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

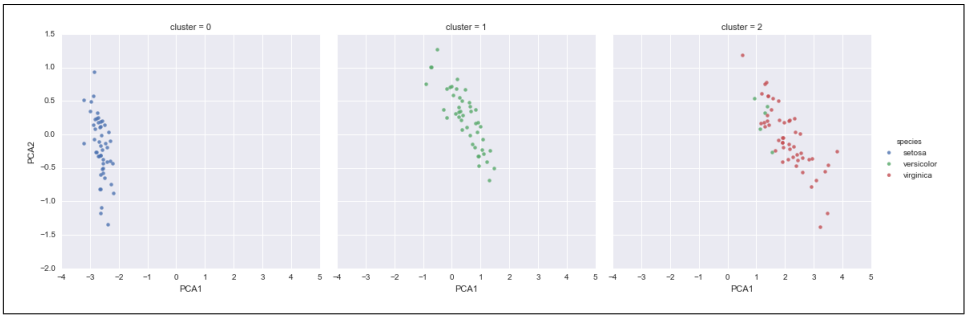

Figure 5-17. k-means clusters within the Iris data

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')
```

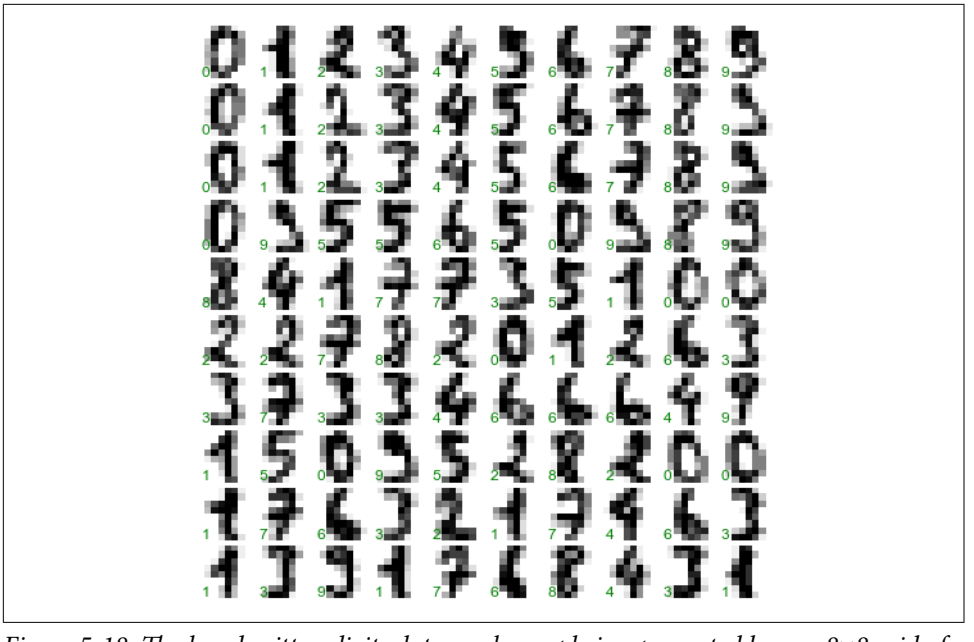

Figure 5-18. The handwritten digits data; each sample is represented by one 8×8 grid of pixels

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

```
In[24]: X = digits.data
        X.shape

Out[24]: (1797, 64)

In[25]: y = digits.target
        y.shape

Out[25]: (1797,)
```

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

Unsupervised learning: Dimensionality reduction

We'd like to visualize our points within the 64-dimensional parameter space, but it's difficult to effectively visualize points in such a high-dimensional space. Instead we'll reduce the dimensions to 2, using an unsupervised method. Here, we'll make use of a

manifold learning algorithm called *Isomap* (see "In-Depth: Manifold Learning" on page 445), and transform the data to two dimensions:

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

```
Out[26]: (1797, 2)
```

We see that the projected data is now two-dimensional. Let's plot this data to see if we can learn anything from its structure (Figure 5-19):

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

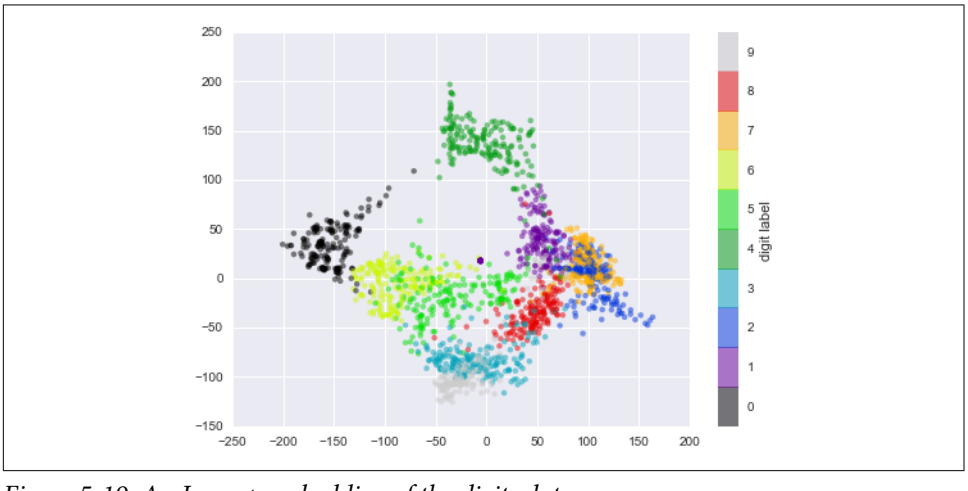

Figure 5-19. An Isomap embedding of the digits data

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

Overall, however, the different groups appear to be fairly well separated in the parameter space: this tells us that even a very straightforward supervised classification algorithm should perform suitably on this data. Let's give it a try.

Classification on digits

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

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

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

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

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

Out[30]: 0.83333333333333337
```

With even this extremely simple model, we find about 80% accuracy for classification of the digits! However, this single number doesn't tell us *where* we've gone wrong—one nice way to do this is to use the *confusion matrix*, which we can compute with Scikit-Learn and plot with Seaborn (Figure 5-20):

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

        mat = confusion_matrix(ytest, y_model)

        sns.heatmap(mat, square=True, annot=True, cbar=False)
        plt.xlabel('predicted value')
        plt.ylabel('true value');
```

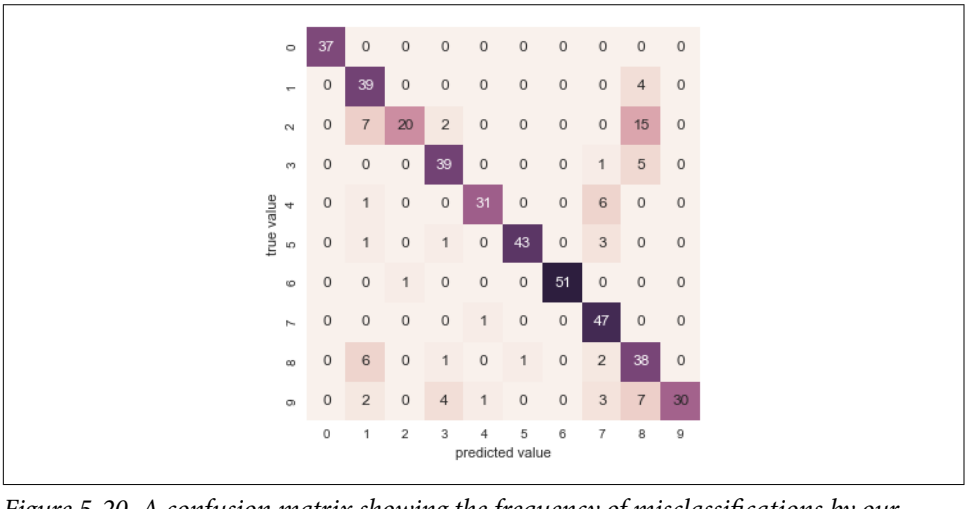

Figure 5-20. A confusion matrix showing the frequency of misclassifications by our classifier

This shows us where the mislabeled points tend to be: for example, a large number of twos here are misclassified as either ones or eights. Another way to gain intuition into the characteristics of the model is to plot the inputs again, with their predicted labels. We'll use green for correct labels, and red for incorrect labels (Figure 5-21):

```
In[32]: 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(y_model[i]),
                    transform=ax.transAxes,
                    color='green' if (ytest[i] == y_model[i]) else 'red')
```

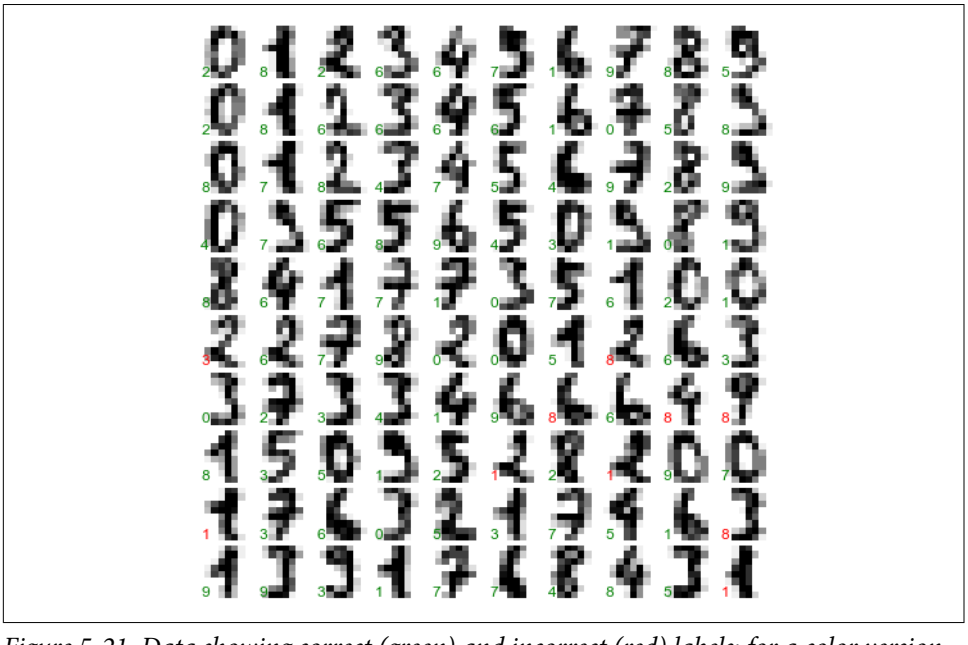

Figure 5-21. Data showing correct (green) and incorrect (red) labels; for a color version of this plot, see the online appendix

Examining this subset of the data, we can gain insight regarding where the algorithm might not be performing optimally. To go beyond our 80% classification rate, we might move to a more sophisticated algorithm, such as support vector machines (see "In-Depth: Support Vector Machines" on page 405) or random forests (see "In-Depth: Decision Trees and Random Forests" on page 421), or another classification approach.

Summary

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

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

Hyperparameters and Model Validation

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

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

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

Thinking About Model Validation

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

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

Model validation the wrong way

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

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