (e.g., Gaussian mixture models; see "In Depth: Gaussian Mixture Models" on page 476) or which *can* choose a suitable number of clusters (e.g., DBSCAN, mean-shift, or affinity propagation, all available in the `sklearn.cluster` submodule).

k-means is limited to linear cluster boundaries

The fundamental model assumptions of *k*-means (points will be closer to their own cluster center than to others) means that the algorithm will often be ineffective if the clusters have complicated geometries.

In particular, the boundaries between *k*-means clusters will always be linear, which means that it will fail for more complicated boundaries. Consider the following data, along with the cluster labels found by the typical *k*-means approach (Figure 5-116):

```
In[8]: from sklearn.datasets import make_moons
       X, y = make_moons(200, noise=.05, random_state=0)

In[9]: labels = KMeans(2, random_state=0).fit_predict(X)
       plt.scatter(X[:, 0], X[:, 1], c=labels,
                   s=50, cmap='viridis');
```

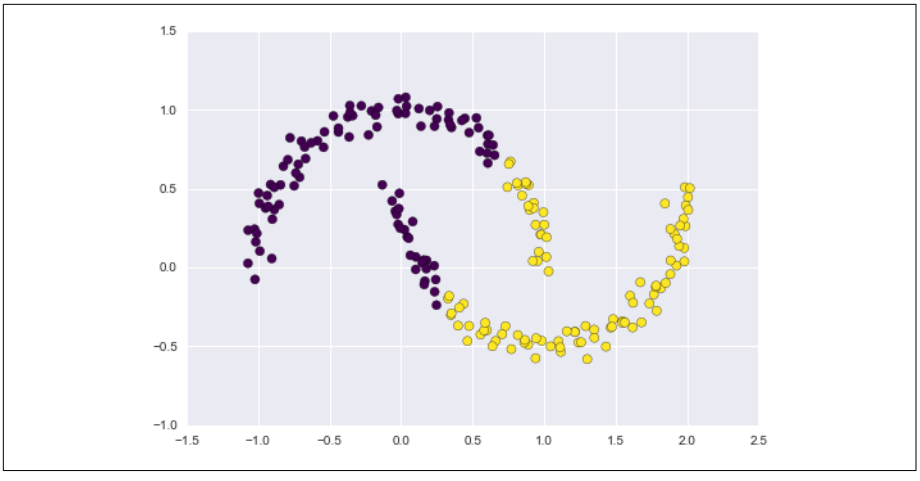

Figure 5-116. Failure of k-means with nonlinear boundaries

This situation is reminiscent of the discussion in "In-Depth: Support Vector Machines" on page 405, where we used a kernel transformation to project the data into a higher dimension where a linear separation is possible. We might imagine using the same trick to allow *k*-means to discover nonlinear boundaries.

One version of this kernelized *k*-means is implemented in Scikit-Learn within the `SpectralClustering` estimator. It uses the graph of nearest neighbors to compute a higher-dimensional representation of the data, and then assigns labels using a *k*-means algorithm (Figure 5-117):

```
In[10]: from sklearn.cluster import SpectralClustering
        model = SpectralClustering(n_clusters=2,
                                   affinity='nearest_neighbors',
                                   assign_labels='kmeans')
        labels = model.fit_predict(X)
        plt.scatter(X[:, 0], X[:, 1], c=labels,
                    s=50, cmap='viridis');
```

We see that with this kernel transform approach, the kernelized *k*-means is able to find the more complicated nonlinear boundaries between clusters.

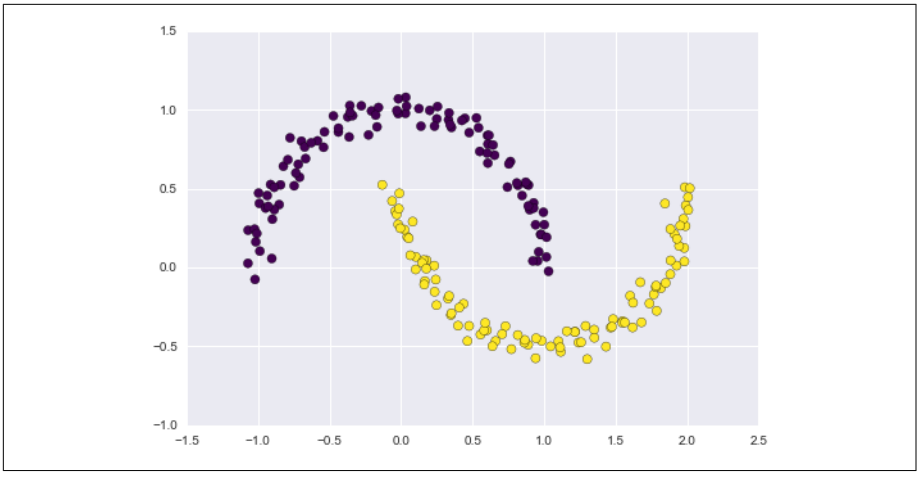

Figure 5-117. Nonlinear boundaries learned by SpectralClustering

k-means can be slow for large numbers of samples

Because each iteration of *k*-means must access every point in the dataset, the algorithm can be relatively slow as the number of samples grows. You might wonder if this requirement to use all data at each iteration can be relaxed; for example, you might just use a subset of the data to update the cluster centers at each step. This is the idea behind batch-based *k*-means algorithms, one form of which is implemented in `sklearn.cluster.MiniBatchKMeans`. The interface for this is the same as for standard `KMeans`; we will see an example of its use as we continue our discussion.

Examples

Being careful about these limitations of the algorithm, we can use *k*-means to our advantage in a wide variety of situations. We'll now take a look at a couple examples.

Example 1: k-Means on digits

To start, let's take a look at applying *k*-means on the same simple digits data that we saw in "In-Depth: Decision Trees and Random Forests" on page 421 and "In Depth: Principal Component Analysis" on page 433. Here we will attempt to use *k*-means to try to identify similar digits *without using the original label information*; this might be similar to a first step in extracting meaning from a new dataset about which you don't have any *a priori* label information.

We will start by loading the digits and then finding the `KMeans` clusters. Recall that the digits consist of 1,797 samples with 64 features, where each of the 64 features is the brightness of one pixel in an 8×8 image:

```
In[11]: from sklearn.datasets import load_digits
        digits = load_digits()
        digits.data.shape
```

```
Out[11]: (1797, 64)
```

The clustering can be performed as we did before:

```
In[12]: kmeans = KMeans(n_clusters=10, random_state=0)
        clusters = kmeans.fit_predict(digits.data)
        kmeans.cluster_centers_.shape
```

```
Out[12]: (10, 64)
```

The result is 10 clusters in 64 dimensions. Notice that the cluster centers themselves are 64-dimensional points, and can themselves be interpreted as the "typical" digit within the cluster. Let's see what these cluster centers look like (Figure 5-118):

```
In[13]: fig, ax = plt.subplots(2, 5, figsize=(8, 3))
        centers = kmeans.cluster_centers_.reshape(10, 8, 8)
        for axi, center in zip(ax.flat, centers):
            axi.set(xticks=[], yticks=[])
            axi.imshow(center, interpolation='nearest', cmap=plt.cm.binary)
```

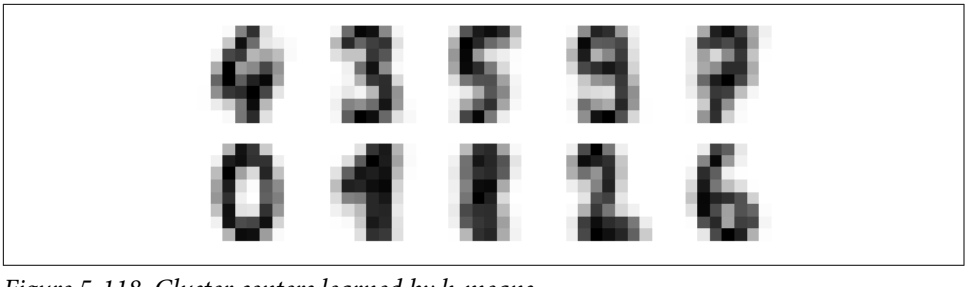

Figure 5-118. Cluster centers learned by k-means

We see that *even without the labels*, `KMeans` is able to find clusters whose centers are recognizable digits, with perhaps the exception of 1 and 8.

Because *k*-means knows nothing about the identity of the cluster, the 0–9 labels may be permuted. We can fix this by matching each learned cluster label with the true labels found in them:

```
In[14]: from scipy.stats import mode

        labels = np.zeros_like(clusters)
        for i in range(10):
            mask = (clusters == i)
            labels[mask] = mode(digits.target[mask])[0]
```

Now we can check how accurate our unsupervised clustering was in finding similar digits within the data:

```
In[15]: from sklearn.metrics import accuracy_score
        accuracy_score(digits.target, labels)
```

```
Out[15]: 0.79354479688369506
```

With just a simple *k*-means algorithm, we discovered the correct grouping for 80% of the input digits! Let's check the confusion matrix for this (Figure 5-119):

```
In[16]: from sklearn.metrics import confusion_matrix
        mat = confusion_matrix(digits.target, labels)
        sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
                    xticklabels=digits.target_names,
                    yticklabels=digits.target_names)
        plt.xlabel('true label')
        plt.ylabel('predicted label');
```

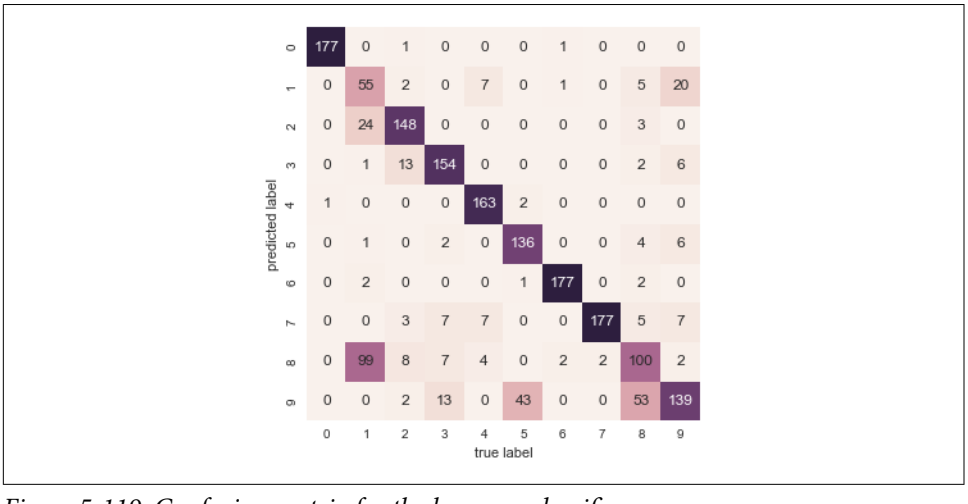

Figure 5-119. Confusion matrix for the k-means classifier

As we might expect from the cluster centers we visualized before, the main point of confusion is between the eights and ones. But this still shows that using *k*-means, we can essentially build a digit classifier *without reference to any known labels*!

Just for fun, let's try to push this even further. We can use the t-distributed stochastic neighbor embedding (t-SNE) algorithm (mentioned in "In-Depth: Manifold Learning" on page 445) to preprocess the data before performing *k*-means. t-SNE is a nonlinear embedding algorithm that is particularly adept at preserving points within clusters. Let's see how it does:

```
In[17]: from sklearn.manifold import TSNE

        # Project the data: this step will take several seconds
        tsne = TSNE(n_components=2, init='pca', random_state=0)
        digits_proj = tsne.fit_transform(digits.data)
```

```
        # Compute the clusters
        kmeans = KMeans(n_clusters=10, random_state=0)
        clusters = kmeans.fit_predict(digits_proj)

        # Permute the labels
        labels = np.zeros_like(clusters)
        for i in range(10):
            mask = (clusters == i)
            labels[mask] = mode(digits.target[mask])[0]

        # Compute the accuracy
        accuracy_score(digits.target, labels)
```

```
Out[17]: 0.93356149137451305
```

That's nearly 94% classification accuracy *without using the labels*. This is the power of unsupervised learning when used carefully: it can extract information from the dataset that it might be difficult to do by hand or by eye.

Example 2: k-means for color compression

One interesting application of clustering is in color compression within images. For example, imagine you have an image with millions of colors. In most images, a large number of the colors will be unused, and many of the pixels in the image will have similar or even identical colors.

For example, consider the image shown in Figure 5-120, which is from Scikit-Learn's `datasets` module (for this to work, you'll have to have the `pillow` Python package installed):

```
In[18]: # Note: this requires the pillow package to be installed
        from sklearn.datasets import load_sample_image
        china = load_sample_image("china.jpg")
        ax = plt.axes(xticks=[], yticks=[])
        ax.imshow(china);
```

The image itself is stored in a three-dimensional array of size `(height, width, RGB)`, containing red/blue/green contributions as integers from 0 to 255:

```
In[19]: china.shape
```

```
Out[19]: (427, 640, 3)
```

Figure 5-120. The input image

One way we can view this set of pixels is as a cloud of points in a three-dimensional color space. We will reshape the data to `[n_samples x n_features]`, and rescale the colors so that they lie between 0 and 1:

```
In[20]: data = china / 255.0 # use 0...1 scale
        data = data.reshape(427 * 640, 3)
        data.shape
```

```
Out[20]: (273280, 3)
```

We can visualize these pixels in this color space, using a subset of 10,000 pixels for efficiency (Figure 5-121):

```
In[21]: def plot_pixels(data, title, colors=None, N=10000):
            if colors is None:
                colors = data

            # choose a random subset
            rng = np.random.RandomState(0)
            i = rng.permutation(data.shape[0])[:N]
            colors = colors[i]
            R, G, B = data[i].T

            fig, ax = plt.subplots(1, 2, figsize=(16, 6))
            ax[0].scatter(R, G, color=colors, marker='.')
            ax[0].set(xlabel='Red', ylabel='Green', xlim=(0, 1), ylim=(0, 1))

            ax[1].scatter(R, B, color=colors, marker='.')
            ax[1].set(xlabel='Red', ylabel='Blue', xlim=(0, 1), ylim=(0, 1))

            fig.suptitle(title, size=20);
```

```
In[22]: plot_pixels(data, title='Input color space: 16 million possible colors')
```

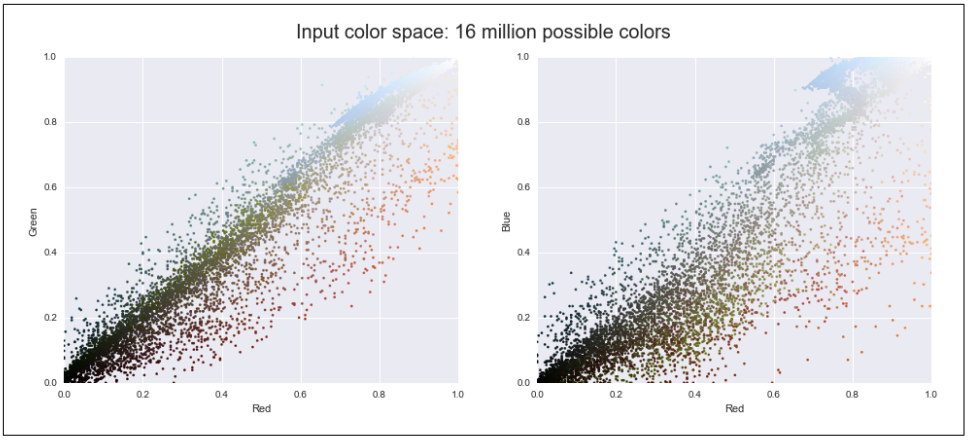

Figure 5-121. The distribution of the pixels in RGB color space

Now let's reduce these 16 million colors to just 16 colors, using a *k*-means clustering across the pixel space. Because we are dealing with a very large dataset, we will use the mini batch *k*-means, which operates on subsets of the data to compute the result much more quickly than the standard *k*-means algorithm (Figure 5-122):

```
In[23]: from sklearn.cluster import MiniBatchKMeans
        kmeans = MiniBatchKMeans(16)
        kmeans.fit(data)
        new_colors = kmeans.cluster_centers_[kmeans.predict(data)]

        plot_pixels(data, colors=new_colors,
                    title="Reduced color space: 16 colors")
```

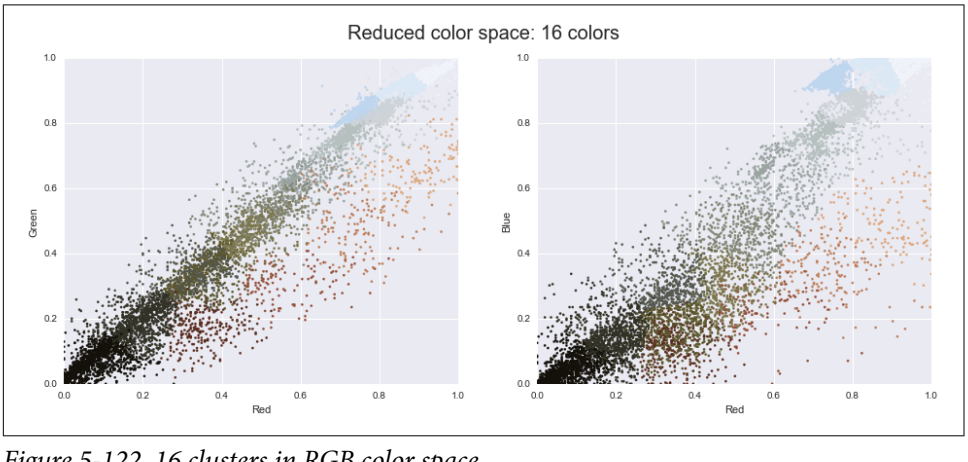

Figure 5-122. 16 clusters in RGB color space

The result is a recoloring of the original pixels, where each pixel is assigned the color of its closest cluster center. Plotting these new colors in the image space rather than the pixel space shows us the effect of this (Figure 5-123):

```
In[24]:
china_recolored = new_colors.reshape(china.shape)

fig, ax = plt.subplots(1, 2, figsize=(16, 6),
          subplot_kw=dict(xticks=[], yticks=[]))
fig.subplots_adjust(wspace=0.05)
ax[0].imshow(china)
ax[0].set_title('Original Image', size=16)
ax[1].imshow(china_recolored)
ax[1].set_title('16-color Image', size=16);
```

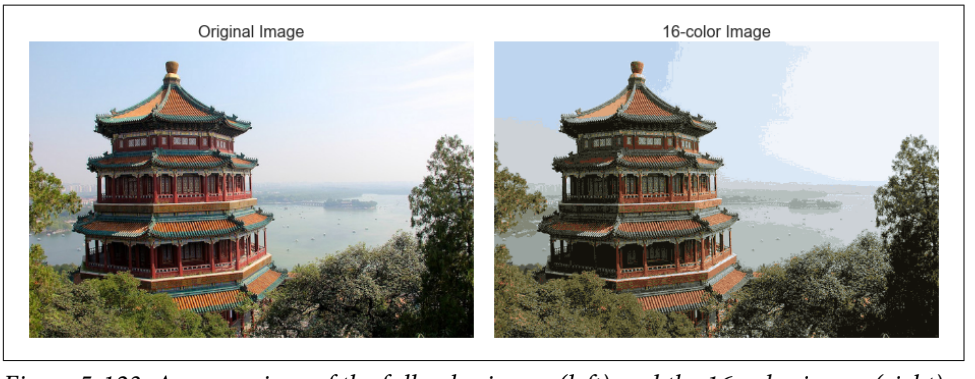

Figure 5-123. A comparison of the full-color image (left) and the 16-color image (right)

Some detail is certainly lost in the rightmost panel, but the overall image is still easily recognizable. This image on the right achieves a compression factor of around 1 million! While this is an interesting application of *k*-means, there are certainly better way to compress information in images. But the example shows the power of thinking outside of the box with unsupervised methods like *k*-means.

In Depth: Gaussian Mixture Models

The *k*-means clustering model explored in the previous section is simple and relatively easy to understand, but its simplicity leads to practical challenges in its application. In particular, the nonprobabilistic nature of *k*-means and its use of simple distance-from-cluster-center to assign cluster membership leads to poor performance for many real-world situations. In this section we will take a look at Gaussian mixture models, which can be viewed as an extension of the ideas behind *k*-means, but can also be a powerful tool for estimation beyond simple clustering. We begin with the standard imports:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
       import numpy as np
```

Motivating GMM: Weaknesses of k-Means

Let's take a look at some of the weaknesses of *k*-means and think about how we might improve the cluster model. As we saw in the previous section, given simple, well-separated data, *k*-means finds suitable clustering results.

For example, if we have simple blobs of data, the *k*-means algorithm can quickly label those clusters in a way that closely matches what we might do by eye (Figure 5-124):

```
In[2]: # Generate some data
       from sklearn.datasets.samples_generator import make_blobs
       X, y_true = make_blobs(n_samples=400, centers=4,
                              cluster_std=0.60, random_state=0)
       X = X[:, ::-1] # flip axes for better plotting
```

```
In[3]: # Plot the data with k-means labels
       from sklearn.cluster import KMeans
       kmeans = KMeans(4, random_state=0)
       labels = kmeans.fit(X).predict(X)
       plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis');
```

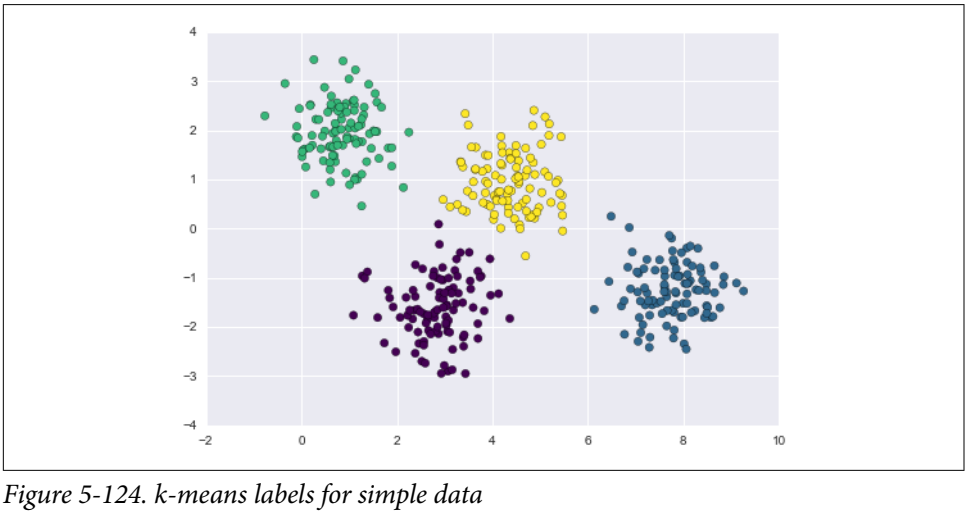

Figure 5-124. k-means labels for simple data

From an intuitive standpoint, we might expect that the clustering assignment for some points is more certain than others; for example, there appears to be a very slight overlap between the two middle clusters, such that we might not have complete confidence in the cluster assignment of points between them. Unfortunately, the *k*-means model has no intrinsic measure of probability or uncertainty of cluster assignments

(although it may be possible to use a bootstrap approach to estimate this uncertainty). For this, we must think about generalizing the model.

One way to think about the *k*-means model is that it places a circle (or, in higher dimensions, a hyper-sphere) at the center of each cluster, with a radius defined by the most distant point in the cluster. This radius acts as a hard cutoff for cluster assignment within the training set: any point outside this circle is not considered a member of the cluster. We can visualize this cluster model with the following function (Figure 5-125):

```
In[4]:
from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist

def plot_kmeans(kmeans, X, n_clusters=4, rseed=0, ax=None):
    labels = kmeans.fit_predict(X)

    # plot the input data
    ax = ax or plt.gca()
    ax.axis('equal')
    ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis', zorder=2)

    # plot the representation of the k-means model
    centers = kmeans.cluster_centers_
    radii = [cdist(X[labels == i], [center]).max()
             for i, center in enumerate(centers)]
    for c, r in zip(centers, radii):
        ax.add_patch(plt.Circle(c, r, fc='#CCCCCC', lw=3, alpha=0.5, zorder=1))
```

```
In[5]: kmeans = KMeans(n_clusters=4, random_state=0)
       plot_kmeans(kmeans, X)
```

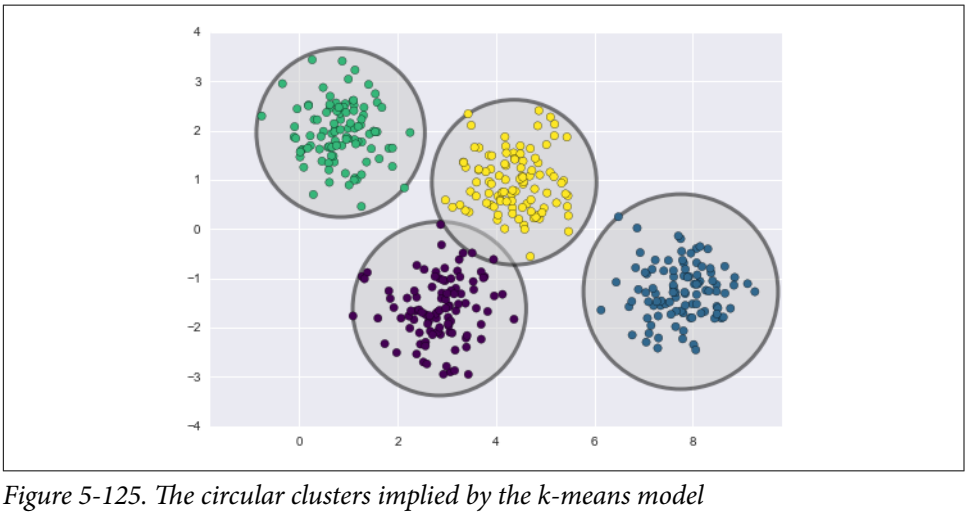

Figure 5-125. The circular clusters implied by the k-means model

An important observation for *k*-means is that these cluster models *must be circular*: *k*-means has no built-in way of accounting for oblong or elliptical clusters. So, for example, if we take the same data and transform it, the cluster assignments end up becoming muddled (Figure 5-126):

```
In[6]: rng = np.random.RandomState(13)
       X_stretched = np.dot(X, rng.randn(2, 2))

       kmeans = KMeans(n_clusters=4, random_state=0)
       plot_kmeans(kmeans, X_stretched)
```

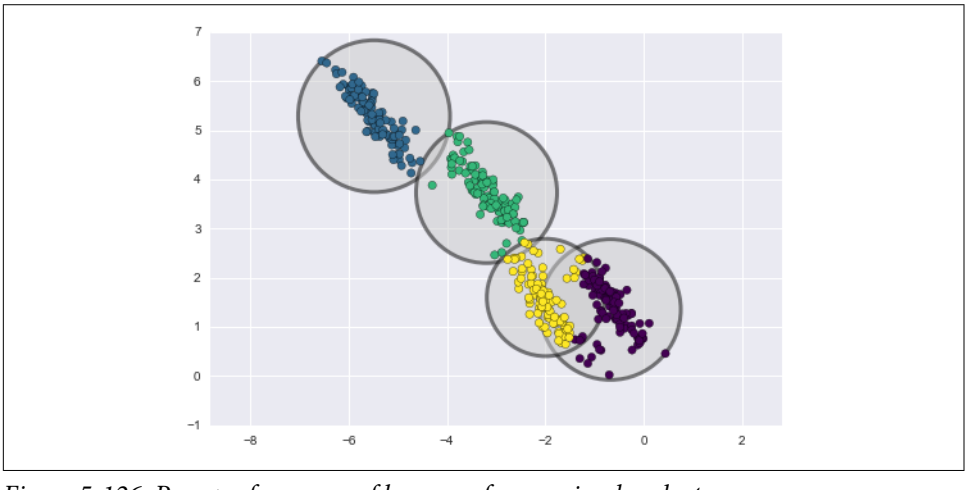

Figure 5-126. Poor performance of k-means for noncircular clusters

By eye, we recognize that these transformed clusters are noncircular, and thus circular clusters would be a poor fit. Nevertheless, *k*-means is not flexible enough to account for this, and tries to force-fit the data into four circular clusters. This results in a mixing of cluster assignments where the resulting circles overlap: see especially the bottom right of this plot. One might imagine addressing this particular situation by preprocessing the data with PCA (see "In Depth: Principal Component Analysis" on page 433), but in practice there is no guarantee that such a global operation will circularize the individual data.

These two disadvantages of *k*-means—its lack of flexibility in cluster shape and lack of probabilistic cluster assignment—mean that for many datasets (especially low-dimensional datasets) it may not perform as well as you might hope.

You might imagine addressing these weaknesses by generalizing the *k*-means model: for example, you could measure uncertainty in cluster assignment by comparing the distances of each point to *all* cluster centers, rather than focusing on just the closest. You might also imagine allowing the cluster boundaries to be ellipses rather than cir-

cles, so as to account for noncircular clusters. It turns out these are two essential components of a different type of clustering model, Gaussian mixture models.

Generalizing E–M: Gaussian Mixture Models

A Gaussian mixture model (GMM) attempts to find a mixture of multidimensional Gaussian probability distributions that best model any input dataset. In the simplest case, GMMs can be used for finding clusters in the same manner as *k*-means (Figure 5-127):

```
In[7]: from sklearn.mixture import GMM
       gmm = GMM(n_components=4).fit(X)
       labels = gmm.predict(X)
       plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis');
```

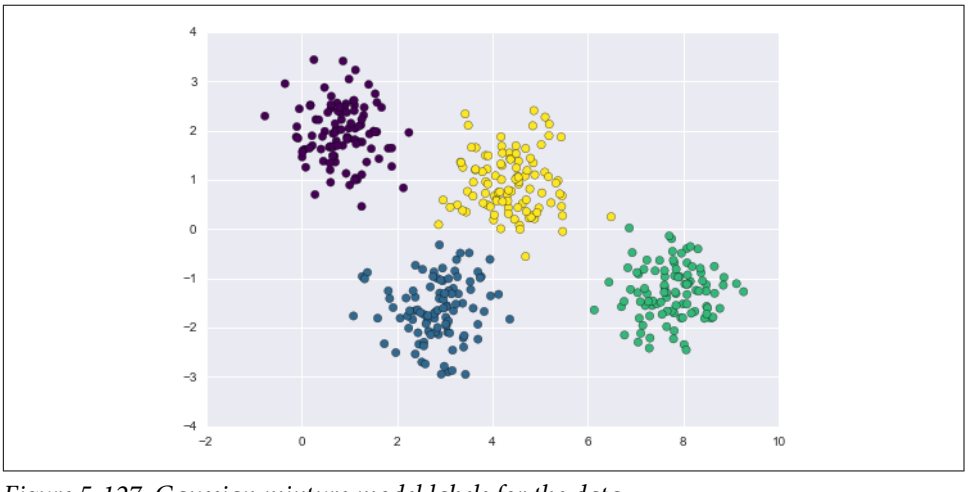

Figure 5-127. Gaussian mixture model labels for the data

But because GMM contains a probabilistic model under the hood, it is also possible to find probabilistic cluster assignments—in Scikit-Learn we do this using the `predict_proba` method. This returns a matrix of size `[n_samples, n_clusters]` that measures the probability that any point belongs to the given cluster:

```
In[8]: probs = gmm.predict_proba(X)
       print(probs[:5].round(3))
```

```
[[ 0.     0.     0.475  0.525]
 [ 0.     1.     0.     0.   ]
 [ 0.     1.     0.     0.   ]
 [ 0.     0.     0.     1.   ]
 [ 0.     1.     0.     0.   ]]
```

We can visualize this uncertainty by, for example, making the size of each point proportional to the certainty of its prediction; looking at Figure 5-128, we can see that it

is precisely the points at the boundaries between clusters that reflect this uncertainty of cluster assignment:

```
In[9]: size = 50 * probs.max(1) ** 2  # square emphasizes differences
       plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=size);
```

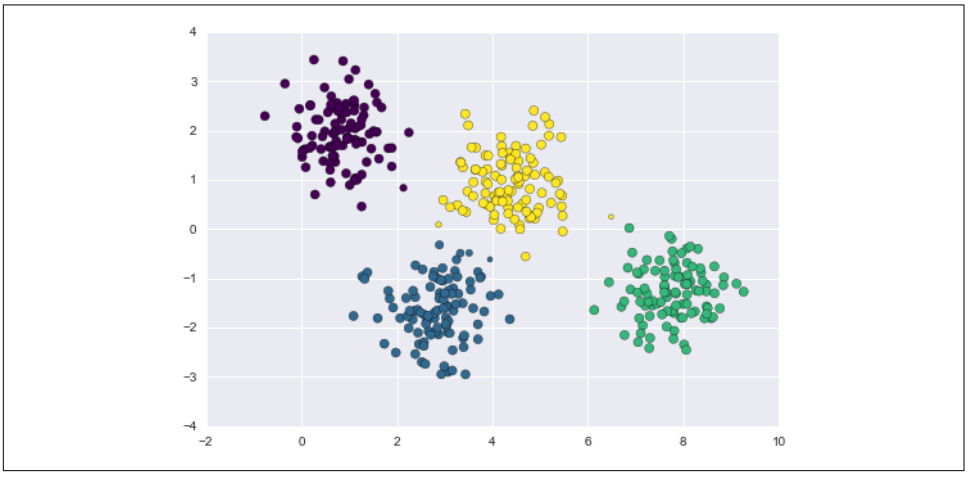

Figure 5-128. GMM probablistic labels: probabilities are shown by the size of points

Under the hood, a Gaussian mixture model is very similar to *k*-means: it uses an expectation–maximization approach that qualitatively does the following:

1. Choose starting guesses for the location and shape
2. Repeat until converged:
   a. *E-step*: for each point, find weights encoding the probability of membership in each cluster
   b. *M-step*: for each cluster, update its location, normalization, and shape based on *all* data points, making use of the weights

The result of this is that each cluster is associated not with a hard-edged sphere, but with a smooth Gaussian model. Just as in the *k*-means expectation–maximization approach, this algorithm can sometimes miss the globally optimal solution, and thus in practice multiple random initializations are used.

Let's create a function that will help us visualize the locations and shapes of the GMM clusters by drawing ellipses based on the `gmm` output:

```
In[10]:
from matplotlib.patches import Ellipse

def draw_ellipse(position, covariance, ax=None, **kwargs):
    """Draw an ellipse with a given position and covariance"""
```

```
        ax = ax or plt.gca()

        # Convert covariance to principal axes
        if covariance.shape == (2, 2):
            U, s, Vt = np.linalg.svd(covariance)
            angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))
            width, height = 2 * np.sqrt(s)
        else:
            angle = 0
            width, height = 2 * np.sqrt(covariance)

        # Draw the ellipse
        for nsig in range(1, 4):
            ax.add_patch(Ellipse(position, nsig * width, nsig * height,
                                 angle, **kwargs))

    def plot_gmm(gmm, X, label=True, ax=None):
        ax = ax or plt.gca()
        labels = gmm.fit(X).predict(X)
        if label:
            ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis', zorder=2)
        else:
            ax.scatter(X[:, 0], X[:, 1], s=40, zorder=2)
        ax.axis('equal')

        w_factor = 0.2 / gmm.weights_.max()
        for pos, covar, w in zip(gmm.means_, gmm.covars_, gmm.weights_):
            draw_ellipse(pos, covar, alpha=w * w_factor)
```

With this in place, we can take a look at what the four-component GMM gives us for our initial data (Figure 5-129):

```
In[11]: gmm = GMM(n_components=4, random_state=42)
        plot_gmm(gmm, X)
```

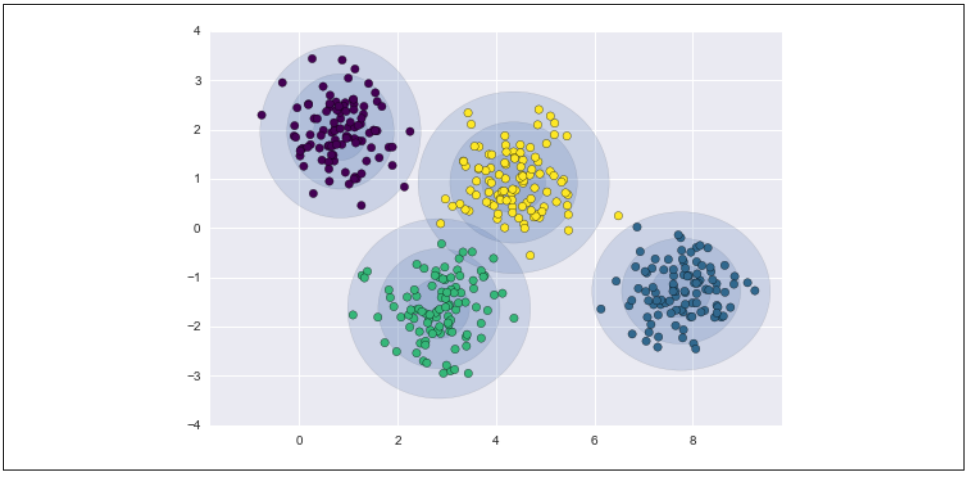

Figure 5-129. Representation of the four-component GMM in the presence of circular clusters

Similarly, we can use the GMM approach to fit our stretched dataset; allowing for a full covariance, the model will fit even very oblong, stretched-out clusters (Figure 5-130):

```
In[12]: gmm = GMM(n_components=4, covariance_type='full', random_state=42)
        plot_gmm(gmm, X_stretched)
```

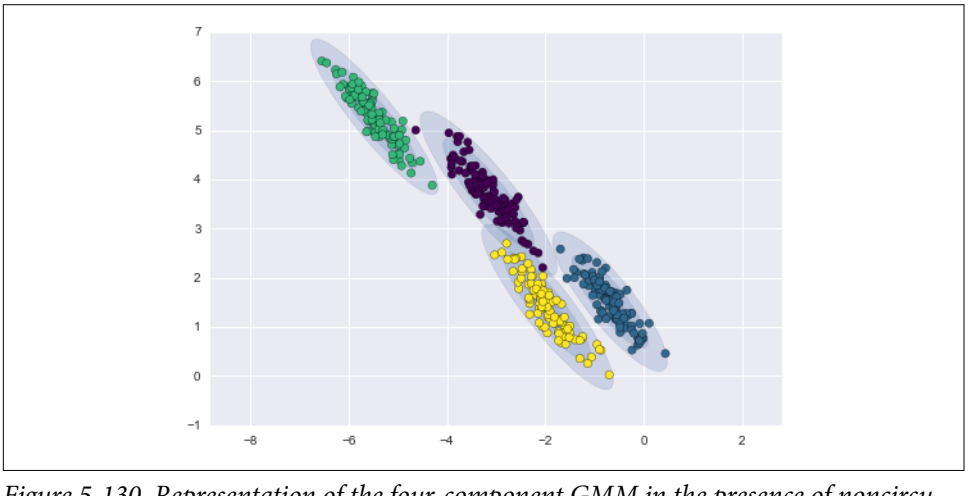

Figure 5-130. Representation of the four-component GMM in the presence of noncircular clusters

This makes clear that GMMs address the two main practical issues with *k*-means encountered before.

Choosing the covariance type

If you look at the details of the preceding fits, you will see that the `covariance_type` option was set differently within each. This hyperparameter controls the degrees of freedom in the shape of each cluster; it is essential to set this carefully for any given problem. The default is `covariance_type="diag"`, which means that the size of the cluster along each dimension can be set independently, with the resulting ellipse constrained to align with the axes. A slightly simpler and faster model is `covariance_type="spherical"`, which constrains the shape of the cluster such that all dimensions are equal. The resulting clustering will have similar characteristics to that of *k*-means, though it is not entirely equivalent. A more complicated and computationally expensive model (especially as the number of dimensions grows) is to use `covariance_type="full"`, which allows each cluster to be modeled as an ellipse with arbitrary orientation.

We can see a visual representation of these three choices for a single cluster within Figure 5-131:

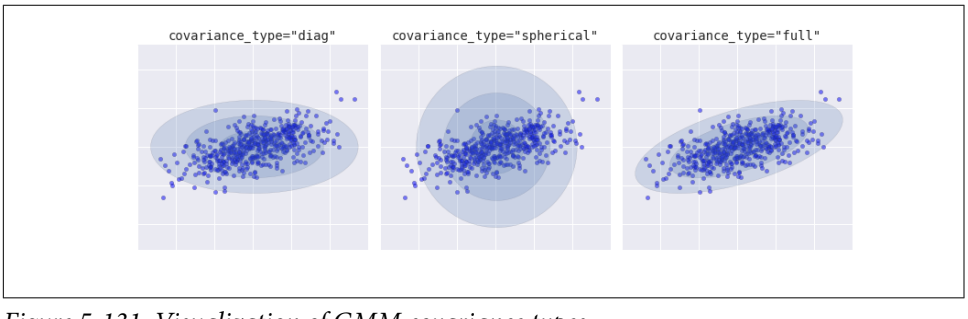

Figure 5-131. Visualization of GMM covariance types

GMM as Density Estimation

Though GMM is often categorized as a clustering algorithm, fundamentally it is an algorithm for *density estimation*. That is to say, the result of a GMM fit to some data is technically not a clustering model, but a generative probabilistic model describing the distribution of the data.

As an example, consider some data generated from Scikit-Learn's `make_moons` function (visualized in Figure 5-132), which we saw in "In Depth: k-Means Clustering" on page 462:

```
In[13]: from sklearn.datasets import make_moons
        Xmoon, ymoon = make_moons(200, noise=.05, random_state=0)
        plt.scatter(Xmoon[:, 0], Xmoon[:, 1]);
```

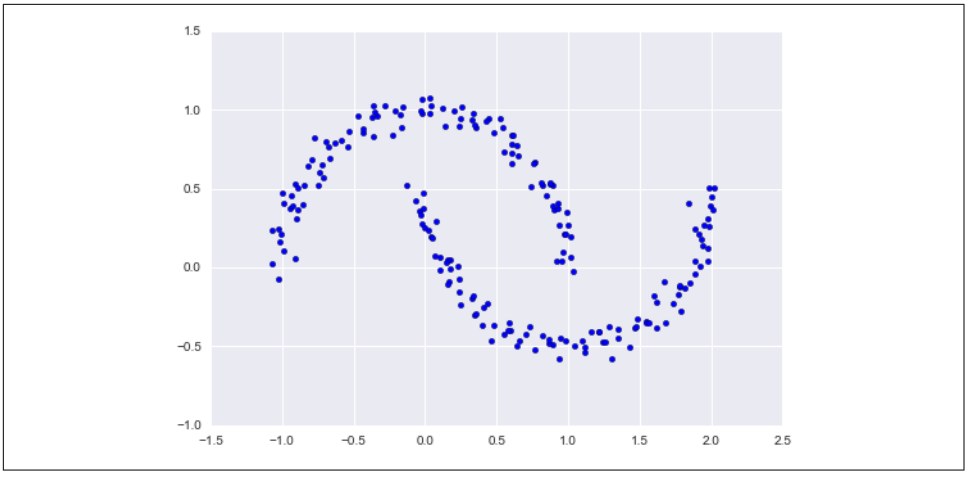

Figure 5-132. GMM applied to clusters with nonlinear boundaries

If we try to fit this to a two-component GMM viewed as a clustering model, the results are not particularly useful (Figure 5-133):

```
In[14]: gmm2 = GMM(n_components=2, covariance_type='full', random_state=0)
        plot_gmm(gmm2, Xmoon)
```

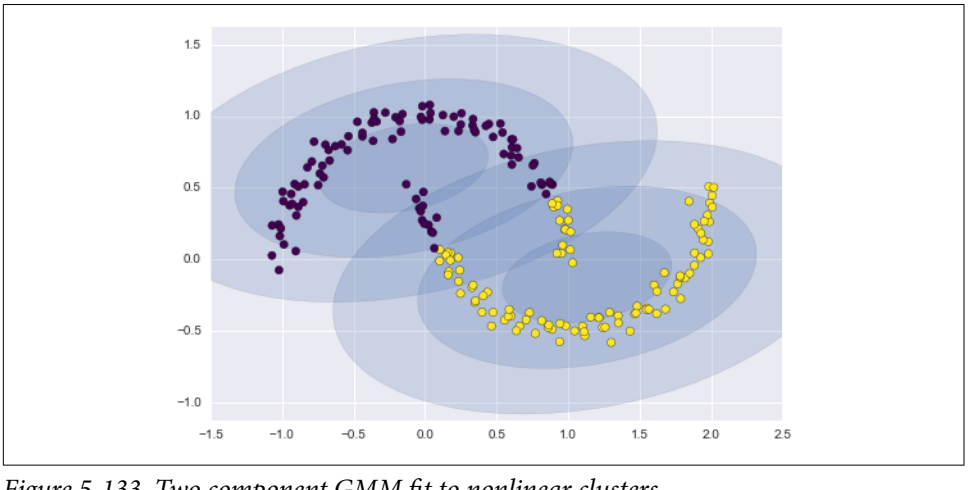

Figure 5-133. Two component GMM fit to nonlinear clusters

But if we instead use many more components and ignore the cluster labels, we find a fit that is much closer to the input data (Figure 5-134):

```
In[15]: gmm16 = GMM(n_components=16, covariance_type='full', random_state=0)
        plot_gmm(gmm16, Xmoon, label=False)
```

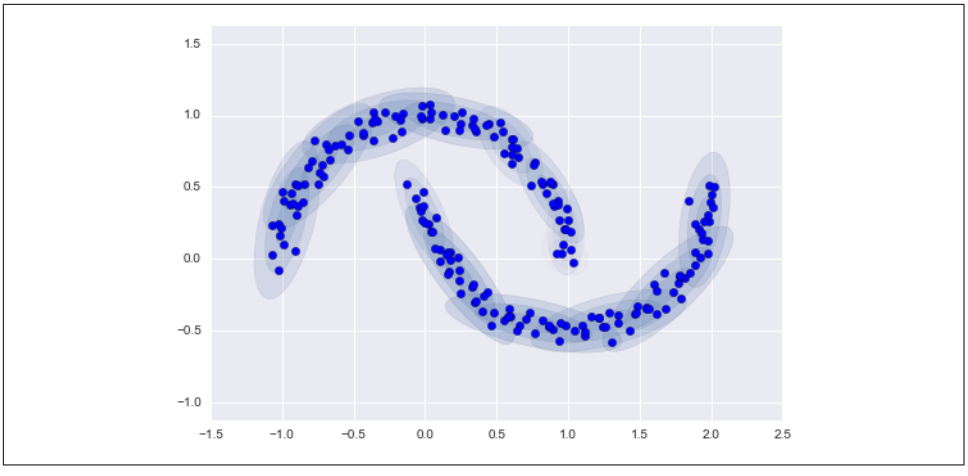

Figure 5-134. Using many GMM clusters to model the distribution of points

Here the mixture of 16 Gaussians serves not to find separated clusters of data, but rather to model the overall *distribution* of the input data. This is a generative model of the distribution, meaning that the GMM gives us the recipe to generate new random data distributed similarly to our input. For example, here are 400 new points drawn from this 16-component GMM fit to our original data (Figure 5-135):

```
In[16]: Xnew = gmm16.sample(400, random_state=42)
        plt.scatter(Xnew[:, 0], Xnew[:, 1]);
```

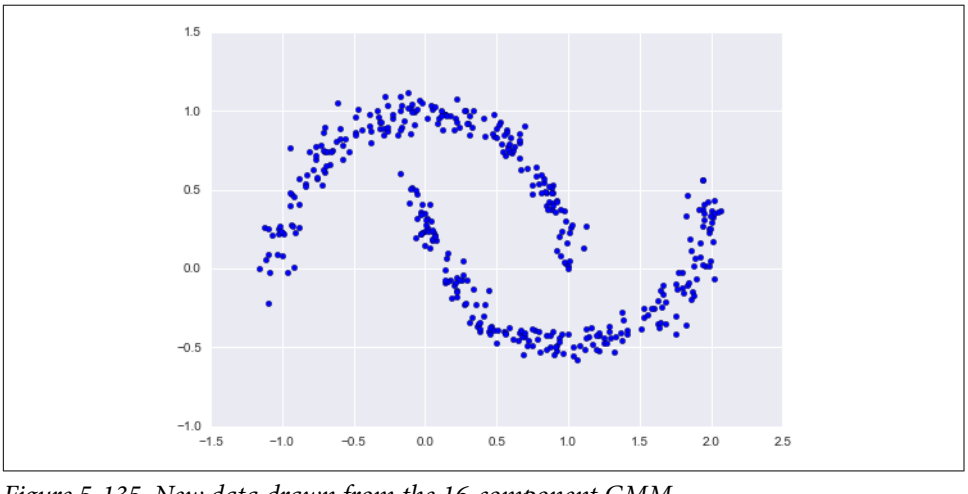

Figure 5-135. New data drawn from the 16-component GMM

GMM is convenient as a flexible means of modeling an arbitrary multidimensional distribution of data.

How many components?

The fact that GMM is a generative model gives us a natural means of determining the optimal number of components for a given dataset. A generative model is inherently a probability distribution for the dataset, and so we can simply evaluate the *likelihood* of the data under the model, using cross-validation to avoid overfitting. Another means of correcting for overfitting is to adjust the model likelihoods using some analytic criterion such as the Akaike information criterion (AIC) or the Bayesian information criterion (BIC). Scikit-Learn's GMM estimator actually includes built-in methods that compute both of these, and so it is very easy to operate on this approach.

Let's look at the AIC and BIC as a function as the number of GMM components for our moon dataset (Figure 5-136):

```
In[17]: n_components = np.arange(1, 21)
        models = [GMM(n, covariance_type='full', random_state=0).fit(Xmoon)
                  for n in n_components]

        plt.plot(n_components, [m.bic(Xmoon) for m in models], label='BIC')
        plt.plot(n_components, [m.aic(Xmoon) for m in models], label='AIC')
        plt.legend(loc='best')
        plt.xlabel('n_components');
```

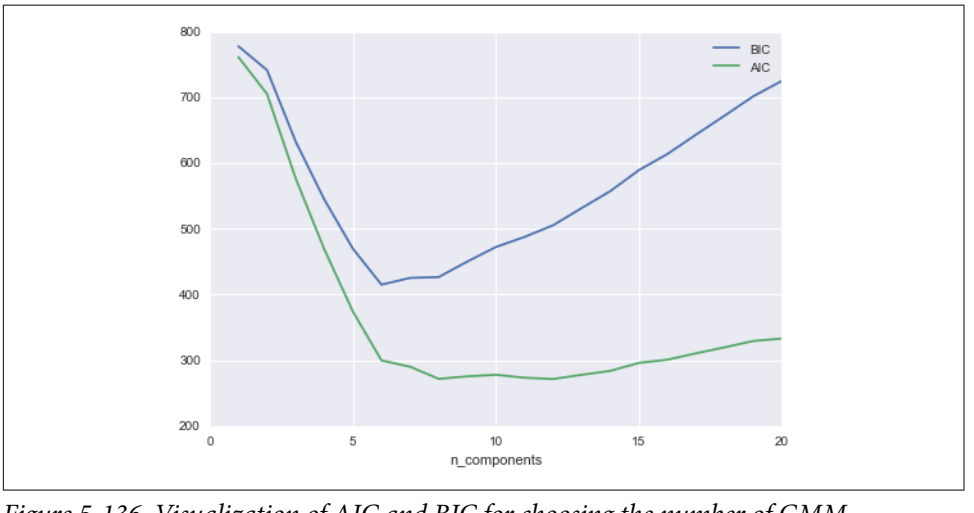

Figure 5-136. Visualization of AIC and BIC for choosing the number of GMM components

The optimal number of clusters is the value that minimizes the AIC or BIC, depending on which approximation we wish to use. The AIC tells us that our choice of 16 components was probably too many: around 8–12 components would have been a

better choice. As is typical with this sort of problem, the BIC recommends a simpler model.

Notice the important point: this choice of number of components measures how well GMM works *as a density estimator*, not how well it works *as a clustering algorithm*. I'd encourage you to think of GMM primarily as a density estimator, and use it for clustering only when warranted within simple datasets.

Example: GMM for Generating New Data

We just saw a simple example of using GMM as a generative model of data in order to create new samples from the distribution defined by the input data. Here we will run with this idea and generate *new handwritten digits* from the standard digits corpus that we have used before.

To start with, let's load the digits data using Scikit-Learn's data tools:

```
In[18]: from sklearn.datasets import load_digits
        digits = load_digits()
        digits.data.shape
```

```
Out[18]: (1797, 64)
```

Next let's plot the first 100 of these to recall exactly what we're looking at (Figure 5-137):

```
In[19]: def plot_digits(data):
            fig, ax = plt.subplots(10, 10, figsize=(8, 8),
                                   subplot_kw=dict(xticks=[], yticks=[]))
            fig.subplots_adjust(hspace=0.05, wspace=0.05)
            for i, axi in enumerate(ax.flat):
                im = axi.imshow(data[i].reshape(8, 8), cmap='binary')
                im.set_clim(0, 16)
        plot_digits(digits.data)
```

We have nearly 1,800 digits in 64 dimensions, and we can build a GMM on top of these to generate more. GMMs can have difficulty converging in such a high dimensional space, so we will start with an invertible dimensionality reduction algorithm on the data. Here we will use a straightforward PCA, asking it to preserve 99% of the variance in the projected data:

```
In[20]: from sklearn.decomposition import PCA
        pca = PCA(0.99, whiten=True)
        data = pca.fit_transform(digits.data)
        data.shape
```

```
Out[20]: (1797, 41)
```

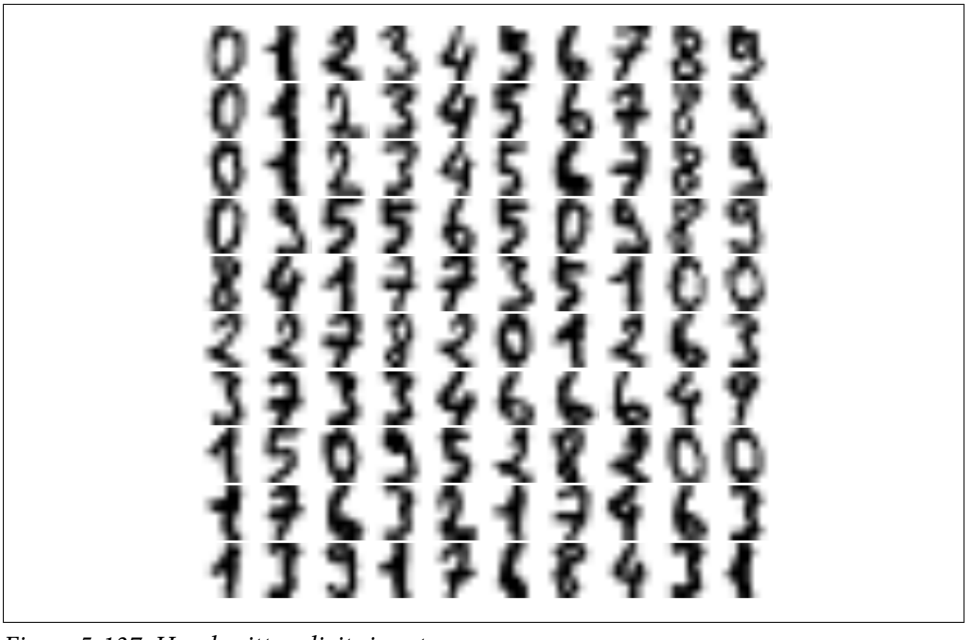

Figure 5-137. Handwritten digits input

The result is 41 dimensions, a reduction of nearly 1/3 with almost no information loss. Given this projected data, let's use the AIC to get a gauge for the number of GMM components we should use (Figure 5-138):

```
In[21]: n_components = np.arange(50, 210, 10)
        models = [GMM(n, covariance_type='full', random_state=0)
                  for n in n_components]
        aics = [model.fit(data).aic(data) for model in models]
        plt.plot(n_components, aics);
```

It appears that around 110 components minimizes the AIC; we will use this model. Let's quickly fit this to the data and confirm that it has converged:

```
In[22]: gmm = GMM(110, covariance_type='full', random_state=0)
        gmm.fit(data)
        print(gmm.converged_)
```

```
True
```

Now we can draw samples of 100 new points within this 41-dimensional projected space, using the GMM as a generative model:

```
In[23]: data_new = gmm.sample(100, random_state=0)
        data_new.shape
```

```
Out[23]: (100, 41)
```

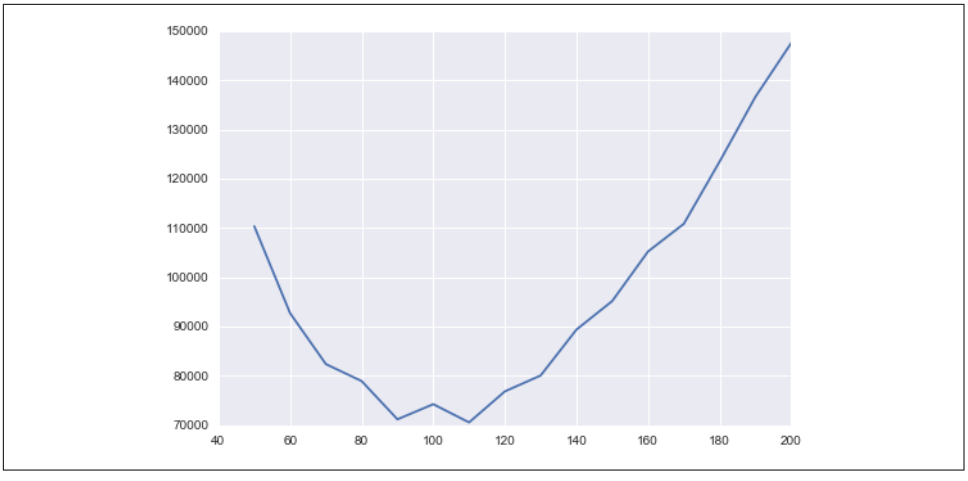

Figure 5-138. AIC curve for choosing the appropriate number of GMM components

Finally, we can use the inverse transform of the PCA object to construct the new digits (Figure 5-139):

```
In[24]: digits_new = pca.inverse_transform(data_new)
        plot_digits(digits_new)
```

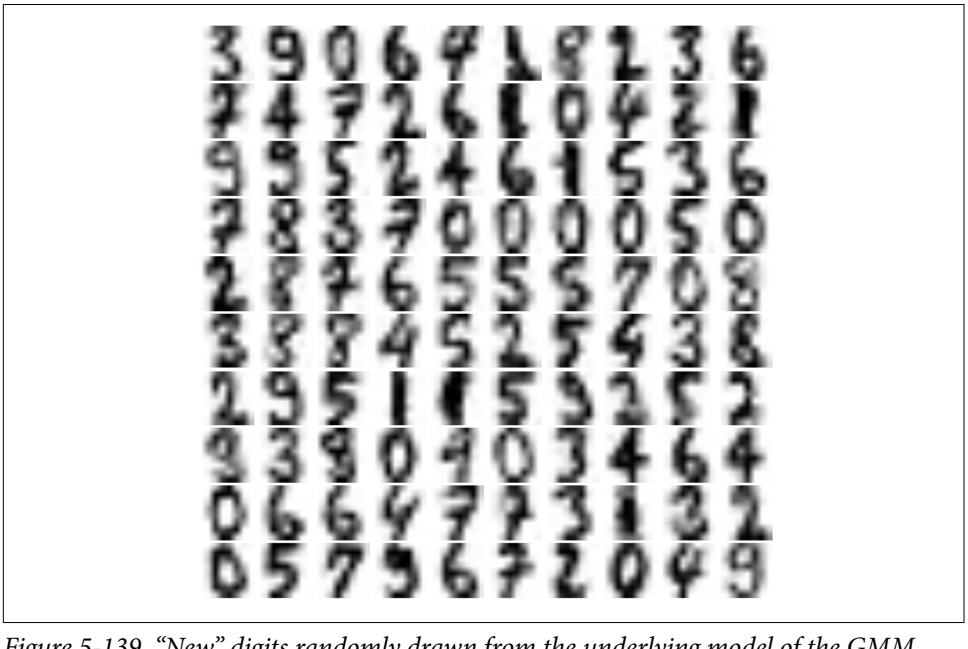

Figure 5-139. "New" digits randomly drawn from the underlying model of the GMM estimator

The results for the most part look like plausible digits from the dataset!

Consider what we've done here: given a sampling of handwritten digits, we have modeled the distribution of that data in such a way that we can generate brand new samples of digits from the data: these are "handwritten digits" that do not individually appear in the original dataset, but rather capture the general features of the input data as modeled by the mixture model. Such a generative model of digits can prove very useful as a component of a Bayesian generative classifier, as we shall see in the next section.

In-Depth: Kernel Density Estimation

In the previous section we covered Gaussian mixture models (GMM), which are a kind of hybrid between a clustering estimator and a density estimator. Recall that a density estimator is an algorithm that takes a *D*-dimensional dataset and produces an estimate of the *D*-dimensional probability distribution which that data is drawn from. The GMM algorithm accomplishes this by representing the density as a weighted sum of Gaussian distributions. *Kernel density estimation* (KDE) is in some senses an algorithm that takes the mixture-of-Gaussians idea to its logical extreme: it uses a mixture consisting of one Gaussian component *per point*, resulting in an essentially nonparametric estimator of density. In this section, we will explore the motivation and uses of KDE. We begin with the standard imports:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
       import numpy as np
```

Motivating KDE: Histograms

As already discussed, a density estimator is an algorithm that seeks to model the probability distribution that generated a dataset. For one-dimensional data, you are probably already familiar with one simple density estimator: the histogram. A histogram divides the data into discrete bins, counts the number of points that fall in each bin, and then visualizes the results in an intuitive manner.

For example, let's create some data that is drawn from two normal distributions:

```
In[2]: def make_data(N, f=0.3, rseed=1):
           rand = np.random.RandomState(rseed)
           x = rand.randn(N)
           x[int(f * N):] += 5
           return x

       x = make_data(1000)
```

We have previously seen that the standard count-based histogram can be created with the `plt.hist()` function. By specifying the `normed` parameter of the histogram, we end up with a normalized histogram where the height of the bins does not reflect counts, but instead reflects probability density (Figure 5-140):

```
In[3]: hist = plt.hist(x, bins=30, normed=True)
```

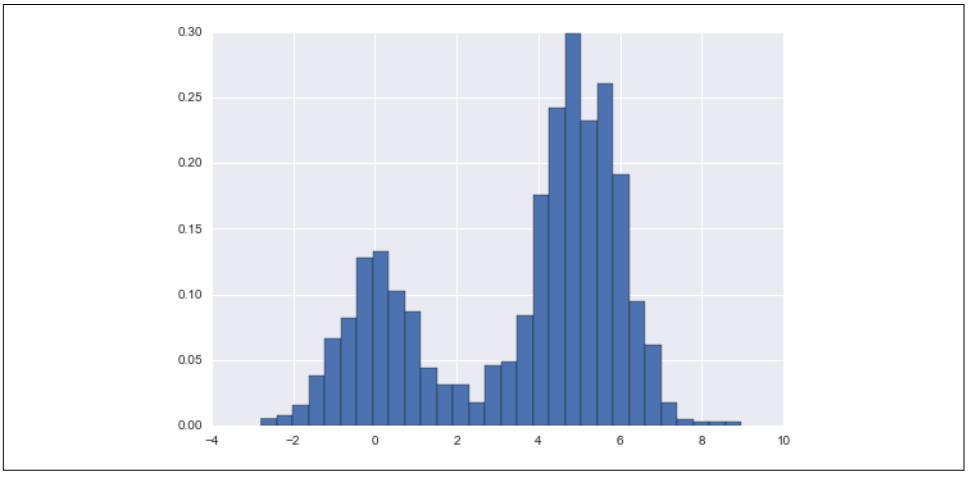

Figure 5-140. Data drawn from a combination of normal distributions

Notice that for equal binning, this normalization simply changes the scale on the y-axis, leaving the relative heights essentially the same as in a histogram built from counts. This normalization is chosen so that the total area under the histogram is equal to 1, as we can confirm by looking at the output of the histogram function:

```
In[4]: density, bins, patches = hist
       widths = bins[1:] - bins[:-1]
       (density * widths).sum()

Out[4]: 1.0
```

One of the issues with using a histogram as a density estimator is that the choice of bin size and location can lead to representations that have qualitatively different features. For example, if we look at a version of this data with only 20 points, the choice of how to draw the bins can lead to an entirely different interpretation of the data! Consider this example (visualized in Figure 5-141):

```
In[5]: x = make_data(20)
       bins = np.linspace(-5, 10, 10)

In[6]: fig, ax = plt.subplots(1, 2, figsize=(12, 4),
                              sharex=True, sharey=True,
                              subplot_kw={'xlim':(-4, 9),
                                          'ylim':(-0.02, 0.3)})
       fig.subplots_adjust(wspace=0.05)
```

```
    for i, offset in enumerate([0.0, 0.6]):
        ax[i].hist(x, bins=bins + offset, normed=True)
        ax[i].plot(x, np.full_like(x, -0.01), '|k',
                   markeredgewidth=1)
```

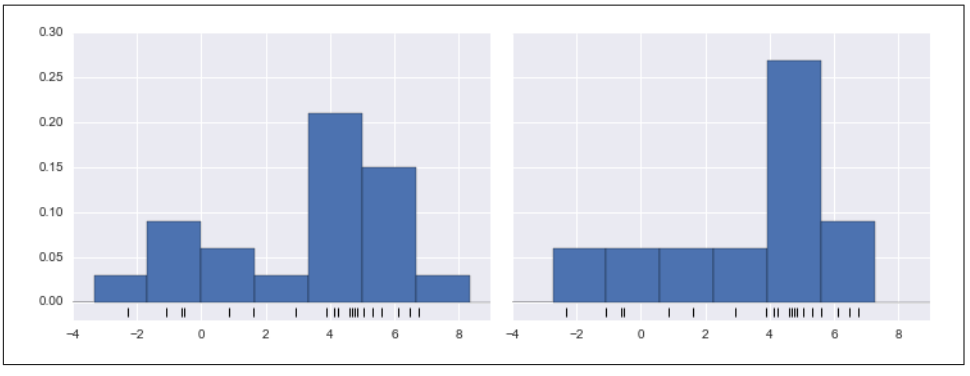

Figure 5-141. The problem with histograms: the location of bins can affect interpretation

On the left, the histogram makes clear that this is a bimodal distribution. On the right, we see a unimodal distribution with a long tail. Without seeing the preceding code, you would probably not guess that these two histograms were built from the same data. With that in mind, how can you trust the intuition that histograms confer? And how might we improve on this?

Stepping back, we can think of a histogram as a stack of blocks, where we stack one block within each bin on top of each point in the dataset. Let's view this directly (Figure 5-142):

```
In[7]: fig, ax = plt.subplots()
       bins = np.arange(-3, 8)
       ax.plot(x, np.full_like(x, -0.1), '|k',
               markeredgewidth=1)
       for count, edge in zip(*np.histogram(x, bins)):
           for i in range(count):
               ax.add_patch(plt.Rectangle((edge, i), 1, 1,
                                          alpha=0.5))
       ax.set_xlim(-4, 8)
       ax.set_ylim(-0.2, 8)

Out[7]: (-0.2, 8)
```

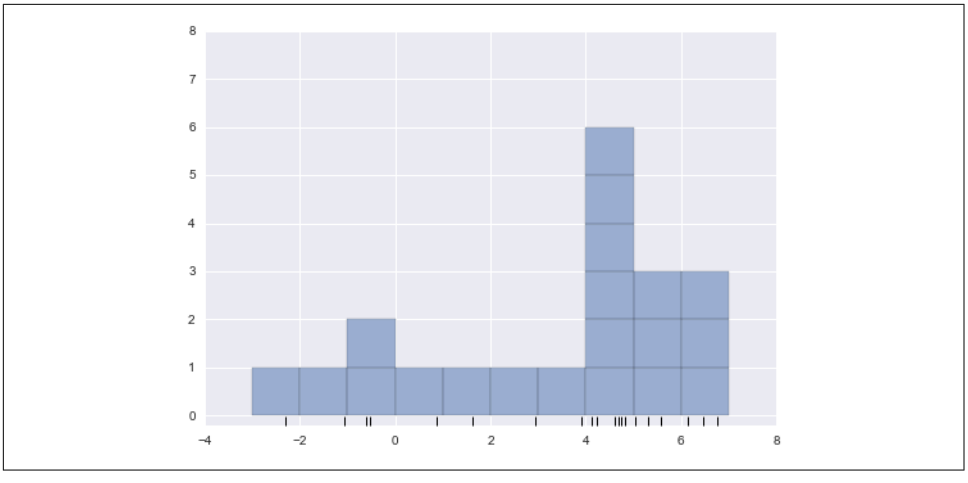

Figure 5-142. Histogram as stack of blocks

The problem with our two binnings stems from the fact that the height of the block stack often reflects not on the actual density of points nearby, but on coincidences of how the bins align with the data points. This misalignment between points and their blocks is a potential cause of the poor histogram results seen here. But what if, instead of stacking the blocks aligned with the *bins*, we were to stack the blocks aligned with the *points they represent*? If we do this, the blocks won't be aligned, but we can add their contributions at each location along the x-axis to find the result. Let's try this (Figure 5-143):

```
In[8]: x_d = np.linspace(-4, 8, 2000)
       density = sum((abs(xi - x_d) < 0.5) for xi in x)

       plt.fill_between(x_d, density, alpha=0.5)
       plt.plot(x, np.full_like(x, -0.1), '|k', markeredgewidth=1)

       plt.axis([-4, 8, -0.2, 8]);
```

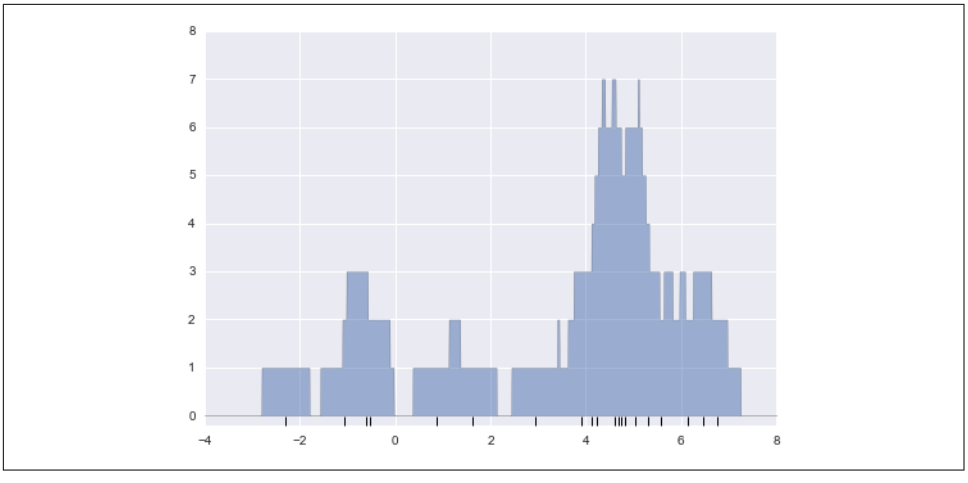

Figure 5-143. A "histogram" where blocks center on each individual point; this is an example of a kernel density estimate

The result looks a bit messy, but is a much more robust reflection of the actual data characteristics than is the standard histogram. Still, the rough edges are not aesthetically pleasing, nor are they reflective of any true properties of the data. In order to smooth them out, we might decide to replace the blocks at each location with a smooth function, like a Gaussian. Let's use a standard normal curve at each point instead of a block (Figure 5-144):

```
In[9]: from scipy.stats import norm
       x_d = np.linspace(-4, 8, 1000)
       density = sum(norm(xi).pdf(x_d) for xi in x)

       plt.fill_between(x_d, density, alpha=0.5)
       plt.plot(x, np.full_like(x, -0.1), '|k', markeredgewidth=1)

       plt.axis([-4, 8, -0.2, 5]);
```

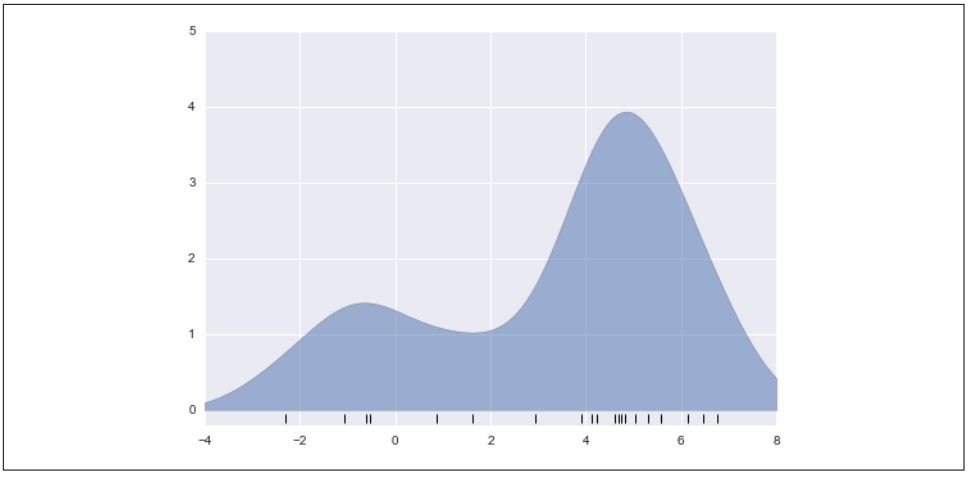

Figure 5-144. A kernel density estimate with a Gaussian kernel

This smoothed-out plot, with a Gaussian distribution contributed at the location of each input point, gives a much more accurate idea of the shape of the data distribution, and one that has much less variance (i.e., changes much less in response to differences in sampling).

These last two plots are examples of kernel density estimation in one dimension: the first uses a so-called "tophat" kernel and the second uses a Gaussian kernel. We'll now look at kernel density estimation in more detail.

Kernel Density Estimation in Practice

The free parameters of kernel density estimation are the *kernel*, which specifies the shape of the distribution placed at each point, and the *kernel bandwidth*, which controls the size of the kernel at each point. In practice, there are many kernels you might use for a kernel density estimation: in particular, the Scikit-Learn KDE implementation supports one of six kernels, which you can read about in Scikit-Learn's Density Estimation documentation.

While there are several versions of kernel density estimation implemented in Python (notably in the SciPy and StatsModels packages), I prefer to use Scikit-Learn's version because of its efficiency and flexibility. It is implemented in the `sklearn.neighbors.KernelDensity` estimator, which handles KDE in multiple dimensions with one of six kernels and one of a couple dozen distance metrics. Because KDE can be fairly computationally intensive, the Scikit-Learn estimator uses a tree-based algorithm under the hood and can trade off computation time for accuracy using the `atol` (absolute tolerance) and `rtol` (relative tolerance) parameters. We can determine the

kernel bandwidth, which is a free parameter, using Scikit-Learn's standard cross-validation tools, as we will soon see.

Let's first see a simple example of replicating the preceding plot using the Scikit-Learn `KernelDensity` estimator (Figure 5-145):

```
In[10]: from sklearn.neighbors import KernelDensity

        # instantiate and fit the KDE model
        kde = KernelDensity(bandwidth=1.0, kernel='gaussian')
        kde.fit(x[:, None])

        # score_samples returns the log of the probability density
        logprob = kde.score_samples(x_d[:, None])

        plt.fill_between(x_d, np.exp(logprob), alpha=0.5)
        plt.plot(x, np.full_like(x, -0.01), '|k', markeredgewidth=1)
        plt.ylim(-0.02, 0.22)

Out[10]: (-0.02, 0.22)
```

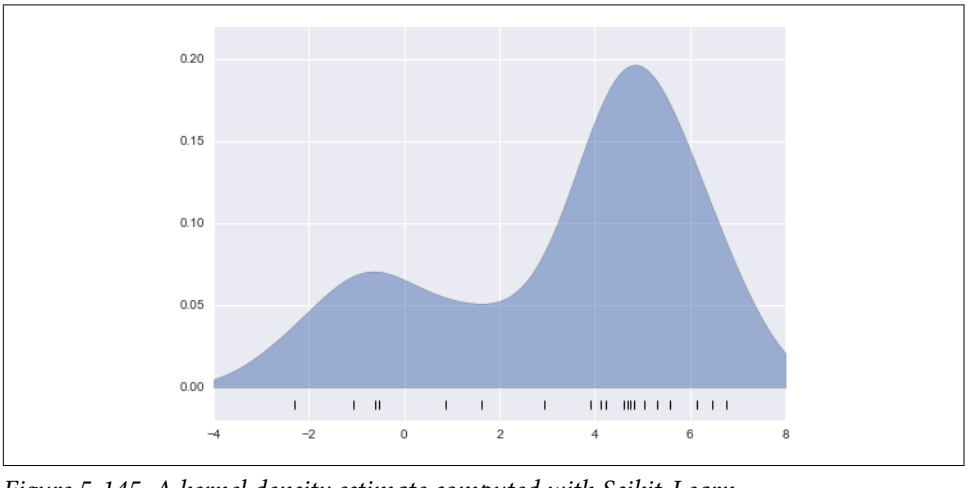

Figure 5-145. A kernel density estimate computed with Scikit-Learn

The result here is normalized such that the area under the curve is equal to 1.

Selecting the bandwidth via cross-validation

The choice of bandwidth within KDE is extremely important to finding a suitable density estimate, and is the knob that controls the bias–variance trade-off in the estimate of density: too narrow a bandwidth leads to a high-variance estimate (i.e., overfitting), where the presence or absence of a single point makes a large difference. Too wide a bandwidth leads to a high-bias estimate (i.e., underfitting) where the structure in the data is washed out by the wide kernel.

There is a long history in statistics of methods to quickly estimate the best bandwidth based on rather stringent assumptions about the data: if you look up the KDE implementations in the SciPy and StatsModels packages, for example, you will see implementations based on some of these rules.

In machine learning contexts, we've seen that such hyperparameter tuning often is done empirically via a cross-validation approach. With this in mind, the `KernelDensity` estimator in Scikit-Learn is designed such that it can be used directly within Scikit-Learn's standard grid search tools. Here we will use `GridSearchCV` to optimize the bandwidth for the preceding dataset. Because we are looking at such a small dataset, we will use leave-one-out cross-validation, which minimizes the reduction in training set size for each cross-validation trial:

```
In[11]: from sklearn.grid_search import GridSearchCV
        from sklearn.cross_validation import LeaveOneOut

        bandwidths = 10 ** np.linspace(-1, 1, 100)
        grid = GridSearchCV(KernelDensity(kernel='gaussian'),
                            {'bandwidth': bandwidths},
                            cv=LeaveOneOut(len(x)))
        grid.fit(x[:, None]);
```

Now we can find the choice of bandwidth that maximizes the score (which in this case defaults to the log-likelihood):

```
In[12]: grid.best_params_

Out[12]: {'bandwidth': 1.1233240329780276}
```

The optimal bandwidth happens to be very close to what we used in the example plot earlier, where the bandwidth was 1.0 (i.e., the default width of `scipy.stats.norm`).

Example: KDE on a Sphere

Perhaps the most common use of KDE is in graphically representing distributions of points. For example, in the Seaborn visualization library (discussed earlier in "Visualization with Seaborn" on page 311), KDE is built in and automatically used to help visualize points in one and two dimensions.

Here we will look at a slightly more sophisticated use of KDE for visualization of distributions. We will make use of some geographic data that can be loaded with Scikit-Learn: the geographic distributions of recorded observations of two South American mammals, *Bradypus variegatus* (the brown-throated sloth) and *Microryzomys minutus* (the forest small rice rat).

With Scikit-Learn, we can fetch this data as follows:

```
In[13]: from sklearn.datasets import fetch_species_distributions

        data = fetch_species_distributions()
```

```
# Get matrices/arrays of species IDs and locations
latlon = np.vstack([data.train['dd lat'],
                    data.train['dd long']]).T
species = np.array([d.decode('ascii').startswith('micro')
                    for d in data.train['species']], dtype='int')
```

With this data loaded, we can use the Basemap toolkit (mentioned previously in "Geographic Data with Basemap" on page 298) to plot the observed locations of these two species on the map of South America (Figure 5-146):

```
In[14]: from mpl_toolkits.basemap import Basemap
        from sklearn.datasets.species_distributions import construct_grids

        xgrid, ygrid = construct_grids(data)

        # plot coastlines with Basemap
        m = Basemap(projection='cyl', resolution='c',
                    llcrnrlat=ygrid.min(), urcrnrlat=ygrid.max(),
                    llcrnrlon=xgrid.min(), urcrnrlon=xgrid.max())
        m.drawmapboundary(fill_color='#DDEEFF')
        m.fillcontinents(color='#FFEEDD')
        m.drawcoastlines(color='gray', zorder=2)
        m.drawcountries(color='gray', zorder=2)

        # plot locations
        m.scatter(latlon[:, 1], latlon[:, 0], zorder=3,
                  c=species, cmap='rainbow', latlon=True);
```

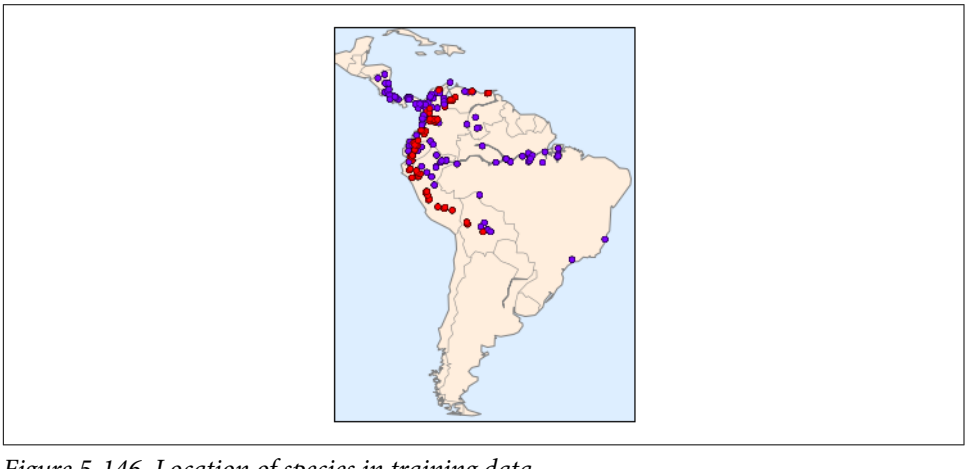

Figure 5-146. Location of species in training data

Unfortunately, this doesn't give a very good idea of the density of the species, because points in the species range may overlap one another. You may not realize it by looking at this plot, but there are over 1,600 points shown here!

Let's use kernel density estimation to show this distribution in a more interpretable way: as a smooth indication of density on the map. Because the coordinate system here lies on a spherical surface rather than a flat plane, we will use the `haversine` distance metric, which will correctly represent distances on a curved surface.

There is a bit of boilerplate code here (one of the disadvantages of the Basemap toolkit), but the meaning of each code block should be clear (Figure 5-147):

```
In[15]:
# Set up the data grid for the contour plot
X, Y = np.meshgrid(xgrid[::5], ygrid[::5][::-1])
land_reference = data.coverages[6][::5, ::5]
land_mask = (land_reference > -9999).ravel()
xy = np.vstack([Y.ravel(), X.ravel()]).T
xy = np.radians(xy[land_mask])

# Create two side-by-side plots
fig, ax = plt.subplots(1, 2)
fig.subplots_adjust(left=0.05, right=0.95, wspace=0.05)
species_names = ['Bradypus Variegatus', 'Microryzomys Minutus']
cmaps = ['Purples', 'Reds']

for i, axi in enumerate(ax):
    axi.set_title(species_names[i])

    # plot coastlines with Basemap
    m = Basemap(projection='cyl', llcrnrlat=Y.min(),
                urcrnrlat=Y.max(), llcrnrlon=X.min(),
                urcrnrlon=X.max(), resolution='c', ax=axi)
    m.drawmapboundary(fill_color='#DDEEFF')
    m.drawcoastlines()
    m.drawcountries()

    # construct a spherical kernel density estimate of the distribution
    kde = KernelDensity(bandwidth=0.03, metric='haversine')
    kde.fit(np.radians(latlon[species == i]))

    # evaluate only on the land: -9999 indicates ocean
    Z = np.full(land_mask.shape[0], -9999.0)
    Z[land_mask] = np.exp(kde.score_samples(xy))
    Z = Z.reshape(X.shape)

    # plot contours of the density
    levels = np.linspace(0, Z.max(), 25)
    axi.contourf(X, Y, Z, levels=levels, cmap=cmaps[i])
```

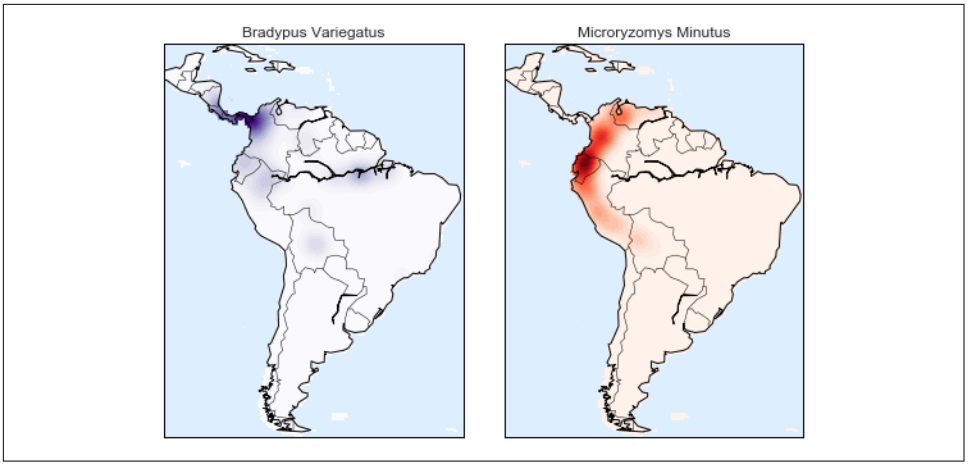

Figure 5-147. A kernel density representation of the species distributions

Compared to the simple scatter plot we initially used, this visualization paints a much clearer picture of the geographical distribution of observations of these two species.

Example: Not-So-Naive Bayes

This example looks at Bayesian generative classification with KDE, and demonstrates how to use the Scikit-Learn architecture to create a custom estimator.

In "In Depth: Naive Bayes Classification" on page 382, we took a look at naive Bayesian classification, in which we created a simple generative model for each class, and used these models to build a fast classifier. For naive Bayes, the generative model is a simple axis-aligned Gaussian. With a density estimation algorithm like KDE, we can remove the "naive" element and perform the same classification with a more sophisticated generative model for each class. It's still Bayesian classification, but it's no longer naive.

The general approach for generative classification is this:

1. Split the training data by label.
2. For each set, fit a KDE to obtain a generative model of the data. This allows you for any observation x and label y to compute a likelihood $P(x \mid y)$.
3. From the number of examples of each class in the training set, compute the *class prior*, $P(y)$.
4. For an unknown point x, the posterior probability for each class is $P(y \mid x) \propto P(x \mid y)P(y)$. The class that maximizes this posterior is the label assigned to the point.

The algorithm is straightforward and intuitive to understand; the more difficult piece is couching it within the Scikit-Learn framework in order to make use of the grid search and cross-validation architecture.

This is the code that implements the algorithm within the Scikit-Learn framework; we will step through it following the code block:

```
In[16]: from sklearn.base import BaseEstimator, ClassifierMixin


        class KDEClassifier(BaseEstimator, ClassifierMixin):
            """Bayesian generative classification based on KDE

            Parameters
            ----------
            bandwidth : float
                the kernel bandwidth within each class
            kernel : str
                the kernel name, passed to KernelDensity
            """
            def __init__(self, bandwidth=1.0, kernel='gaussian'):
                self.bandwidth = bandwidth
                self.kernel = kernel

            def fit(self, X, y):
                self.classes_ = np.sort(np.unique(y))
                training_sets = [X[y == yi] for yi in self.classes_]
                self.models_ = [KernelDensity(bandwidth=self.bandwidth,
                                              kernel=self.kernel).fit(Xi)
                                for Xi in training_sets]
                self.logpriors_ = [np.log(Xi.shape[0] / X.shape[0])
                                   for Xi in training_sets]
                return self

            def predict_proba(self, X):
                logprobs = np.array([model.score_samples(X)
                                     for model in self.models_]).T
                result = np.exp(logprobs + self.logpriors_)
                return result / result.sum(1, keepdims=True)

            def predict(self, X):
                return self.classes_[np.argmax(self.predict_proba(X), 1)]
```

The anatomy of a custom estimator

Let's step through this code and discuss the essential features:

```
from sklearn.base import BaseEstimator, ClassifierMixin

class KDEClassifier(BaseEstimator, ClassifierMixin):
    """Bayesian generative classification based on KDE
```

```
    Parameters
    ----------
    bandwidth : float
        the kernel bandwidth within each class
    kernel : str
        the kernel name, passed to KernelDensity
    """
```

Each estimator in Scikit-Learn is a class, and it is most convenient for this class to inherit from the `BaseEstimator` class as well as the appropriate mixin, which provides standard functionality. For example, among other things, here the `BaseEstimator` contains the logic necessary to clone/copy an estimator for use in a cross-validation procedure, and `ClassifierMixin` defines a default `score()` method used by such routines. We also provide a docstring, which will be captured by IPython's help functionality (see "Help and Documentation in IPython" on page 3).

Next comes the class initialization method:

```
    def __init__(self, bandwidth=1.0, kernel='gaussian'):
        self.bandwidth = bandwidth
        self.kernel = kernel
```

This is the actual code executed when the object is instantiated with `KDEClassifier()`. In Scikit-Learn, it is important that *initialization contains no operations* other than assigning the passed values by name to `self`. This is due to the logic contained in `BaseEstimator` required for cloning and modifying estimators for cross-validation, grid search, and other functions. Similarly, all arguments to `__init__` should be explicit; that is, `*args` or `**kwargs` should be avoided, as they will not be correctly handled within cross-validation routines.

Next comes the `fit()` method, where we handle training data:

```
    def fit(self, X, y):
        self.classes_ = np.sort(np.unique(y))
        training_sets = [X[y == yi] for yi in self.classes_]
        self.models_ = [KernelDensity(bandwidth=self.bandwidth,
                                      kernel=self.kernel).fit(Xi)
                        for Xi in training_sets]
        self.logpriors_ = [np.log(Xi.shape[0] / X.shape[0])
                           for Xi in training_sets]
        return self
```

Here we find the unique classes in the training data, train a `KernelDensity` model for each class, and compute the class priors based on the number of input samples. Finally, `fit()` should always return `self` so that we can chain commands. For example:

```
label = model.fit(X, y).predict(X)
```

Notice that each persistent result of the fit is stored with a trailing underscore (e.g., `self.logpriors_`). This is a convention used in Scikit-Learn so that you can quickly scan the members of an estimator (using IPython's tab completion) and see exactly which members are fit to training data.

Finally, we have the logic for predicting labels on new data:

```
    def predict_proba(self, X):
        logprobs = np.vstack([model.score_samples(X)
                              for model in self.models_]).T
        result = np.exp(logprobs + self.logpriors_)
        return result / result.sum(1, keepdims=True)

    def predict(self, X):
        return self.classes_[np.argmax(self.predict_proba(X), 1)]
```

Because this is a probabilistic classifier, we first implement `predict_proba()`, which returns an array of class probabilities of shape `[n_samples, n_classes]`. Entry `[i, j]` of this array is the posterior probability that sample `i` is a member of class `j`, computed by multiplying the likelihood by the class prior and normalizing.

Finally, the `predict()` method uses these probabilities and simply returns the class with the largest probability.

Using our custom estimator

Let's try this custom estimator on a problem we have seen before: the classification of handwritten digits. Here we will load the digits, and compute the cross-validation score for a range of candidate bandwidths using the `GridSearchCV` meta-estimator (refer back to "Hyperparameters and Model Validation" on page 359 for more information on this):

```
In[17]: from sklearn.datasets import load_digits
        from sklearn.grid_search import GridSearchCV

        digits = load_digits()

        bandwidths = 10 ** np.linspace(0, 2, 100)
        grid = GridSearchCV(KDEClassifier(), {'bandwidth': bandwidths})
        grid.fit(digits.data, digits.target)

        scores = [val.mean_validation_score for val in grid.grid_scores_]
```

Next we can plot the cross-validation score as a function of bandwidth (Figure 5-148):

```
In[18]: plt.semilogx(bandwidths, scores)
        plt.xlabel('bandwidth')
        plt.ylabel('accuracy')
        plt.title('KDE Model Performance')
```

```
        print(grid.best_params_)
        print('accuracy =', grid.best_score_)
```

```
{'bandwidth': 7.0548023107186433}
accuracy = 0.966611018364
```

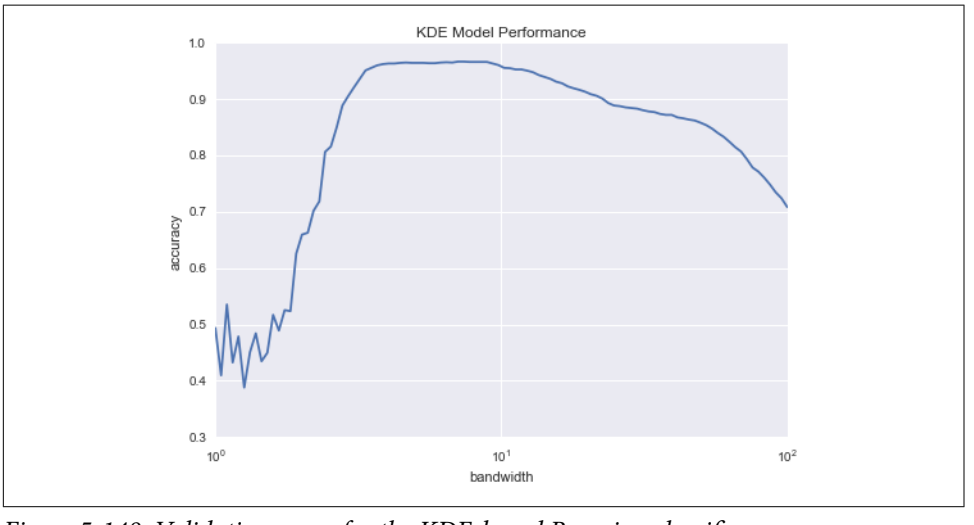

Figure 5-148. Validation curve for the KDE-based Bayesian classifier

We see that this not-so-naive Bayesian classifier reaches a cross-validation accuracy of just over 96%; this is compared to around 80% for the naive Bayesian classification:

```
In[19]: from sklearn.naive_bayes import GaussianNB
        from sklearn.cross_validation import cross_val_score
        cross_val_score(GaussianNB(), digits.data, digits.target).mean()
```

```
Out[19]: 0.81860038035501381
```

One benefit of such a generative classifier is interpretability of results: for each unknown sample, we not only get a probabilistic classification, but a *full model* of the distribution of points we are comparing it to! If desired, this offers an intuitive window into the reasons for a particular classification that algorithms like SVMs and random forests tend to obscure.

If you would like to take this further, there are some improvements that could be made to our KDE classifier model:

- We could allow the bandwidth in each class to vary independently.
- We could optimize these bandwidths not based on their prediction score, but on the likelihood of the training data under the generative model within each class (i.e., use the scores from `KernelDensity` itself rather than the global prediction accuracy).

Finally, if you want some practice building your own estimator, you might tackle building a similar Bayesian classifier using Gaussian mixture models instead of KDE.

Application: A Face Detection Pipeline

This chapter has explored a number of the central concepts and algorithms of machine learning. But moving from these concepts to real-world application can be a challenge. Real-world datasets are noisy and heterogeneous, may have missing features, and may include data in a form that is difficult to map to a clean `[n_samples, n_features]` matrix. Before applying any of the methods discussed here, you must first extract these features from your data; there is no formula for how to do this that applies across all domains, and thus this is where you as a data scientist must exercise your own intuition and expertise.

One interesting and compelling application of machine learning is to images, and we have already seen a few examples of this where pixel-level features are used for classification. In the real world, data is rarely so uniform and simple pixels will not be suitable, a fact that has led to a large literature on *feature extraction* methods for image data (see "Feature Engineering" on page 375).

In this section, we will take a look at one such feature extraction technique, the Histogram of Oriented Gradients (HOG), which transforms image pixels into a vector representation that is sensitive to broadly informative image features regardless of confounding factors like illumination. We will use these features to develop a simple face detection pipeline, using machine learning algorithms and concepts we've seen throughout this chapter. We begin with the standard imports:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
       import numpy as np
```

HOG Features

The Histogram of Gradients is a straightforward feature extraction procedure that was developed in the context of identifying pedestrians within images. HOG involves the following steps:

1. Optionally prenormalize images. This leads to features that resist dependence on variations in illumination.
2. Convolve the image with two filters that are sensitive to horizontal and vertical brightness gradients. These capture edge, contour, and texture information.
3. Subdivide the image into cells of a predetermined size, and compute a histogram of the gradient orientations within each cell.

4. Normalize the histograms in each cell by comparing to the block of neighboring cells. This further suppresses the effect of illumination across the image.
5. Construct a one-dimensional feature vector from the information in each cell.

A fast HOG extractor is built into the Scikit-Image project, and we can try it out relatively quickly and visualize the oriented gradients within each cell (Figure 5-149):

```
In[2]: from skimage import data, color, feature
       import skimage.data

       image = color.rgb2gray(data.chelsea())
       hog_vec, hog_vis = feature.hog(image, visualise=True)

       fig, ax = plt.subplots(1, 2, figsize=(12, 6),
                              subplot_kw=dict(xticks=[], yticks=[]))
       ax[0].imshow(image, cmap='gray')
       ax[0].set_title('input image')

       ax[1].imshow(hog_vis)
       ax[1].set_title('visualization of HOG features');
```

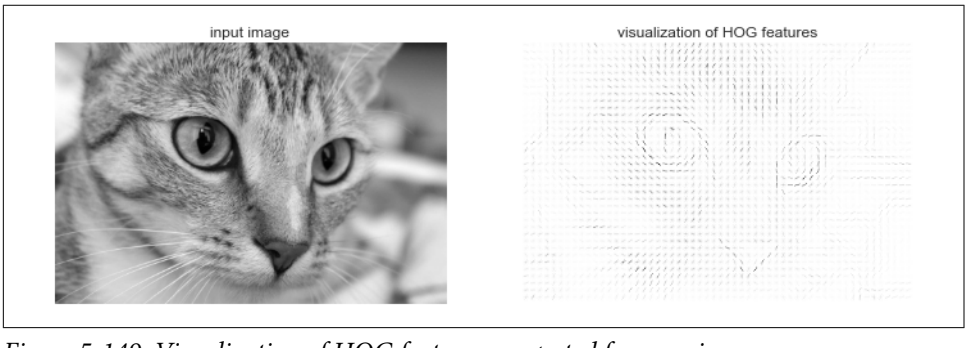

Figure 5-149. Visualization of HOG features computed from an image

HOG in Action: A Simple Face Detector

Using these HOG features, we can build up a simple facial detection algorithm with any Scikit-Learn estimator; here we will use a linear support vector machine (refer back to "In-Depth: Support Vector Machines" on page 405 if you need a refresher on this). The steps are as follows:

1. Obtain a set of image thumbnails of faces to constitute "positive" training samples.
2. Obtain a set of image thumbnails of nonfaces to constitute "negative" training samples.
3. Extract HOG features from these training samples.

4. Train a linear SVM classifier on these samples.
5. For an "unknown" image, pass a sliding window across the image, using the model to evaluate whether that window contains a face or not.
6. If detections overlap, combine them into a single window.

Let's go through these steps and try it out:

1. Obtain a set of positive training samples.

   Let's start by finding some positive training samples that show a variety of faces. We have one easy set of data to work with—the Labeled Faces in the Wild dataset, which can be downloaded by Scikit-Learn:

   ```
   In[3]: from sklearn.datasets import fetch_lfw_people
          faces = fetch_lfw_people()
          positive_patches = faces.images
          positive_patches.shape
   ```

   ```
   Out[3]: (13233, 62, 47)
   ```

   This gives us a sample of 13,000 face images to use for training.

2. Obtain a set of negative training samples.

   Next we need a set of similarly sized thumbnails that *do not* have a face in them. One way to do this is to take any corpus of input images, and extract thumbnails from them at a variety of scales. Here we can use some of the images shipped with Scikit-Image, along with Scikit-Learn's `PatchExtractor`:

   ```
   In[4]: from skimage import data, transform

          imgs_to_use = ['camera', 'text', 'coins', 'moon',
                         'page', 'clock', 'immunohistochemistry',
                         'chelsea', 'coffee', 'hubble_deep_field']
          images = [color.rgb2gray(getattr(data, name)())
                    for name in imgs_to_use]
   ```

   ```
   In[5]:
   from sklearn.feature_extraction.image import PatchExtractor

   def extract_patches(img, N, scale=1.0,
                       patch_size=positive_patches[0].shape):
       extracted_patch_size = \
       tuple((scale * np.array(patch_size)).astype(int))
       extractor = PatchExtractor(patch_size=extracted_patch_size,
                                  max_patches=N, random_state=0)
       patches = extractor.transform(img[np.newaxis])
       if scale != 1:
           patches = np.array([transform.resize(patch, patch_size)
                               for patch in patches])
       return patches
   ```

```
negative_patches = np.vstack([extract_patches(im, 1000, scale)
                              for im in images for scale in [0.5, 1.0, 2.0]])
negative_patches.shape
```

```
Out[5]: (30000, 62, 47)
```

We now have 30,000 suitable image patches that do not contain faces. Let's take a look at a few of them to get an idea of what they look like (Figure 5-150):

```
In[6]: fig, ax = plt.subplots(6, 10)
       for i, axi in enumerate(ax.flat):
           axi.imshow(negative_patches[500 * i], cmap='gray')
           axi.axis('off')
```

Figure 5-150. Negative image patches, which don't include faces

Our hope is that these would sufficiently cover the space of "nonfaces" that our algorithm is likely to see.

3. Combine sets and extract HOG features.

   Now that we have these positive samples and negative samples, we can combine them and compute HOG features. This step takes a little while, because the HOG features involve a nontrivial computation for each image:

```
In[7]: from itertools import chain
       X_train = np.array([feature.hog(im)
                           for im in chain(positive_patches,
                                           negative_patches)])
       y_train = np.zeros(X_train.shape[0])
       y_train[:positive_patches.shape[0]] = 1
```

```
In[8]: X_train.shape
```

```
Out[8]: (43233, 1215)
```

We are left with 43,000 training samples in 1,215 dimensions, and we now have our data in a form that we can feed into Scikit-Learn!

4. Train a support vector machine.

   Next we use the tools we have been exploring in this chapter to create a classifier of thumbnail patches. For such a high-dimensional binary classification task, a linear support vector machine is a good choice. We will use Scikit-Learn's `LinearSVC`, because in comparison to `SVC` it often has better scaling for large number of samples.

   First, though, let's use a simple Gaussian naive Bayes to get a quick baseline:

```
In[9]: from sklearn.naive_bayes import GaussianNB
       from sklearn.cross_validation import cross_val_score

       cross_val_score(GaussianNB(), X_train, y_train)
```

```
Out[9]: array([ 0.9408785 ,  0.8752342 ,  0.93976823])
```

   We see that on our training data, even a simple naive Bayes algorithm gets us upward of 90% accuracy. Let's try the support vector machine, with a grid search over a few choices of the C parameter:

```
In[10]: from sklearn.svm import LinearSVC
        from sklearn.grid_search import GridSearchCV
        grid = GridSearchCV(LinearSVC(), {'C': [1.0, 2.0, 4.0, 8.0]})
        grid.fit(X_train, y_train)
        grid.best_score_
```

```
Out[10]: 0.98667684407744083
```

```
In[11]: grid.best_params_
```

```
Out[11]: {'C': 4.0}
```

   Let's take the best estimator and retrain it on the full dataset:

```
In[12]: model = grid.best_estimator_
        model.fit(X_train, y_train)
```

```
Out[12]: LinearSVC(C=4.0, class_weight=None, dual=True,
              fit_intercept=True, intercept_scaling=1,
              loss='squared_hinge', max_iter=1000,
              multi_class='ovr', penalty='l2',
              random_state=None, tol=0.0001, verbose=0)
```

5. Find faces in a new image.

   Now that we have this model in place, let's grab a new image and see how the model does. We will use one portion of the astronaut image for simplicity (see discussion of this in "Caveats and Improvements" on page 512), and run a sliding window over it and evaluate each patch (Figure 5-151):

```
In[13]: test_image = skimage.data.astronaut()
        test_image = skimage.color.rgb2gray(test_image)
        test_image = skimage.transform.rescale(test_image, 0.5)
        test_image = test_image[:160, 40:180]

        plt.imshow(test_image, cmap='gray')
        plt.axis('off');
```

Figure 5-151. An image in which we will attempt to locate a face

Next, let's create a window that iterates over patches of this image, and compute HOG features for each patch:

```
In[14]: def sliding_window(img, patch_size=positive_patches[0].shape,
                           istep=2, jstep=2, scale=1.0):
            Ni, Nj = (int(scale * s) for s in patch_size)
            for i in range(0, img.shape[0] - Ni, istep):
                for j in range(0, img.shape[1] - Ni, jstep):
                    patch = img[i:i + Ni, j:j + Nj]
                    if scale != 1:
                        patch = transform.resize(patch, patch_size)
                    yield (i, j), patch

        indices, patches = zip(*sliding_window(test_image))
        patches_hog = np.array([feature.hog(patch) for patch in patches])
        patches_hog.shape
```

```
Out[14]: (1911, 1215)
```

Finally, we can take these HOG-featured patches and use our model to evaluate whether each patch contains a face:

```
In[15]: labels = model.predict(patches_hog)
        labels.sum()
```

```
Out[15]: 33.0
```

We see that out of nearly 2,000 patches, we have found 30 detections. Let's use the information we have about these patches to show where they lie on our test image, drawing them as rectangles (Figure 5-152):

```
In[16]: fig, ax = plt.subplots()
        ax.imshow(test_image, cmap='gray')
        ax.axis('off')

        Ni, Nj = positive_patches[0].shape
        indices = np.array(indices)

        for i, j in indices[labels == 1]:
            ax.add_patch(plt.Rectangle((j, i), Nj, Ni, edgecolor='red',
                                       alpha=0.3, lw=2,
                                       facecolor='none'))
```

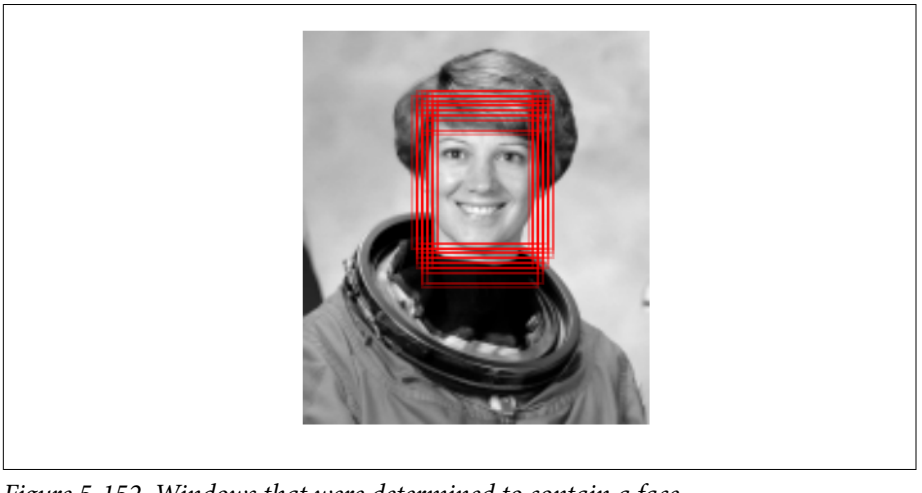

Figure 5-152. Windows that were determined to contain a face

All of the detected patches overlap and found the face in the image! Not bad for a few lines of Python.

Caveats and Improvements

If you dig a bit deeper into the preceding code and examples, you'll see that we still have a bit of work before we can claim a production-ready face detector. There are several issues with what we've done, and several improvements that could be made. In particular:

Our training set, especially for negative features, is not very complete
: The central issue is that there are many face-like textures that are not in the training set, and so our current model is very prone to false positives. You can see this if you try out the preceding algorithm on the *full* astronaut image: the current model leads to many false detections in other regions of the image.

  We might imagine addressing this by adding a wider variety of images to the negative training set, and this would probably yield some improvement. Another way to address this is to use a more directed approach, such as *hard negative mining*. In hard negative mining, we take a new set of images that our classifier has not seen, find all the patches representing false positives, and explicitly add them as negative instances in the training set before retraining the classifier.

Our current pipeline searches only at one scale
: As currently written, our algorithm will miss faces that are not approximately 62×47 pixels. We can straightforwardly address this by using sliding windows of a variety of sizes, and resizing each patch using `skimage.transform.resize` before feeding it into the model. In fact, the `sliding_window()` utility used here is already built with this in mind.

We should combine overlapped detection patches
: For a production-ready pipeline, we would prefer not to have 30 detections of the same face, but to somehow reduce overlapping groups of detections down to a single detection. This could be done via an unsupervised clustering approach (MeanShift Clustering is one good candidate for this), or via a procedural approach such as *nonmaximum suppression*, an algorithm common in machine vision.

The pipeline should be streamlined
: Once we address these issues, it would also be nice to create a more streamlined pipeline for ingesting training images and predicting sliding-window outputs. This is where Python as a data science tool really shines: with a bit of work, we could take our prototype code and package it with a well-designed object-oriented API that gives the user the ability to use this easily. I will leave this as a proverbial "exercise for the reader."

More recent advances, such as deep learning, should be considered
: Finally, I should add that HOG and other procedural feature extraction methods for images are no longer state-of-the-art techniques. Instead, many modern object detection pipelines use variants of deep neural networks. One way to think of neural networks is that they are an estimator that determines optimal feature extraction strategies from the data, rather than relying on the intuition of the user. An intro to these deep neural net methods is conceptually (and computationally!) beyond the scope of this section, although open tools like Google's

TensorFlow have recently made deep learning approaches much more accessible than they once were. As of the writing of this book, deep learning in Python is still relatively young, and so I can't yet point to any definitive resource. That said, the list of references in the following section should provide a useful place to start.

Further Machine Learning Resources

This chapter has been a quick tour of machine learning in Python, primarily using the tools within the Scikit-Learn library. As long as the chapter is, it is still too short to cover many interesting and important algorithms, approaches, and discussions. Here I want to suggest some resources for those who would like to learn more about machine learning.

Machine Learning in Python

To learn more about machine learning in Python, I'd suggest some of the following resources:

The Scikit-Learn website
: The Scikit-Learn website has an impressive breadth of documentation and examples covering some of the models discussed here, and much, much more. If you want a brief survey of the most important and often used machine learning algorithms, this website is a good place to start.

SciPy, PyCon, and PyData tutorial videos
: Scikit-Learn and other machine learning topics are perennial favorites in the tutorial tracks of many Python-focused conference series, in particular the PyCon, SciPy, and PyData conferences. You can find the most recent ones via a simple web search.

Introduction to Machine Learning with Python
: Written by Andreas C. Mueller and Sarah Guido, this book includes a fuller treatment of the topics in this chapter. If you're interested in reviewing the fundamentals of machine learning and pushing the Scikit-Learn toolkit to its limits, this is a great resource, written by one of the most prolific developers on the Scikit-Learn team.

Python Machine Learning
: Sebastian Raschka's book focuses less on Scikit-Learn itself, and more on the breadth of machine learning tools available in Python. In particular, there is some very useful discussion on how to scale Python-based machine learning approaches to large and complex datasets.

General Machine Learning

Of course, machine learning is much broader than just the Python world. There are many good resources to take your knowledge further, and here I highlight a few that I have found useful:

Machine Learning
: Taught by Andrew Ng (Coursera), this is a very clearly taught, free online course covering the basics of machine learning from an algorithmic perspective. It assumes undergraduate-level understanding of mathematics and programming, and steps through detailed considerations of some of the most important machine learning algorithms. Homework assignments, which are algorithmically graded, have you actually implement some of these models yourself.

Pattern Recognition and Machine Learning
: Written by Christopher Bishop, this classic technical text covers the concepts of machine learning discussed in this chapter in detail. If you plan to go further in this subject, you should have this book on your shelf.

Machine Learning: A Probabilistic Perspective
: Written by Kevin Murphy, this is an excellent graduate-level text that explores nearly all important machine learning algorithms from a ground-up, unified probabilistic perspective.

These resources are more technical than the material presented in this book, but to really understand the fundamentals of these methods requires a deep dive into the mathematics behind them. If you're up for the challenge and ready to bring your data science to the next level, don't hesitate to dive in!

Index

Symbols

%automagic, 19
%cpaste, 11
%debug, 22
%history, 16
%lprun, 28
%lsmagic, 13
%magic, 13
%matplotlib, 219
%memit, 29
%mode, 20-22
%mprun, 29
%paste, 11
%prun, 27
%run, 12
%time, 25-27
%timeit, 12, 25-27
& (ampersand), 77
* (asterisk), 7
: (colon), 44
? (question mark), 3
?? (double question mark), 5
_ (underscore) shortcut, 15
| (operator), 77

A

absolute value function, 54
aggregate() method, 166
aggregates
    computed directly from object, 57
    multidimensional, 60
    summarizing set of values with, 61
aggregation (NumPy), 58-63
    minimum and maximum, 59
    multidimensional aggregates, 60
    presidents average height example, 61
    summing the values in an array, 59
    various functions, 61
aggregation (Pandas), 158-170
    groupby() operation, 161-170
    MultiIndex, 140
    Planets dataset for, 159
    simple aggregation, 159-161
Akaike information criterion (AIC), 487, 489
Albers equal-area projection, 303
algorithmic efficiency
    big-O notation, 92
    dataset size and, 85
ampersand (&), 77
Anaconda, xiv
and keyword, 77
annotation of plots, 268-275
    arrows, 272-275
    holidays/US births example, 269
    transforms and text position, 270-272
APIs (see Estimator API)
append() method, Pandas vs. Python, 146
apply() method, 167
arithmetic operators, 52
arrays
    accessing single rows/columns, 45
    arithmetic operators, 52
    attributes, 42
    basics, 42
    Boolean, 73-75
    broadcasting, 63-69
    centering, 68
    computation on, 50-58

concatenation, 48, 142
creating copies, 46
creating from Python lists, 39
creating from scratch, 39
data as, 33
DataFrame object as, 102
DataFrame object constructed from, 105
fixed-type, 38
Index object as immutable array, 106
Index object vs., 106
indexing: accessing single elements, 43
reshaping, 47
Series object vs., 99
slicing, 44
slicing multidimensional subarrays, 45
slicing one-dimensional subarrays, 44
sorting, 85-96
specifying output to, 56
splitting, 49
standard data types, 41
structured, 92-96
subarrays as no-copy views, 46
summing values in, 59
universal functions, 50-58
arrows, 272-275
asfreq() method, 197-199
asterisk (*), 7
automagic function, 19
axes limits, 228-230

B

bagging, 426
bandwidth (see kernel bandwidth)
bar (|) operator, 77
bar plots, 321
Basemap toolkit
geographic data with, 298
(see also geographic data)
installation, 298
basis function regression, 378, 392-396
Gaussian basis functions, 394-396
polynomial basis functions, 393
Bayesian classification, 383, 501-506
(see also naive Bayes classification)
Bayesian information criterion (BIC), 487
Bayesian Methods for Hackers stylesheet, 288
Bayess theorem, 383
bias–variance trade-off
kernel bandwidth and, 497
model selection and, 364-366
bicycle traffic prediction
linear regression, 400
time series, 202-209
big-O notation, 92
binary ufuncs, 52
binnings, 248
bitwise logic operators, 74
bogosort, 86
Bokeh, 330
Boolean arrays
Boolean operators and, 74
counting entries in, 73
working with, 73-75
Boolean masks, 70-78
Boolean arrays as, 75-78
rainfall statistics, 70
working with Boolean arrays, 73-75
Boolean operators, 74
broadcasting, 63-69
adding two-dimensional array to one-dimensional array, 66
basics, 63-65
centering an array, 68
defined, 58, 63
in practice, 68
plotting two-dimensional function, 69
rules, 65-68
two compatible arrays, 66
two incompatible arrays, 67

C

categorical data, 376
class labels (for data point), 334
classification task
defined, 332
machine learning, 333-335
clustering, 332
basics, 338-339
GMMs, 353, 476-491
k-means, 339, 462-476
code
magic commands for determining execution time, 12
magic commands for pasting blocks, 11
magic commands for running external, 12
profiling and timing, 25-30
timing of snippets, 25-27
coefficient of determination, 365

colon (:), 44
color compression, 473-476
colorbars
  colormap selection, 256-259
  customizing, 255-262
  discrete, 260
  handwritten digit example, 261-262
colormap, 256-259
column(s)
  accessing single, 45
  indexing, 163
  MultiIndex for, 133
  sorting arrays along, 87
  suffixes keyword and overlapping names, 153
column-wise operations, 211-213
command history shortcuts, 9
comparison operators, 71-73
concatenation
  datasets, 141-146
  of arrays, 48, 142
  with pd.concat(), 142-146
confusion matrix, 357
conic projections, 303
contour plots, 241-245
  density and, 241-245
  three-dimensional function, 241-245
  three-dimensional plot, 292
Conway, Drew, xi
cross-validation, 361-370
cubehelix colormap, 258
cylindrical projections, 301

D

data
  as arrays, 33
  missing (see missing data)
data representation (Scikit-Learn package), 343-346
  data as table, 343
  features matrix, 344
  target array, 344-345
data science, defining, xi
data types, 34
  fixed-type arrays, 38
  integers, 35
  lists in, 37-41
  NumPy, 41
DataFrame object (Pandas), 102-105
  as dictionary, 110-112
  as generalized NumPy array, 102
  as specialized dictionary, 103
  as two-dimensional array, 112-114
  constructing, 104
  data selection in, 110
  defined, 97
  index alignment in, 117
  masking, 114
  multiply indexed, 136
  operations between Series object and, 118
  slicing, 114
DataFrame.eval() method, 211-213
  assignment in, 212
  local variables in, 213
DataFrame.query() method, 213
datasets
  appending, 146
  combining (Panda), 141-158
  concatenation, 141-146
  merging/joining, 146-158
datetime module, 189
datetime64 dtype, 189
dateutil module, 189
debugging, 22-24
decision trees, 421-426
  (see also random forests)
  creating, 422-425
  overfitting, 425
deep learning, 513
density estimator
  GMM, 484-488
  histogram as, 492
  KDE (see kernel density estimation (KDE))
describe() method, 164
development, IPython
  profiling and timing code, 25-30
  profiling full scripts, 27
  timing of code snippets, 25-27
dictionary(-ies)
  DataFrame as specialization of, 103
  DataFrame object constructed from list of, 104
  Pandas Series object vs., 100
digits, recognition of (see optical character recognition)
dimensionality reduction, 261
  machine learning, 340-342
  PCA and, 433

discriminative classification, 405-407
documentation, accessing
  IPython, 3-8, 98
  Pandas, 98
double question mark (??), 5
dropna() method, 125
dynamic typing, 34

E

eigenfaces, 442-445
ensemble estimator/method, 421
  (see also random forests)
ensemble learner, 421
equidistant cylindrical projection, 301
errors, visualizing
  basic errorbars, 238
  continuous quantities, 239
  Matplotlib, 237-240
Estimator API, 346-359
  basics, 347
  Iris classification example, 351
  Iris clustering example, 353
  Iris dimensionality example, 352
  simple linear regression example, 347-354
eval() function, 210-211
  DataFrame.eval() method and, 211-213
  pd.eval() function and, 210-211
  when to use, 214
exceptions, controlling, 20-22
expectation-maximization (E-M) algorithm
  caveats, 467-470
  GMM as generalization of, 480-484
  k-means clustering and, 465-476
exponentials, 55
external code, magic commands for running, 12

F

face recognition
  HOG, 506-514
  Isomap, 456-460
  PCA, 442-445
  SVMs, 416-420
faceted histograms, 318
factor plots, 319
fancy indexing, 78-85
  basics, 79
  binning data, 83
  combined with other indexing schemes, 80
  modifying values with, 82
  selection of random points, 81
feature engineering, 375-382
  categorical features, 376
  derived features, 378-380
  image features, 378
  imputation of missing data, 381
  processing pipeline, 381
  text features, 377
feature, data point, 334
features matrix, 344
fillna() method, 126
filter() method, 166
FiveThirtyEight stylesheet, 287
fixed-type arrays, 38

G

Gaussian basis functions, 394-396
Gaussian mixture models (GMMs), 476-491
  choosing covariance type, 484
  clustering with, 353
  density estimation algorithm, 484-488
  E–M generalization, 480-484
  handwritten data generation example, 488-491
  k-means weaknesses addressed by, 477-480
  KDE and, 491
Gaussian naive Bayes classification, 351, 357, 383-386, 510
Gaussian process regression (GPR), 239
generative models, 383
geographic data, 298
  Basemap toolkit for, 298
  California city population example, 308
  drawing a map background, 304-307
  map projections, 300-304
  plotting data on maps, 307
  surface temperature data example, 309
get() operation, 183
get_dummies() method, 183
ggplot stylesheet, 287
graphics libraries, 330
GroupBy aggregation, 170
GroupBy object, 163-165
  aggregate() method, 166
  apply() method, 167
  column indexing, 163
  dispatch methods, 164
  filter() method, 166

iteration over groups, 164
transform() method, 167
groupby() operation (Pandas), 161-170
GroupBy object and, 163-165
grouping example, 169
pivot tables vs., 171
split key specification, 168
split-apply-combine example, 161-163

H

handwritten digits, recognition of (see optical character recognition)
hard negative mining, 513
help
IPython, 3-8
magic functions, 13
help() function, 4
hexagonal binnings, 248
hierarchical indexing
in one-dimensional Series, 128-141
MultiIndex, 128-141, 129-131
(see also MultiIndex type)
rearranging multi-indices, 137-140
unstack() method, 130
with Python tuples as keys, 128
Histogram of Oriented Gradients (HOG)
caveats and improvements, 512-514
features, 506
for face detection pipeline, 506-514
simple face detector, 507-512
histograms, 245-249
binning data to create, 83
faceted, 318
KDE and, 248, 491-496
manual customization, 282-284
plt.hexbin() function, 248
plt.hist2d() function, 247
Seaborn, 314-317
simple, 245-246
two-dimensional, 247-249
holdout sets, 360
Hunter, John, 217
hyperparameters, 349
(see also model validation)

I

iloc attribute (Pandas), 110
images, encoding for machine learning analysis, 378
immutable array, Index object as, 106
importing, tab completion for, 7
In objects, IPython, 13
index alignment
in DataFrame, 117
in Series, 116
Index object (Pandas), 105-107
as immutable array, 106
as ordered set, 106
indexing
fancy, 78-85
(see also fancy indexing)
hierarchical (see hierarchical indexing)
NumPy arrays: accessing single elements, 43
Pandas, 107
IndexSlice object, 137
indicator variables, 183
inner join, 153
input/output history, IPython, 13-16
In and Out objects, 13
related magic commands, 16
suppressing output, 15
underscore shortcuts and previous outputs, 15
installation, Python, xiv
integers, Python, 35
IPython, 1
accessing documentation with ?, 3
accessing source code with ??, 5
command-line commands in shell, 18
controlling exceptions, 20-22
debugging, 22-24
documentation, 3-8, 34
errors handling, 20-24
exploring modules with tab completion, 6-7
help and documentation, 3-8
input/output history, 13-16
keyboard shortcuts in shell, 8
launching Jupyter notebook, 2
launching shell, 2
magic commands, 10-13
notebook (see Jupyter notebook)
plotting from shell, 219
profiling and timing code, 25-30
shell commands, 16-19
shell-related magic commands, 19
web resources, 30
wildcard matching, 7
Iris dataset

as table, 343
classification, 351
clustering, 353
dimensionality, 352
pair plots, 317
scatter plots, 236
visualization of, 345
isnull() method, 124
Isomap
dimensionality reduction, 341, 355
face data, 456-460
ix attribute (Pandas), 110

J

jet colormap, 257
joins, 145
(see also merging)
categories of, 147-149
datasets, 146-158
many-to-one, 148
one-to-one, 147
set arithmetic for, 152
joint distributions, 316, 320
Jupyter notebook
launching, 2
plotting from, 220

K

k-means clustering, 339, 462-476
basics, 463-465
color compression example, 473-476
expectation-maximization algorithm, 465-476
GMM as means of addressing weaknesses of, 477-480
simple digits data application, 470-473
kernel (defined), 496
kernel bandwidth
defined, 496
selection via cross-validation, 497
kernel density estimation (KDE), 491-506
bandwidth selection via cross-validation, 497
Bayesian generative classification with, 501-506
custom estimator, 501-506
histograms and, 491-496
in practice, 496-506
Matplotlib, 248
Seaborn, 314
visualization of geographic distributions, 498-501
kernel SVM, 411-414
kernel transformation, 413
kernel trick, 413
keyboard shortcuts, IPython shell, 8
command history, 9
navigation, 8
text entry, 9
Knuth, Donald, 25

L

labels/labeling
classification task, 333-335
clustering, 338-339
dimensionality reduction and, 340-342
regression task, 335-338
simple line plots, 230-232
Lambert conformal conic projection, 303
lasso regularization (L1 regularization), 399
learning curves, computing, 372
left join, 153
left_index keyword, 151-152
legends, plot
choosing elements for, 251
customizing, 249-255
multiple legends on same axes, 254
point size, 252
levels, naming, 133
line plots
axes limits for, 228-230
labeling, 230-232
line colors and styles, 226-228
Matplotlib, 224-232
line-by-line profiling, 28
linear regression (in machine learning), 390
basis function regression, 392-396
regularization, 396-400
Seattle bicycle traffic prediction example, 400
simple, 390-392
lists, Python, 37-41
loc attribute (Pandas), 110
locally linear embedding (LLE), 453-455
logarithms, 55

M

machine learning, 331

basics, 331-342
categories of, 332
classification task, 333-335
clustering, 338-339
decision trees and random forests, 421
defined, 332
dimensionality reduction, 340-342
educational resources, 514
face detection pipeline, 506-514
feature engineering, 375-382
GMM (see Gaussian mixture models)
hyperparameters and model validation, 359-375
KDE (see kernel density estimation)
linear regression (see linear regression)
manifold learning (see manifold learning)
naive Bayes classification, 382-390
PCA (see principal component analysis)
qualitative examples, 333-342
regression task, 335-338
Scikit-Learn basics, 343
supervised, 332
SVMs (see support vector machines)
unsupervised, 332
magic commands
code block pasting, 11
code execution timing, 12
help commands, 13
IPython input/output history, 16
running external code, 12
shell-related, 19
manifold learning, 445-462
"HELLO" function, 446
advantages/disadvantages, 455
applying Isomap on faces data, 456-460
defined, 446
k-means clustering (see k-means clustering)
multidimensional scaling, 450-452
PCA vs., 455
visualizing structure in digits, 460-462
many-to-one joins, 148
map projections, 300-304
conic, 303
cylindrical, 301
perspective, 302
pseudo-cylindrical, 302
maps, geographic (see geographic data)
margins, maximizing, 407-416
masking, 114
(see also Boolean masks)
Boolean arrays, 75-78
Boolean masks, 70-78
MATLAB-style interface, 222
Matplotlib, 217, 329
axes limits for line plots, 228-230
changing defaults via rcParams, 284
colorbar customization, 255-262
configurations and stylesheets, 282-290
density and contour plots, 241-245
error visualization, 237-240
general tips, 218-222
geographic data with Basemap toolkit, 298
gotchas, 232
histograms, binnings, and density, 245-249
importing, 218
interfaces, 222
labeling simple line plots, 230-232
line colors and styles, 226-228
MATLAB-style interfaces, 222
multiple subplots, 262-268
object hierarchy of plots, 275
object-oriented interfaces, 223
plot customization, 282-284
plot display contexts, 218-220
plot legend customization, 249-255
plotting from a script, 219
plotting from IPython notebook, 220
plotting from IPython shell, 219
resources and documentation for, 329
saving figures to file, 221
Seaborn vs., 311-313
setting styles, 218
simple line plots, 224-232
stylesheets, 285-290
text and annotation, 268-275
three-dimensional function visualization, 241-245
three-dimensional plotting, 290-298
tick customization, 275-282
max() function, 59
maximum margin estimator, 408
(see also support vector machines (SVMs))
memory use, profiling, 29
merge key
on keyword, 149
specification of, 149-152
merging, 146-158
(see also joins)

key specification, 149-152
relational algebra and, 146
US state population data example, 154-158
min() function, 59
Miniconda, xiv
missing data, 120-124
feature engineering and, 381
handling, 119-120
NaN and None, 123
operating on null values in Pandas, 124-127
Möbius strip, 296-298
model (defined), 334
model parameters (defined), 334
model selection
bias–variance trade-off, 364-366
validation curves in Scikit-Learn, 366-370
model validation, 359-375
bias–variance trade-off, 364-366
cross-validation, 361-370
grid search example, 373
holdout sets, 360
learning curves, 370-373
naive approach to, 359
validation curves, 366-370
modules, IPython, 6-7
Mollweide projection, 302
multi-indexing (see hierarchical indexing)
multidimensional scaling (MDS), 450-452
basics, 447-450
locally linear embedding and, 453-455
nonlinear embeddings, 452
MultiIndex type, 129-131
creation methods, 131-134
data aggregations on, 140
explicit constructors for, 132
extra dimension of data with, 130
for columns, 133
index setting/resetting, 139
indexing and slicing, 134-137
keys option, 144
level names, 133
multiply indexed DataFrames, 136
multiply indexed Series, 134
rearranging, 137-140
sorted/unsorted indices with, 137
stacking/unstacking indices, 138
multinomial naive Bayes classification, 386-389

N

naive Bayes classification, 382-390
advantages/disadvantages, 389
Bayesian classification and, 383
Gaussian, 383-386
multinomial, 386-389
text classification example, 386-389
NaN value, 104, 116, 122
navigation shortcuts, 8
neural networks, 513
noise filter, PCA as, 440-442
None object, 121, 123
nonlinear embeddings, MDS and, 452
notnull() method, 124
np.argsort() function, 86
np.concatenate() function, 48, 143
np.sort() function, 86
null values, 124-127
detecting, 124
dropping, 125
filling, 126
NumPy, 33
aggregations, 58-63
array attributes, 42
array basics, 42
array indexing: accessing single elements, 43
array slicing: accessing subarrays, 44
Boolean masks, 70-78
broadcasting, 63-69
comparison operators as ufuncs, 71-73
computation on arrays, 50-58
data types in Python, 34
datetime64 dtype, 189
documentation, 34
fancy indexing, 78-85
keywords and/or vs. operators &/|, 77
sorting arrays, 85-92
standard data types, 41
structured arrays, 92-96
universal functions, 50-58

O

object-oriented interface, 223
offsets, time series, 196
on keyword, 149
one-hot encoding, 376
one-to-one joins, 147
optical character recognition
digit classification, 357-358

GMMs, 488-491
k-means clustering, 470-473
loading/visualizing digits data, 354
Matplotlib, 261-262
PCA as noise filtering, 440-442
PCA for visualization, 437
random forests for classifying digits, 430-432
Scikit-Learn application, 354-358
visualizing structure in digits, 460-462
or keyword, 77
ordered set, Index object as, 106
orthographic projection, 302
Out objects, IPython, 13
outer join, 153
outer products, 58
outliers, PCA and, 445
output, suppressing, 15
overfitting, 371, 425

P

pair plots, 317
Pandas, 97
aggregation and grouping, 158-170
and compound expressions, 209
appending datasets, 146
built-in documentation, 98
combining datasets, 141-158
concatenation of datasets, 141-146
data indexing and selection, 107
data selection in DataFrame, 110-215
data selection in Series, 107-110
DataFrame object, 102-105
eval() and query(), 208-209
handling missing data, 119-120
hierarchical indexing, 128-141
Index object, 105-107
installation, 97
merging/joining datasets, 146-158
NaN and None in, 123
null values, 124-127
objects, 98-107
operating on data in, 115-127
(see also universal functions)
pandas.eval(), 210-211
Panel data, 141
pivot tables, 170-178
Series object, 99-102
time series, 188-214
vectorized string operations, 178-188
pandas.eval() function, 210-211
Panel data, 141
partial slicing, 135
partitioning (partial sorts), 88
pasting code blocks, magic commands for, 11
pd.concat() function
catching repeats as error, 144
concatenation with, 142-146
concatenation with joins, 145
duplicate indices, 143
ignoring the index, 144
MultiIndex keys, 144
pd.date_range() function, 193
pd.eval() function, 210-211
pd.merge() function, 146-158
categories of joins, 147-149
keywords, 149-152
left_index/right_index keywords, 151-152
merge key specification, 149-152
relational algebra and, 146
specifying set arithmetic for joins, 152
pdb (Python debugger), 22
Perez, Fernando, 1, 217
Period type, 193
perspective projections, 302
pipelines, 366, 381
pivot tables, 170-178
groupby() operation vs., 171
multi-level, 172
syntax, 171-173
Titanic passengers example, 170
US birthrate data example, 174-178
Planets dataset
aggregation and grouping, 159
bar plots, 321
plot legends
choosing elements for, 251
customizing, 249-255
multiple legends on same axes, 254
points size, 252
Plotly, 330
plotting
axes limits for simple line plots, 228-230
bar plots, 321
changing defaults via rcParams, 284
colorbars, 255-262
data on maps, 307-329
density and contour plots, 241-245

display contexts, 218-220
factor plots, 319
from an IPython shell, 219
from script, 219
histograms, binnings, and density, 245-249
IPython notebook, 220
joint distributions, 320
labeling simple line plots, 230-232
line colors and styles, 226-228
manual customization, 282-284
Matplotlib, 217
multiple subplots, 262-268
of errors, 237-240
pair plots, 317
plot legends, 249-255
Seaborn, 311-313
simple line plots, 224-232
simple scatter plots, 233-237
stylesheets for, 285-290
text and annotation for, 268-275
three-dimensional, 290-298
three-dimensional function, 241-245
ticks, 275-282
two-dimensional function, 69
various Python graphics libraries, 330
plt.axes() function, 263-264
plt.contour() function, 241-244
plt.GridSpec() function, 266-268
plt.imshow() function, 243-244
plt.legend() command, 249-254
plt.plot() function
color arguments, 226
plt.scatter vs., 237
scatter plots with, 233-235
plt.scatter() function
plt.plot vs., 237
simple scatter plots with, 235-237
plt.subplot() function, 264
plt.subplots() function, 265
polynomial basis functions, 393
polynomial regression model, 366
pop() method, 111
population data, US, merge and join operations with, 154-158
principal axes, 434-436
principal component analysis (PCA), 433-515
basics, 433-442
choosing number of components, 440
eigenfaces example, 442-445
facial recognition example, 442-445
for dimensionality reduction, 436
handwritten digit example, 437-440, 440-442
manifold learning vs., 455
meaning of components, 438-439
noise filtering, 440-442
strengths/weaknesses, 445
visualization with, 437
profiling
full scripts, 27
line-by-line, 28
memory use, 29
projections (see map projections)
pseudo-cylindrical projections, 302
Python
installation considerations, xiv
Python 2.x vs. Python 3, xiii
reasons for using, xii

Q

query() method
DataFrame.query() method, 213
when to use, 214
question mark (?), accessing IPython documentation with, 3
quicksort algorithm, 87

R

radial basis function, 412
rainfall statistics, 70
random forests
advantages/disadvantages, 432
classifying digits with, 430-432
defined, 426
ensembles of estimators, 426-428
motivating with decision trees, 421-426
regression, 428
RandomizedPCA, 442
rcParams dictionary, changing defaults via, 284
RdBu colormap, 258
record arrays, 96
reduce() method, 57
regression, 428-433
(see also specific forms, e.g.: linear regression)
regression task
defined, 332
machine learning, 335-338

regular expressions, 181
regularization, 396-400
  lasso regularization, 399
  ridge regression, 398
relational algebra, 146
resample() method, 197-199
reset_index() method, 139
reshaping, 47
ridge regression (L2 regularization), 398
right join, 153
right_index keyword, 151-152
rolling statistics, 201
runtime configuration (rc), 284

S

scatter plots (see simple scatter plots)
Scikit-Learn package, 331, 343-346
  API (see Estimator API)
  basics, 343-359
  data as table, 343
  data representation in, 343-346
  Estimator API, 346-354
  features matrix, 344
  handwritten digit application, 354-358
  support vector classifier, 408-411
  target array, 344-345
scipy.special submodule, 56
script
  plotting from, 219
  profiling, 27
Seaborn
  bar plots, 321
  datasets and plot types, 313-329
  faceted histograms, 318
  factor plots, 319
  histograms, KDE, and densities, 314-317
  joint distributions, 320
  marathon finishing times example, 322-329
  Matplotlib vs., 311-313
  pair plots, 317
  stylesheet, 289
  visualization with, 311-313
Seattle, bicycle traffic prediction in
  linear regression, 400-405
  time series, 202-209
Seattle, rainfall statistics in, 70
semi-supervised learning, 333
Series object (Pandas), 99-102
  as dictionary, 100, 107
  constructing, 101
  data indexing/selection in, 107-110
  DataFrame as dictionary of, 110-112
  DataFrame object constructed from, 104
  DataFrame object constructed from dictionary of, 105
  generalized NumPy array, 99
  hierarchical indexing in, 128-141
  index alignment in, 116
  indexer attributes, 109
  multiply indexed, 134
  one-dimensional array, 108
  operations between DataFrame and, 118
shell, IPython
  basics, 16
  command-line commands, 18
  commands, 16-19
  keyboard shortcuts in, 8
  launching, 2
  magic commands, 19
  passing values to and from, 18
shift() function, 199-201
shortcuts
  accessing previous output, 15
  command history, 9
  IPython shell, 8-31
  navigation, 8
  text entry, 9
simple histograms, 245-246
simple line plots
  axes limits for, 228-230
  labeling, 230-232
  line colors and styles, 226-228
  Matplotlib, 224-232
  simple (Matplotlib), 224-232
simple linear regression, 390-392
simple scatter plots
  California city populations, 249-254
  Matplotlib, 233-237
  plt.plot, 233-235
  plt.plot vs. plt.scatter, 237
  plt.scatter, 235-237
slice() operation, 183
slicing
  MultiIndex with sorted/unsorted indices, 137
  NumPy arrays, 44-47
  NumPy arrays: accessing subarrays, 44

NumPy arrays: multidimensional subarrays, 45
NumPy arrays: one-dimensional subarrays, 44
NumPy vs. Python, 46
Pandas conventions, 114
sorting arrays, 85-92
along rows or columns, 87
basics, 85
fast sorting with np.sort and np.argsort, 86
k-nearest neighbors example, 88-92
partitioning, 88
source code, accessing, 5
splitting arrays, 49
string operations (see vectorized string operations)
structured arrays, 92-96
advanced compound types, 95
creating, 94
record arrays, 96
stylesheets
Bayesian Methods for Hackers, 288
default style, 286
FiveThirtyEight style, 287
ggplot, 287
Matplotlib, 285-290
Seaborn, 289
subarrays
as no-copy views, 46
creating copies, 46
slicing multidimensional, 45
slicing one-dimensional, 44
subplots
manual customization, 263-264
multiple, 262-268
plt.axes() for, 263-264
plt.GridSpec() for, 266-268
plt.subplot() for, 264
plt.subplots() for, 265
subsets, faceted histograms, 318
suffixes keyword, 153
supervised learning, 332
classification task, 333-335
regression task, 335-338
support vector (defined), 409
support vector classifier, 408-411
support vector machines (SVMs), 405
advantages/disadvantages, 420
face recognition example, 416-420
fitting, 408-411
kernels and, 411-414
maximizing the margin, 407-416
motivating, 405-420
simple face detector, 507
softening margins, 414-416
surface plots, three-dimensional, 293-298

T

t-distributed stochastic neighbor embedding (t-SNE), 456, 472
tab completion
exploring IPython modules with, 6-7
of object contents, 6
when importing, 7
table, data as, 343
target array, 344-345
term frequency-inverse document frequency (TF-IDF), 378
text, 377
(see also annotation of plots)
transforms and position of, 270-272
text entry shortcuts, 9
three-dimensional plotting
contour plots, 292
Möbius strip visualization, 296-298
points and lines, 291
surface plots, 293-298
surface triangulations, 295-298
wireframes, 293
with Matplotlib, 290-298
ticks (tick marks)
customizing, 275-282
fancy formats, 279-281
formatter/locator options, 281
major and minor, 276
reducing/increasing number of, 278
Tikhonov regularization, 398
time series
bar plots, 321
dates and times in Pandas, 191
datetime64, 189
frequency codes, 195
indexing data by timestamps, 192
native Python dates and times, 189
offsets, 196
Pandas, 188-209
Pandas data structures for, 192-194
pd.date_range(), 193

Python vs. Pandas, 188-192
resampling and converting frequencies, 197-199
rolling statistics, 201
Seattle bicycle counts example, 202-209
time-shifts, 199-201
typed arrays, 189
Timedelta type, 193
Timestamp type, 193
timestamps, indexing data by, 192
timing, of code, 12, 25-27
transform() method, 167
transforms
modifying, 270-272
text position and, 270-272
triangulated surface plots, 295-298
trigonometric functions, 54
tshift() function, 199-201
two-fold cross-validation, 361

U

ufuncs (see universal functions)
unary ufuncs, 52
underfitting, 364, 371
underscore (_) shortcut, 15
universal functions (ufuncs), 50-58
absolute value, 54
advanced features, 56
aggregates, 57
array arithmetic, 52
basics, 51
comparison operators as, 71-73
exponentials, 55
index alignment, 116-118
index preservation, 115
logarithms, 55
operating on data in Pandas, 115-127
operations between DataFrame and Series, 118
outer products, 58
slowness of Python loops, 50
specialized ufuncs, 56
specifying output, 56
trigonometric functions, 54
unstack() method, 130
unsupervised learning
clustering, 338-339, 353
defined, 332
dimensionality reduction, 261, 340-342, 352, 355
PCA (see principal component analysis)

V

validation (see model validation)
validation curves, 366-370
variables
dynamic typing, 34
passing to and from shell, 18
variance, in bias–variance trade-off, 364-366
vectorized operations, 63
vectorized string operations, 178-188
basics, 178
indicator variables, 183
methods similar to Python string methods, 180
methods using regular expressions, 181
recipe database example, 184-188
tables of, 180-184
vectorized item access and slicing, 183
Vega/Vega-Lite, 330
violin plot, 327
viridis colormap, 258
Vispy, 330
visualization software (see Matplotlib) (see Seaborn)

W

Wickham, Hadley, 161
wildcard matching, 7
wireframe plot, 293
word counts, 377-378

About the Author

Jake VanderPlas is a long-time user and developer of the Python scientific stack. He currently works as an interdisciplinary research director at the University of Washington, conducts his own astronomy research, and spends time advising and consulting with local scientists from a wide range of fields.

Colophon

The animal on the cover of *Python Data Science Handbook* is a Mexican beaded lizard (*Heloderma horridum*), a reptile found in Mexico and parts of Guatemala. It and the Gila monster (a close relative) are the only venomous lizards in the world. This animal primarily feeds on eggs, however, so the venom is used as a defense mechanism. When it feels threatened, the lizard will bite—and because it cannot release a large quantity of venom at once, it firmly clamps its jaws and uses a chewing motion to move the toxin deeper into the wound. This bite and the aftereffects of the venom are extremely painful, though rarely fatal to humans.

The Greek word *heloderma* translates to "studded skin," referring to the distinctive beaded texture of the reptile's skin. These bumps are osteoderms, which each contain a small piece of bone and serve as protective armor. The Mexican beaded lizard is black with yellow patches and bands. It has a broad head and a thick tail that stores fat to help the animal survive during the hot summer months when it is inactive. On average, these lizards are 22–36 inches long, and weigh around 1.8 pounds.

As with most snakes and lizards, the tongue of the Mexican beaded lizard is its primary sensory organ. It will flick it out repeatedly to gather scent particles from the environment and detect prey (or, during mating season, a potential partner). When the forked tongue is retracted into the mouth, it touches the Jacobson's organ, a patch of sensory cells that identify various chemicals and pheromones.

The beaded lizard's venom contains enzymes that have been synthesized to help treat diabetes, and further pharmacological research is in progress. It is endangered by loss of habitat, poaching for the pet trade, and being killed by locals who are simply afraid of it. This animal is protected by legislation in both countries where it lives.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from Wood's *Animate Creation*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.