CHAPTER

5

Getting Started with Scikit-learn for Machine Learning

Introduction to Scikit-learn

In Chapters 2–4, you learned how to use Python together with libraries such as NumPy and Pandas to perform number crunching, data visualization, and analysis. For machine learning, you can also use these libraries to build your own learning models. However, doing so would require you to have a strong appreciation of the mathematical foundation for the various machine learning algorithms—not a trivial matter.

Instead of implementing the various machine learning algorithms manually by hand, fortunately, someone else has already done the hard work for you. Introducing *Scikit-learn*, a Python library that implements the various types of machine learning algorithms, such as classification, regression, clustering, decision tree, and more. Using Scikit-learn, implementing machine learning is now simply a matter of calling a function with the appropriate data so that you can fit and train the model.

In this chapter, first you will learn the various venues where you can get the sample datasets to learn how to perform machine learning. You will then learn how to use Scikit-learn to perform simple linear regression on a simple dataset. Finally, you will learn how to perform data cleansing.

Python® Machine Learning, First Edition. Wei-Meng Lee.

Getting Datasets

Often, one of the challenges in machine learning is obtaining sample datasets for experimentation. In machine learning, when you are just getting started with an algorithm, it is often useful to get started with a simple dataset that you can create yourself to test that the algorithm is working correctly according to your understanding. Once you clear this stage, it is time to work with a large dataset, and for this you would need to find the relevant source so that your machine learning model can be as realistic as possible.

Here are some places where you can get the sample dataset to practice your machine learning:

- Scikit-learn's built-in dataset
- Kaggle dataset
- UCI (University of California, Irvine) Machine Learning Repository

Let's take a look at each of these in the following sections.

Using the Scikit-learn Dataset

Scikit-learn comes with a few standard sample datasets, which makes learning machine learning easy. To load the sample datasets, import the datasets module and load the desired dataset. For example, the following code snippets load the *Iris dataset*:

```
from sklearn import datasets
iris = datasets.load_iris()   # raw data of type Bunch
```

TIP **The Iris flower dataset or Fisher's Iris dataset is a multivariate dataset introduced by the British statistician and biologist Ronald Fisher. The dataset consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica, and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals in centimeters. Based on the combination of these four features, Fisher developed a linear discriminant model to distinguish the species from each other.**

The dataset loaded is represented as a `Bunch` object, a Python dictionary that provides attribute-style access. You can use the `DESCR` property to obtain a description of the dataset:

```
print(iris.DESCR)
```

More importantly, however, you can obtain the features of the dataset using the `data` property:

```
print(iris.data)                          # Features
```

The preceding statement prints the following:

```
[[ 5.1  3.5  1.4  0.2]
 [ 4.9  3.   1.4  0.2]
   ...
 [ 6.2  3.4  5.4  2.3]
 [ 5.9  3.   5.1  1.8]]
```

You can also use the `feature_names` property to print the names of the features:

```
print(iris.feature_names)      # Feature Names
```

The preceding statement prints the following:

```
['sepal length (cm)', 'sepal width (cm)',
 'petal length (cm)', 'petal width (cm)']
```

This means that the dataset contains four columns—sepal length, sepal width, petal length, and petal width. If you are wondering what a petal and sepal are, Figure 5.1 shows the Tetramerous flower of Ludwigia octovalvis showing petals and sepals (source: `https://en.wikipedia.org/wiki/Sepal`).

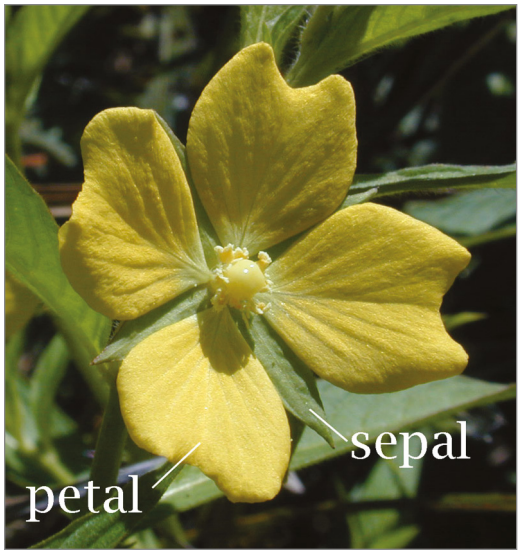

Figure 5.1: The petal and sepal of a flower

To print the label of the dataset, use the `target` property. For the label names, use the `target_names` property:

```
print(iris.target)                  # Labels
print(iris.target_names)            # Label names
```

This prints out the following:

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 2 2 2 2 2 2
 2 2]
['setosa' 'versicolor' 'virginica']
```

In this case, 0 represents *setosa*, 1 represents *versicolor*, and 2 represents *virginica*.

TIP **Note that not all sample datasets in Scikit-learn support the `feature_names` and `target_names` properties.**

Figure 5.2 summarizes what the dataset looks like.

sepal length	sepal width	petal length	petal width	target
5.1	3.5	1.4	0.2	*0*
4.9	3.0	1.4	0.2	*0*
...	...	...	...	...
5.9	3.0	5.1	1.8	*2*

0 represents setosa, 1 represents versicolor, 2 represents virginica

Figure 5.2: The fields in the Iris dataset and its target

Often, it is useful to convert the data to a Pandas dataframe, so that you can manipulate it easily:

```
import pandas as pd
df = pd.DataFrame(iris.data)    # convert features
                                # to dataframe in Pandas
print(df.head())
```

These statements print out the following:

```
     0    1    2    3
0  5.1  3.5  1.4  0.2
1  4.9  3.0  1.4  0.2
2  4.7  3.2  1.3  0.2
3  4.6  3.1  1.5  0.2
4  5.0  3.6  1.4  0.2
```

Besides the Iris dataset, you can also load some interesting datasets in Scikit-learn, such as the following:

```
# data on breast cancer
breast_cancer = datasets.load_breast_cancer()

# data on diabetes
diabetes = datasets.load_diabetes()

# dataset of 1797 8x8 images of hand-written digits
digits = datasets.load_digits()
```

For more information on the Scikit-learn dataset, check out the documentation at `http://scikit-learn.org/stable/datasets/index.html`.

Using the Kaggle Dataset

Kaggle is the world's largest community of data scientists and machine learners. What started off as a platform for offering machine learning competitions, Kaggle now also offers a public data platform, as well as a cloud-based workbench for data scientists. Google acquired Kaggle in March 2017.

For learners of machine learning, you can make use of the sample datasets provided by Kaggle at `https://www.kaggle.com/datasets/`. Some of the interesting datasets include:

- **Women's Shoe Prices:** A list of 10,000 women's shoes and the prices at which they are sold (`https://www.kaggle.com/datafiniti/womens-shoes-prices`)
- **Fall Detection Data from China:** Activity of elderly patients along with their medical information (`https://www.kaggle.com/pitasr/falldata`)
- **NYC Property Sales:** A year's worth of properties sold on the NYC real estate market (`https://www.kaggle.com/new-york-city/nyc-property-sales#nyc-rolling-sales.csv`)
- **US Flight Delay:** Flight Delays for year 2016 (`https://www.kaggle.com/niranjan0272/us-flight-delay`)

Using the UCI (University of California, Irvine) Machine Learning Repository

The UCI Machine Learning Repository (`https://archive.ics.uci.edu/ml/datasets.html`) is a collection of databases, domain theories, and data generators that are used by the machine learning community for the empirical analysis

of machine learning algorithms. Here are some interesting ones from the huge dataset it contains:

- **Auto MPG Data Set:** A collection of data about the fuel efficiency of different types of cars (`https://archive.ics.uci.edu/ml/datasets/Auto+MPG`)
- **Student Performance Data Set:** Predict student performance in secondary education (high school) (`https://archive.ics.uci.edu/ml/datasets/Student+Performance`)
- **Census Income Data Set:** Predict whether income exceeds $50K/yr. based on census data (`https://archive.ics.uci.edu/ml/datasets/census+income`)

Generating Your Own Dataset

If you cannot find a suitable dataset for experimentation, why not generate one yourself? The `sklearn.datasets.samples_generator` module from the Scikit-learn library contains a number of functions to let you generate different types of datasets for different types of problems. You can use it to generate datasets of different distributions, such as the following:

- Linearly distributed datasets
- Clustered datasets
- Clustered datasets distributed in circular fashion

Linearly Distributed Dataset

The `make_regression()` function generates data that is linearly distributed. You can specify the number of features that you want, as well as the standard deviation of the Gaussian noise applied to the output:

```
%matplotlib inline
from matplotlib import pyplot as plt
from sklearn.datasets.samples_generator import make_regression

X, y = make_regression(n_samples=100, n_features=1, noise=5.4)
plt.scatter(X,y)
```

Figure 5.3 shows the scatter plot of the dataset generated.

Clustered Dataset

The `make_blobs()` function generates *n* number of clusters of random data. This is very useful when performing clustering in unsupervised learning (Chapter 9, "Supervised Learning—Classification using K Nearest Neighbors (KNN)"):

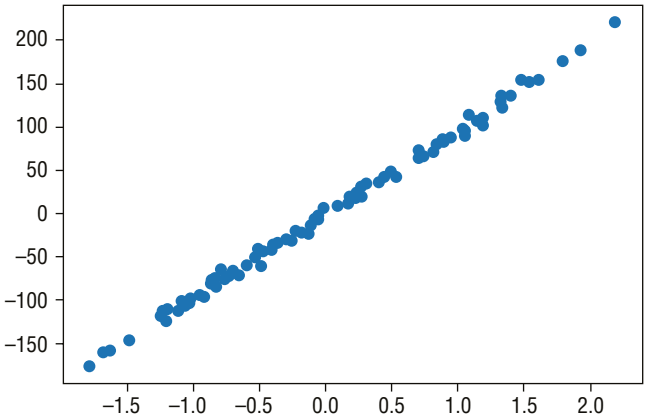

Figure 5.3: Scatter plot showing the linearly distributed data points

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import make_blobs

X, y = make_blobs(500, centers=3)  # Generate isotropic Gaussian
                                   # blobs for clustering

rgb = np.array(['r', 'g', 'b'])

# plot the blobs using a scatter plot and use color coding
plt.scatter(X[:, 0], X[:, 1], color=rgb[y])
```

Figure 5.4 shows the scatter plot of the random dataset generated.

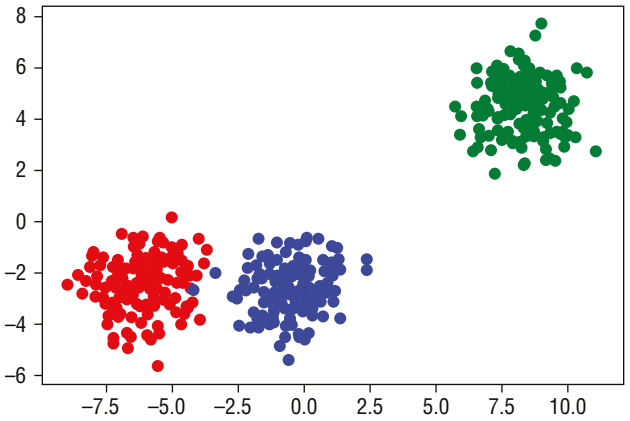

Figure 5.4: Scatter plot showing the three clusters of data points generated

Clustered Dataset Distributed in Circular Fashion

The `make_circles()` function generates a random dataset containing a large circle embedding a smaller circle in two dimensions. This is useful when performing classifications, using algorithms like SVM (Support Vector Machines). SVM will be covered in Chapter 8, "Supervised Learning—Classification using SVM."

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import make_circles

X, y = make_circles(n_samples=100, noise=0.09)

rgb = np.array(['r', 'g', 'b'])
plt.scatter(X[:, 0], X[:, 1], color=rgb[y])
```

Figure 5.5 shows the scatter plot of the random dataset generated.

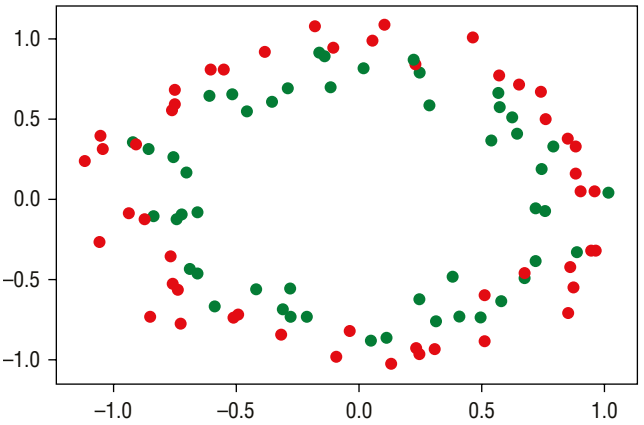

Figure 5.5: Scatter plot showing the two clusters of data points distributed in circular fashion

Getting Started with Scikit-learn

The easiest way to get started with machine learning with Scikit-learn is to start with linear regression. *Linear regression* is a linear approach for modeling the relationship between a scalar dependent variable *y* and one or more explanatory variables (or independent variables). For example, imagine that you have a set of data comprising the heights (in meters) of a group of people and their corresponding weights (in kg):

```
%matplotlib inline
import matplotlib.pyplot as plt
```

```
# represents the heights of a group of people in meters
heights = [[1.6], [1.65], [1.7], [1.73], [1.8]]

# represents the weights of a group of people in kgs
weights = [[60], [65], [72.3], [75], [80]]

plt.title('Weights plotted against heights')
plt.xlabel('Heights in meters')
plt.ylabel('Weights in kilograms')

plt.plot(heights, weights, 'k.')

# axis range for x and y
plt.axis([1.5, 1.85, 50, 90])
plt.grid(True)
```

When you plot a chart of weights against heights, you will see the chart as shown in Figure 5.6.

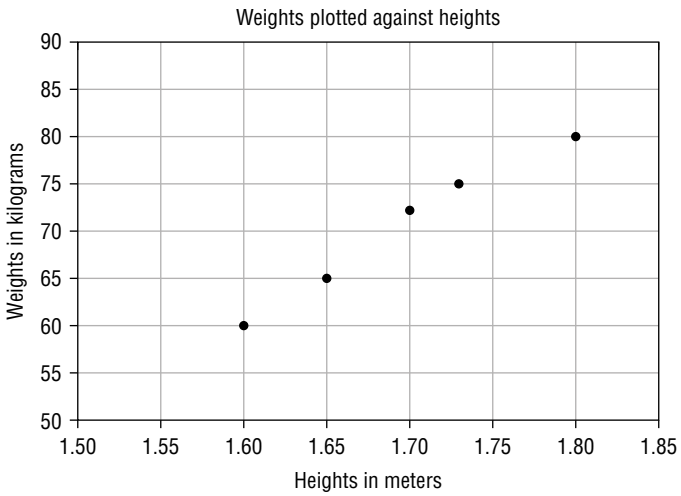

Figure 5.6: Plotting the weights against heights for a group of people

From the chart, you can see that there is a positive correlation between the weights and heights for this group of people. You could draw a straight line through the points and use that to predict the weight of another person based on their height.

Using the LinearRegression Class for Fitting the Model

So how do we draw the straight line that cuts though all of the points? It turns out that the Scikit-learn library has the `LinearRegression` class that helps you to do just that. All you need to do is to create an instance of this class and use

the *heights* and *weights* lists to create a linear regression model using the `fit()` function, like this:

```
from sklearn.linear_model import LinearRegression

# Create and fit the model
model = LinearRegression()
model.fit(X=heights, y=weights)
```

TIP **Observe that the *heights* and *weights* are both represented as two-dimensional lists. This is because the `fit()` function requires both the *X* and *y* arguments to be two-dimensional (of type `list` or `ndarray`).**

Making Predictions

Once you have fitted (trained) the model, you can start to make predictions using the `predict()` function, like this:

```
# make prediction
weight = model.predict([[1.75]])[0][0]
print(round(weight,2))          # 76.04
```

In the preceding example, you want to predict the weight for a person that is 1.75m tall. Based on the model, the weight is predicted to be 76.04kg.

TIP **In Scikit-learn, you typically use the `fit()` function to train a model. Once the model is trained, you use the `predict()` function to make a prediction.**

Plotting the Linear Regression Line

It would be useful to visualize the linear regression line that has been created by the `LinearRegression` class. Let's do this by first plotting the original data points and then sending the *heights* list to the model to predict the weights. We then plot the series of forecasted weights to obtain the line. The following code snippet shows how this is done:

```
import matplotlib.pyplot as plt

heights = [[1.6], [1.65], [1.7], [1.73], [1.8]]
weights = [[60], [65], [72.3], [75], [80]]

plt.title('Weights plotted against heights')
plt.xlabel('Heights in meters')
plt.ylabel('Weights in kilograms')
```

```
plt.plot(heights, weights, 'k.')

plt.axis([1.5, 1.85, 50, 90])
plt.grid(True)

# plot the regression line
plt.plot(heights, model.predict(heights), color='r')
```

Figure 5.7 shows the linear regression line.

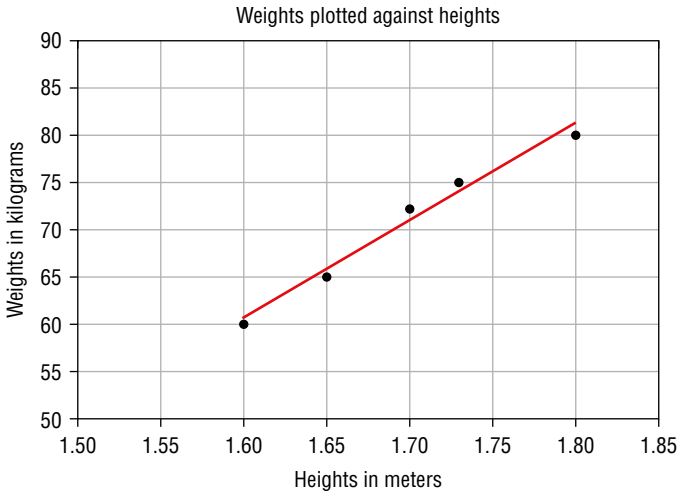

Figure 5.7: Plotting the linear regression line

Getting the Gradient and Intercept of the Linear Regression Line

From Figure 5.7, it is not clear at what value the linear regression line intercepts the y-axis. This is because we have adjusted the x-axis to start plotting at 1.5. A better way to visualize this would be to set the x-axis to start from 0 and enlarge the range of the y-axis. You then plot the line by feeding in two extreme values of the height: 0 and 1.8. The following code snippet re-plots the points and the linear regression line:

```
plt.title('Weights plotted against heights')
plt.xlabel('Heights in meters')
plt.ylabel('Weights in kilograms')

plt.plot(heights, weights, 'k.')

plt.axis([0, 1.85, -200, 200])
plt.grid(True)
```

```
# plot the regression line
extreme_heights = [[0], [1.8]]
plt.plot(extreme_heights, model.predict(extreme_heights), color='b')
```

Figure 5.8 now shows the point where the line cuts the y-axis.

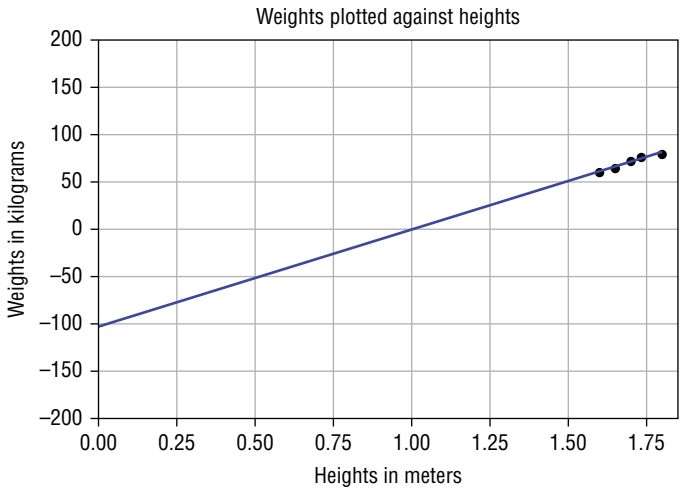

Figure 5.8: The linear regression line

While you can get the y-intercept by predicting the weight if the height is 0:

```
round(model.predict([[0]])[0][0],2)    # -104.75
```

the *model* object provides the answer directly through the `intercept_` property:

```
print(round(model.intercept_[0],2))    # -104.75
```

Using the *model* object, you can also get the gradient of the linear regression line through the `coef_` property:

```
print(round(model.coef_[0][0],2))     # 103.31
```

Examining the Performance of the Model by Calculating the Residual Sum of Squares

To know if your linear regression line is well fitted to all of the data points, we use the *Residual Sum of Squares (RSS)* method. Figure 5.9 shows how the RSS is calculated.

The following code snippet shows how the RSS is calculated in Python:

```
import numpy as np

print('Residual sum of squares: %.2f' %
      np.sum((weights - model.predict(heights)) ** 2))
# Residual sum of squares: 5.34
```

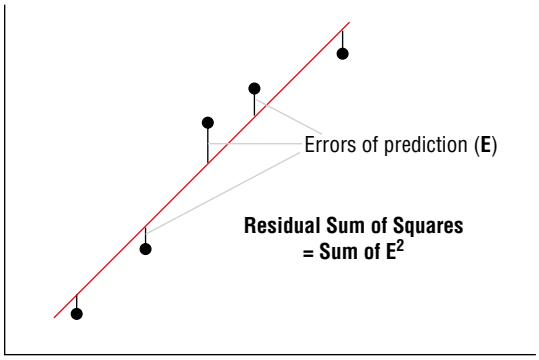

Figure 5.9: Calculating the Residual Sum of Squares for linear regression

The RSS should be as small as possible, with 0 indicating that the regression line fits the points exactly (rarely achievable in the real world).

Evaluating the Model Using a Test Dataset

Now that our model is trained with our training data, we can put it to the test. Assuming that we have the following test dataset:

```
# test data
heights_test = [[1.58], [1.62], [1.69], [1.76], [1.82]]
weights_test = [[58], [63], [72], [73], [85]]
```

we can measure how closely the test data fits the regression line using the *R-Squared method*. The R-Squared method is also known as the *coefficient of determination*, or the *coefficient of multiple determinations for multiple regressions*.

The formula for calculating R-Squared is shown in Figure 5.10.

$$R^2 = 1 - \frac{RSS}{\text{TSS}}$$

$$\text{TSS} = \sum_{i=1}^{n} (y_i - \bar{y})^2$$

$$RSS = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

Figure 5.10: The formula for calculating R-Squared

Using the formula shown for R-Squared, note the following:

- R^2 is R-squared
- TSS is Total Sum of Squares
- RSS is Residual Sum of Squares

You can now calculate it in Python using the following code snippet:

```
# Total Sum of Squares (TSS)
weights_test_mean = np.mean(np.ravel(weights_test))
TSS = np.sum((np.ravel(weights_test) -
              weights_test_mean) ** 2)
print("TSS: %.2f" % TSS)

# Residual Sum of Squares (RSS)
RSS = np.sum((np.ravel(weights_test) -
              np.ravel(model.predict(heights_test)))
                 ** 2)
print("RSS: %.2f" % RSS)

# R_squared
R_squared = 1 - (RSS / TSS)
print("R-squared: %.2f" % R_squared)
```

TIP **The `ravel()` function converts the two-dimensional list into a contiguous flattened (one-dimensional) array.**

The preceding code snippet yields the following result:

```
TSS: 430.80
RSS: 24.62
R-squared: 0.94
```

Fortunately, you don't have to calculate the R-Squared manually yourself—Scikit-learn has the `score()` function to calculate the R-Squared automatically for you:

```
# using scikit-learn to calculate r-squared
print('R-squared: %.4f' % model.score(heights_test,
                                      weights_test))

# R-squared: 0.9429
```

An R-Squared value of 0.9429 (94.29%) indicates a pretty good fit for your test data.

Persisting the Model

Once you have trained a model, it is often useful to be able to save it for later use. Rather than retraining the model every time you have new data to test, a saved model allows you to load the trained model and make predictions immediately without the need to train the model again.

There are two ways to save your trained model in Python:

- Using the standard `pickle` module in Python to serialize and deserialize objects
- Using the `joblib` module in Scikit-learn that is optimized to save and load Python objects that deal with NumPy data

The first example you will see is saving the model using the `pickle` module:

```
import pickle

# save the model to disk
filename = 'HeightsAndWeights_model.sav'
# write to the file using write and binary mode
pickle.dump(model, open(filename, 'wb'))
```

In the preceding code snippet, you first opened a file in "`wb`" mode ("`w`" for write and "`b`" for binary). You then use the `dump()` function from the `pickle` module to save the model into the file.

To load the model from file, use the `load()` function:

```
# load the model from disk
loaded_model = pickle.load(open(filename, 'rb'))
```

You can now use the model as usual:

```
result = loaded_model.score(heights_test,
                            weights_test)
```

Using the `joblib` module is very similar to using the `pickle` module:

```
from sklearn.externals import joblib

# save the model to disk
filename = 'HeightsAndWeights_model2.sav'
joblib.dump(model, filename)

# load the model from disk
loaded_model = joblib.load(filename)
result = loaded_model.score(heights_test,
                            weights_test)
print(result)
```

Data Cleansing

In machine learning, one of the first tasks that you need to perform is *data cleansing*. Very seldom would you have a dataset that you can use straightaway to train your model. Instead, you have to examine the data carefully for any

missing values and either remove them or replace them with some valid values, or you have to normalize them if there are columns with wildly different values. The following sections show some of the common tasks you need to perform when cleaning your data.

Cleaning Rows with NaNs

Consider a CSV file named `NaNDataset.csv` with the following content:

```
A,B,C
1,2,3
4,,6
7,,9
10,11,12
13,14,15
16,17,18
```

Visually, you can spot that there are a few rows with empty fields. Specifically, the second and third rows have missing values for the second columns. For small sets of data, this is easy to spot. But if you have a large dataset, it becomes almost impossible to detect. An effective way to detect for empty rows is to load the dataset into a Pandas dataframe and then use the `isnull()` function to check for null values in the dataframe:

```
import pandas as pd
df = pd.read_csv('NaNDataset.csv')
df.isnull().sum()
```

This code snippet will produce the following output:

```
A    0
B    2
C    0
dtype: int64
```

You can see that column B has two null values. When Pandas loads a dataset containing empty values, it will use `NaN` to represent those empty fields. The following is the output of the dataframe when you print it out:

```
    A     B   C
0   1   2.0   3
1   4   NaN   6
2   7   NaN   9
3  10  11.0  12
4  13  14.0  15
5  16  17.0  18
```

Replacing NaN with the Mean of the Column

One of the ways to deal with `NaN`s in your dataset is to replace them with the mean of the columns in which they are located. The following code snippet replaces all of the `NaN`s in column B with the average value of column B:

```
# replace all the NaNs in column B with the average of column B
df.B = df.B.fillna(df.B.mean())
print(df)
```

The dataframe now looks like this:

```
    A     B   C
0   1   2.0   3
1   4  11.0   6
2   7  11.0   9
3  10  11.0  12
4  13  14.0  15
5  16  17.0  18
```

Removing Rows

Another way to deal with `NaN`s in your dataset is simply to remove the rows containing them. You can do so using the `dropna()` function, like this:

```
df = pd.read_csv('NaNDataset.csv')
df = df.dropna()                              # drop all rows with NaN
print(df)
```

This code snippet will produce the following output:

```
    A     B   C
0   1   2.0   3
3  10  11.0  12
4  13  14.0  15
5  16  17.0  18
```

Observe that after removing the rows containing `NaN`, the index is no longer in sequential order. If you need to reset the index, use the `reset_index()` function:

```
df = df.reset_index(drop=True)                # reset the index
print(df)
```

The dataframe with the reset index will now look like this:

```
    A     B   C
0   1   2.0   3
1  10  11.0  12
2  13  14.0  15
3  16  17.0  18
```

Removing Duplicate Rows

Consider a CSV file named `DuplicateRows.csv` with the following content:

```
A,B,C
1,2,3
4,5,6
4,5,6
7,8,9
7,18,9
10,11,12
10,11,12
13,14,15
16,17,18
```

To find all of the duplicated rows, first load the dataset into a dataframe and then use the `duplicated()` function, like this:

```
import pandas as pd
df = pd.read_csv('DuplicateRows.csv')
print(df.duplicated(keep=False))
```

This will produce the following output:

```
0    False
1     True
2     True
3    False
4    False
5     True
6     True
7    False
8    False
dtype: bool
```

It shows which rows are duplicated. In this example, rows with index 1, 2, 5, and 6 are duplicates. The `keep` argument allows you to specify how to indicate duplicates:

- The default is `'first'`: All duplicates are marked as `True` except for the first occurrence
- `'last'`: All duplicates are marked as `True` except for the last occurrence
- `False`: All duplicates are marked as `True`

So, if you set `keep` to `'first'`, you will see the following output:

```
0    False
1    False
2     True
```

```
3    False
4    False
5    False
6     True
7    False
8    False
dtype: bool
```

Hence, if you want to see all duplicate rows, you can set `keep` to `False` and use the result of the `duplicated()` function as the index into the dataframe:

```
print(df[df.duplicated(keep=False)])
```

The preceding statement will print all of the duplicate rows:

```
    A   B   C
1   4   5   6
2   4   5   6
5  10  11  12
6  10  11  12
```

To drop duplicate rows, you can use the `drop_duplicates()` function, like this:

```
df.drop_duplicates(keep='first', inplace=True)  # remove
duplicates and keep the first
print(df)
```

TIP **By default, the `drop_duplicates()` function will not modify the original dataframe and will return the dataframe containing the dropped rows. If you want to modify the original dataframe, set the `inplace` parameter to `True`, as shown in the preceding code snippet.**

The preceding statements will print the following:

```
    A   B   C
0   1   2   3
1   4   5   6
3   7   8   9
4   7  18   9
5  10  11  12
7  13  14  15
8  16  17  18
```

TIP **To remove all duplicates, set the `keep` parameter to `False`. To keep the last occurrence of duplicate rows, set the `keep` parameter to `'last'`.**

Sometimes, you only want to remove duplicates that are found in certain columns in the dataset. For example, if you look at the dataset that we have

been using, observe that for row 3 and row 4, the values of column A and C are identical:

```
    A   B   C
3   7   8   9
4   7  18   9
```

You can remove duplicates in certain columns by specifying the `subset` parameter:

```
df.drop_duplicates(subset=['A', 'C'], keep='last',
                           inplace=True)     # remove all duplicates in
                                             # columns A and C and keep
                                             # the last
print(df)
```

This statement will yield the following:

```
    A   B   C
0   1   2   3
1   4   5   6
4   7  18   9
5  10  11  12
7  13  14  15
8  16  17  18
```

Normalizing Columns

Normalization is a technique often applied during the data cleansing process. The aim of *normalization* is to change the values of numeric columns in the dataset to use a common scale, without modifying the differences in the ranges of values.

Normalization is crucial for some algorithms to model the data correctly. For example, one of the columns in your dataset may contain values from 0 to 1, while another column has values ranging from 400,000 to 500,000. The huge disparity in the scale of the numbers could introduce problems when you use the two columns to train your model. Using normalization, you could maintain the ratio of the values in the two columns while keeping them to a limited range. In Pandas, you can use the `MinMaxScaler` class to scale each column to a particular range of values.

Consider a CSV file named `NormalizeColumns.csv` with the following content:

```
A,B,C
1000,2,3
400,5,6
700,6,9
100,11,12
1300,14,15
1600,17,18
```

The following code snippet will scale all the columns' values to the (0,1) range:

```
import pandas as pd
from sklearn import preprocessing

df = pd.read_csv('NormalizeColumns.csv')
x = df.values.astype(float)

min_max_scaler = preprocessing.MinMaxScaler()
x_scaled = min_max_scaler.fit_transform(x)
df = pd.DataFrame(x_scaled, columns=df.columns)
print(df)
```

You should see the following output:

```
     A         B    C
0  0.6  0.000000  0.0
1  0.2  0.200000  0.2
2  0.4  0.266667  0.4
3  0.0  0.600000  0.6
4  0.8  0.800000  0.8
5  1.0  1.000000  1.0
```

Removing Outliers

In statistics, an *outlier* is a point that is distant from other observed points. For example, given a set of values—234, 267, 1, 200, 245, 300, 199, 250, 8999, and 245—it is quite obvious that 1 and 8999 are outliers. They distinctly stand out from the rest of the values, and they "lie outside" most of the other values in the dataset; hence the word *outlier*. Outliers occur mainly due to errors in recording or experimental error, and in machine learning it is important to remove them prior to training your model as it may potentially distort your model if you don't.

There are a number of techniques to remove outliers, and in this chapter we discuss two of them:

- Tukey Fences
- Z-Score

Tukey Fences

Tukey Fences is based on *Interquartile Range (IQR)*. IQR is the difference between the first and third quartiles of a set of values. The first quartile, denoted Q1, is the value in the dataset that holds 25% of the values below it. The third quartile,

denoted Q3, is the value in the dataset that holds 25% of the values above it. Hence, by definition, IQR = Q3 – Q1.

Figure 5.11 shows an example of how IQR is obtained for datasets with even and odd numbers of values.

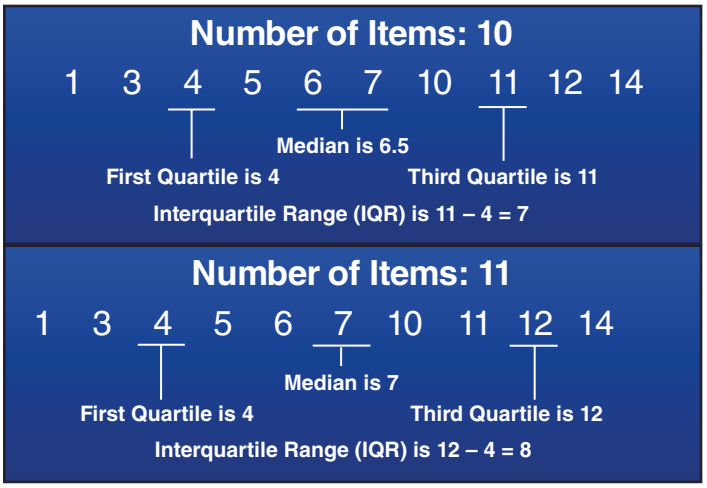

Figure 5.11: Examples of finding the Interquartile Range (IQR)

In Tukey Fences, outliers are values that are as follows:

- Less than Q1 – (1.5 × IQR), or
- More than Q3 + (1.5 × IQR)

The following code snippet shows the implementation of Tukey Fences using Python:

```
import numpy as np

def outliers_iqr(data):
    q1, q3 = np.percentile(data, [25, 75])
    iqr = q3 - q1
    lower_bound = q1 - (iqr * 1.5)
    upper_bound = q3 + (iqr * 1.5)
    return np.where((data > upper_bound) | (data < lower_bound))
```

TIP **The `np.where()` function returns the location of items satisfying the conditions.**

The `outliers_iqr()` function returns a tuple of which the first element is an array of indices of those rows that have outlier values.

To test the Tukey Fences, let's use the famous Galton dataset on the heights of parents and their children. The dataset contains data based on the famous 1885 study of Francis Galton exploring the relationship between the heights of adult children and the heights of their parents. Each case is an adult child, and the variables are as follows:

Family: The family that the child belongs to, labeled by the numbers from 1 to 204 and 136A
Father: The father's height, in inches
Mother: The mother's height, in inches
Gender: The gender of the child, male (M) or female (F)
Height: The height of the child, in inches
Kids: The number of kids in the family of the child

The dataset has 898 cases.
First, import the data:

```
import pandas as pd
df = pd.read_csv("http://www.mosaic-web.org/go/datasets/galton.csv")
print(df.head())
```

You should see the following:

```
  family  father  mother sex  height  nkids
0      1    78.5    67.0   M    73.2      4
1      1    78.5    67.0   F    69.2      4
2      1    78.5    67.0   F    69.0      4
3      1    78.5    67.0   F    69.0      4
4      2    75.5    66.5   M    73.5      4
```

If you want to find the outliers in the `height` column, you can call the `outliers_iqr()` function as follows:

```
print("Outliers using outliers_iqr()")
print("=============================")
for i in outliers_iqr(df.height)[0]:
    print(df[i:i+1])
```

You should see the following output:

```
Outliers using outliers_iqr()
=============================
    family  father  mother sex  height  nkids
288     72    70.0    65.0   M    79.0      7
```

Using the Tukey Fences method, you can see that the `height` column has a single outlier.

Z-Score

The second method for determining outliers is to use the *Z-score* method. A Z-score indicates how many standard deviations a data point is from the mean. The Z-score has the following formula:

$$\mathbf{Z} = \left(\mathbf{x}_i - \mu\right) / \sigma$$

where x_i is the data point, μ is the mean of the dataset, and σ is the standard deviation.

This is how you interpret the Z-score:

- A negative Z-score indicates that the data point is less than the mean, and a positive Z-score indicates the data point in question is larger than the mean
- A Z-score of 0 tells you that the data point is right in the middle (mean), and a Z-score of 1 tells you that your data point is 1 standard deviation above the mean, and so on
- Any Z-score greater than 3 or less than –3 is considered to be an outlier

The following code snippet shows the implementation of the Z-score using Python:

```
def outliers_z_score(data):
    threshold = 3
    mean = np.mean(data)
    std = np.std(data)
    z_scores = [(y - mean) / std for y in data]
    return np.where(np.abs(z_scores) > threshold)
```

Using the same Galton dataset that you used earlier, you can now find the outliers for the `height` column using the `outliers_z_score()` function:

```
print("Outliers using outliers_z_score()")
print("================================")
for i in outliers_z_score(df.height)[0]:
    print(df[i:i+1])
print()
```

You should see the following output:

```
Outliers using outliers_z_score()
================================
    family  father  mother sex  height  nkids
125     35    71.0    69.0   M    78.0      5
    family  father  mother sex  height  nkids
288     72    70.0    65.0   M    79.0      7
    family  father  mother sex  height  nkids
672    155    68.0    60.0   F    56.0      7
```

Using the Z-score method, you can see that the `height` column has three outliers.

Summary

In this chapter, you have seen how to get started with the Scikit-learn library to solve a linear regression problem. In addition, you have also learned how to get sample datasets, generate your own, perform data cleansing, as well as the two techniques that you can use to remove outliers from your datasets.

In subsequent chapters, you will learn more about the various machine learning algorithms and how to use them to solve real-life problems.