



Figure 1-3. Pair plot of the Iris dataset, colored by class label

From the plots, we can see that the three classes seem to be relatively well separated using the sepal and petal measurements. This means that a machine learning model will likely be able to learn to separate them.

Building Your First Model: k-Nearest Neighbors

Now we can start building the actual machine learning model. There are many classification algorithms in `scikit-learn` that we could use. Here we will use a *k*-nearest neighbors classifier, which is easy to understand. Building this model only consists of storing the training set. To make a prediction for a new data point, the algorithm finds the point in the training set that is closest to the new point. Then it assigns the label of this training point to the new data point.

The k in k -nearest neighbors signifies that instead of using only the closest neighbor to the new data point, we can consider any fixed number k of neighbors in the training (for example, the closest three or five neighbors). Then, we can make a prediction using the majority class among these neighbors. We will go into more detail about this in [Chapter 2](#); for now, we'll use only a single neighbor.

All machine learning models in `scikit-learn` are implemented in their own classes, which are called Estimator classes. The k -nearest neighbors classification algorithm is implemented in the `KNeighborsClassifier` class in the `neighbors` module. Before we can use the model, we need to instantiate the class into an object. This is when we will set any parameters of the model. The most important parameter of `KNeighborsClassifier` is the number of neighbors, which we will set to 1:

In[25]:

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
```

The `knn` object encapsulates the algorithm that will be used to build the model from the training data, as well the algorithm to make predictions on new data points. It will also hold the information that the algorithm has extracted from the training data. In the case of `KNeighborsClassifier`, it will just store the training set.

To build the model on the training set, we call the `fit` method of the `knn` object, which takes as arguments the NumPy array `X_train` containing the training data and the NumPy array `y_train` of the corresponding training labels:

In[26]:

```
knn.fit(X_train, y_train)
```

Out[26]:

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=1, n_neighbors=1, p=2,
                     weights='uniform')
```

The `fit` method returns the `knn` object itself (and modifies it in place), so we get a string representation of our classifier. The representation shows us which parameters were used in creating the model. Nearly all of them are the default values, but you can also find `n_neighbors=1`, which is the parameter that we passed. Most models in `scikit-learn` have many parameters, but the majority of them are either speed optimizations or for very special use cases. You don't have to worry about the other parameters shown in this representation. Printing a `scikit-learn` model can yield very long strings, but don't be intimidated by these. We will cover all the important parameters in [Chapter 2](#). In the remainder of this book, we will not show the output of `fit` because it doesn't contain any new information.

Making Predictions

We can now make predictions using this model on new data for which we might not know the correct labels. Imagine we found an iris in the wild with a sepal length of 5 cm, a sepal width of 2.9 cm, a petal length of 1 cm, and a petal width of 0.2 cm. What species of iris would this be? We can put this data into a NumPy array, again by calculating the shape—that is, the number of samples (1) multiplied by the number of features (4):

In[27]:

```
X_new = np.array([[5, 2.9, 1, 0.2]])
print("X_new.shape: {}".format(X_new.shape))
```

Out[27]:

```
X_new.shape: (1, 4)
```

Note that we made the measurements of this single flower into a row in a two-dimensional NumPy array, as `scikit-learn` always expects two-dimensional arrays for the data.

To make a prediction, we call the `predict` method of the `knn` object:

In[28]:

```
prediction = knn.predict(X_new)
print("Prediction: {}".format(prediction))
print("Predicted target name: {}".format(
    iris_dataset['target_names'][prediction]))
```

Out[28]:

```
Prediction: [0]
Predicted target name: ['setosa']
```

Our model predicts that this new iris belongs to the class 0, meaning its species is *setosa*. But how do we know whether we can trust our model? We don't know the correct species of this sample, which is the whole point of building the model!

Evaluating the Model

This is where the test set that we created earlier comes in. This data was not used to build the model, but we do know what the correct species is for each iris in the test set.

Therefore, we can make a prediction for each iris in the test data and compare it against its label (the known species). We can measure how well the model works by computing the *accuracy*, which is the fraction of flowers for which the right species was predicted: