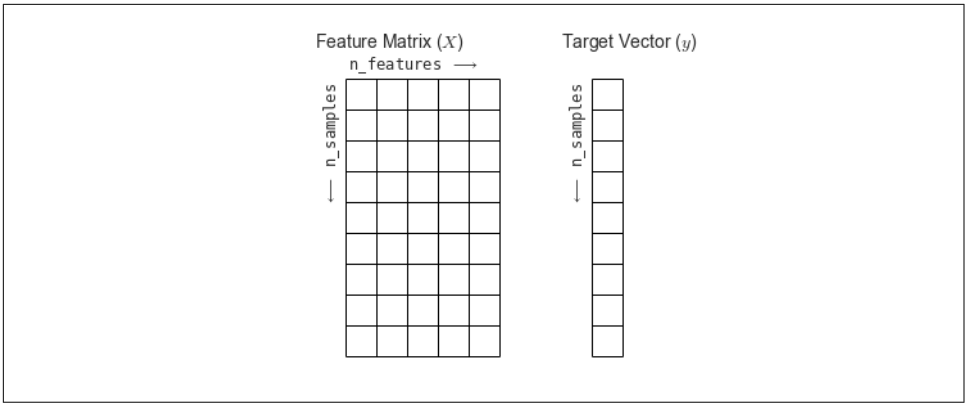

Figure 5-13. Scikit-Learn's data layout

With this data properly formatted, we can move on to consider the *estimator* API of Scikit-Learn.

Scikit-Learn's Estimator API

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

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

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

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

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

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

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

Basics of the API

Most commonly, the steps in using the Scikit-Learn estimator API are as follows (we will step through a handful of detailed examples in the sections that follow):

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

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

Supervised learning example: Simple linear regression

As an example of this process, let's consider a simple linear regression—that is, the common case of fitting a line to (x, y) data. We will use the following simple data for our regression example (Figure 5-14):

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

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

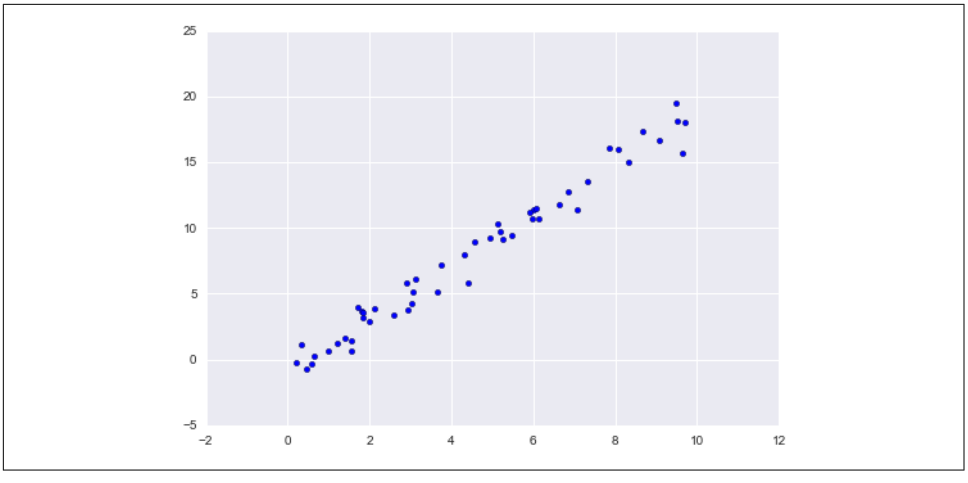

Figure 5-14. Data for linear regression

With this data in place, we can use the recipe outlined earlier. Let's walk through the process:

1. Choose a class of model.

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

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

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

2. Choose model hyperparameters.

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

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

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

These are examples of the important choices that must be made *once the model class is selected*. These choices are often represented as *hyperparameters*, or parameters that must be set before the model is fit to data. In Scikit-Learn, we choose hyperparameters by passing values at model instantiation. We will explore how you can quantitatively motivate the choice of hyperparameters in "Hyperparameters and Model Validation" on page 359.

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

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

Out[7]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
                         normalize=False)
```

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

3. Arrange data into a features matrix and target vector.

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

   ```
   In[8]: X = x[:, np.newaxis]
          X.shape

   Out[8]: (50, 1)
   ```

4. Fit the model to your data.

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

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

   Out[9]:
   LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
                    normalize=False)
   ```

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

```
In[10]: model.coef_

Out[10]: array([ 1.9776566])

In[11]: model.intercept_

Out[11]: -0.90331072553111635
```

These two parameters represent the slope and intercept of the simple linear fit to the data. Comparing to the data definition, we see that they are very close to the input slope of 2 and intercept of –1.

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

5. Predict labels for unknown data.

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

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

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

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

   Finally, let's visualize the results by plotting first the raw data, and then this model fit (Figure 5-15):

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

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

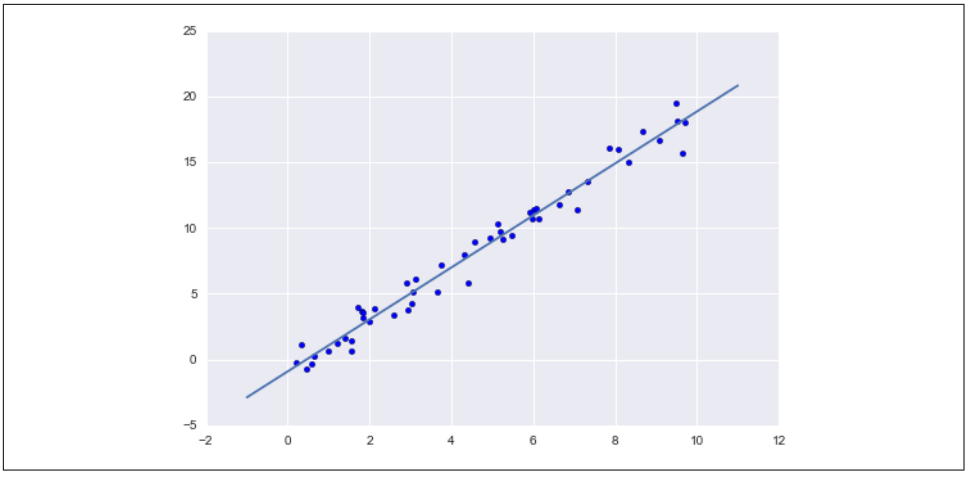

Figure 5-15. A simple linear regression fit to the data

Supervised learning example: Iris classification

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

For this task, we will use an extremely simple generative model known as Gaussian naive Bayes, which proceeds by assuming each class is drawn from an axis-aligned Gaussian distribution (see "In Depth: Naive Bayes Classification" on page 382 for more details). Because it is so fast and has no hyperparameters to choose, Gaussian naive Bayes is often a good model to use as a baseline classification, before you explore whether improvements can be found through more sophisticated models.

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

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

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

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

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

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

Out[17]: 0.97368421052631582
```

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

Unsupervised learning example: Iris dimensionality

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

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

Here we will use principal component analysis (PCA; see "In Depth: Principal Component Analysis" on page 433), which is a fast linear dimensionality reduction technique. We will ask the model to return two components—that is, a two-dimensional representation of the data.

Following the sequence of steps outlined earlier, we have:

```
In[18]:
from sklearn.decomposition import PCA  # 1. Choose the model class
model = PCA(n_components=2)      # 2. Instantiate the model with hyperparameters
model.fit(X_iris)                # 3. Fit to data. Notice y is not specified!
X_2D = model.transform(X_iris)   # 4. Transform the data to two dimensions
```

Now let's plot the results. A quick way to do this is to insert the results into the original Iris `DataFrame`, and use Seaborn's `lmplot` to show the results (Figure 5-16):

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

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

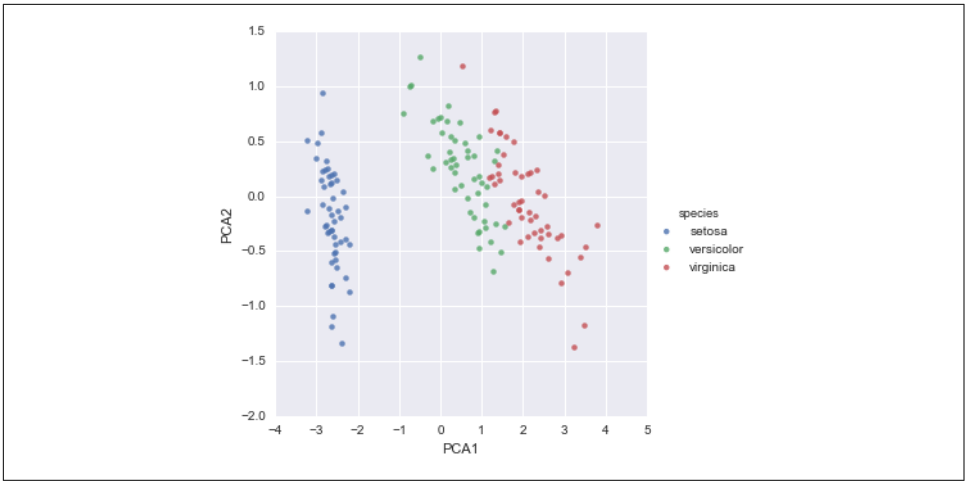

Figure 5-16. The Iris data projected to two dimensions

Unsupervised learning: Iris clustering

Let's next look at applying clustering to the Iris data. A clustering algorithm attempts to find distinct groups of data without reference to any labels. Here we will use a powerful clustering method called a Gaussian mixture model (GMM), discussed in more detail in "In Depth: Gaussian Mixture Models" on page 476. A GMM attempts to model the data as a collection of Gaussian blobs.

We can fit the Gaussian mixture model as follows:

```
In[20]:
from sklearn.mixture import GMM      # 1. Choose the model class
model = GMM(n_components=3,
            covariance_type='full')  # 2. Instantiate the model w/ hyperparameters
model.fit(X_iris)                    # 3. Fit to data. Notice y is not specified!
y_gmm = model.predict(X_iris)        # 4. Determine cluster labels
```

As before, we will add the cluster label to the Iris `DataFrame` and use Seaborn to plot the results (Figure 5-17):

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

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

clustering algorithm! This sort of algorithm might further give experts in the field clues as to the relationship between the samples they are observing.

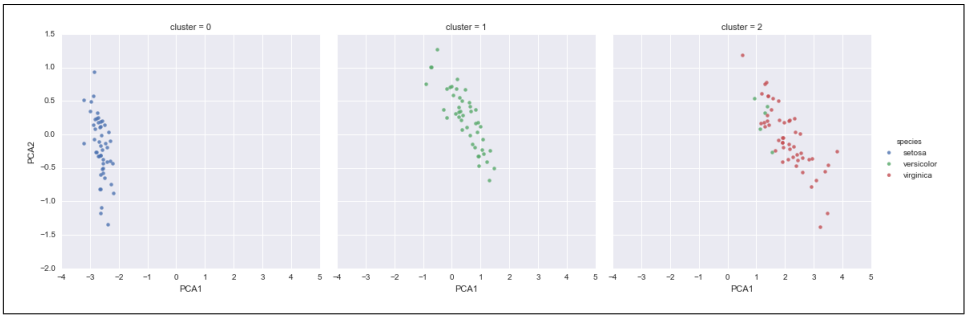

Figure 5-17. k-means clusters within the Iris data

Application: Exploring Handwritten Digits

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

Loading and visualizing the digits data

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

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

```
Out[22]: (1797, 8, 8)
```

The images data is a three-dimensional array: 1,797 samples, each consisting of an 8×8 grid of pixels. Let's visualize the first hundred of these (Figure 5-18):

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

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

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