CHAPTER

6

Supervised Learning—Linear Regression

Types of Linear Regression

In the previous chapter, you learned how to get started with machine learning using simple linear regression, first using Python, and then followed by using the Scikit-learn library. In this chapter, we will look into linear regression in more detail and discuss another variant of linear regression known as *polynomial regression*.

To recap, Figure 6.1 shows the Iris dataset used in Chapter 5, "Getting Started with Scikit-learn for Machine Learning." The first four columns are known as the *features*, or also commonly referred to as the *independent variables*. The last column is known as the *label*, or commonly called the *dependent variable* (or *dependent variables* if there is more than one label).

	Features (Independent variables)			Label (Dependent variable)
sepal length	sepal width	petal length	petal width	target

Figure 6.1: Some terminologies for features and label

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

TIP **Features are also sometimes called *explanatory variables*, while labels are also sometimes called *targets*.**

In simple linear regression, we talked about the linear relationship between one independent variable and one dependent variable. In this chapter, besides simple linear regression, we will also discuss the following:

Multiple Regression Linear relationships between two or more independent variables and one dependent variable.

Polynomial Regression Modeling the relationship between one independent variable and one dependent variable using an n^{th} degree polynomial function.

Polynomial Multiple Regression Modeling the relationship between two or more independent variables and one dependent variable using an n^{th} degree polynomial function.

TIP **There is another form of linear regression, called *multivariate linear regression*, where there is more than one correlated dependent variable in the relationship. Multivariate linear regression is beyond the scope of this book.**

Linear Regression

In machine learning, *linear regression* is one of the simplest algorithms that you can apply to a dataset to model the relationships between features and labels. In Chapter 5, we started by exploring simple linear regression, where we could explain the relationship between a feature and a label by using a straight line. In the following section, you will learn about a variant of simple linear regression, called *multiple linear regression*, by predicting house prices based on multiple features.

Using the Boston Dataset

For this example, we will use the Boston dataset, which contains data about the housing and price data in the Boston area. This dataset was taken from the StatLib library, which is maintained at Carnegie Mellon University. It is commonly used in machine learning, and it is a good candidate to learn about regression problems. The Boston dataset is available from a number of sources, but it is now available directly from the `sklearn.datasets` package. This means you can load it directly in Scikit-learn without needing explicitly to download it.

First, let's import the necessary libraries and then load the dataset using the `load_boston()` function:

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

from sklearn.datasets import load_boston
dataset = load_boston()
```

It is always good to examine the data before you work with it. The `data` property contains the data for the various columns of the dataset:

```
print(dataset.data)
```

You should see the following:

```
[[  6.32000000e-03   1.80000000e+01   2.31000000e+00 ...,   1.53000000e+01
    3.96900000e+02   4.98000000e+00]
 [  2.73100000e-02   0.00000000e+00   7.07000000e+00 ...,   1.78000000e+01
    3.96900000e+02   9.14000000e+00]
 [  2.72900000e-02   0.00000000e+00   7.07000000e+00 ...,   1.78000000e+01
    3.92830000e+02   4.03000000e+00]
 ...,
 [  6.07600000e-02   0.00000000e+00   1.19300000e+01 ...,   2.10000000e+01
    3.96900000e+02   5.64000000e+00]
 [  1.09590000e-01   0.00000000e+00   1.19300000e+01 ...,   2.10000000e+01
    3.93450000e+02   6.48000000e+00]
 [  4.74100000e-02   0.00000000e+00   1.19300000e+01 ...,   2.10000000e+01
    3.96900000e+02   7.88000000e+00]]
```

The data is a two-dimensional array. To know the name of each column (feature), use the `feature_names` property:

```
print(dataset.feature_names)
```

You should see the following:

```
['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'
 'B' 'LSTAT']
```

For the description of each feature, you can use the `DESCR` property:

```
print(dataset.DESCR)
```

The preceding statement will print out the following:

```
Boston House Prices dataset
===========================

Notes
------
```

```
Data Set Characteristics:

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive

    :Median Value (attribute 14) is usually the target

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over
25,000 sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds
river; 0 otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of
blacks by town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset: http://archive.ics.uci.edu/
ml/datasets/Housing


This dataset was taken from the StatLib library which is maintained at
Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
prices and the demand for clean air', J. Environ. Economics & Management,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression
diagnostics
...', Wiley, 1980.   N.B. Various transformations are used in the table on
pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning
papers that address regression
problems.

**References**
```

```
   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying
Influential Data and Sources of Collinearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based
Learning. In Proceedings on the Tenth International Conference of
Machine Learning, 236-243, University of Massachusetts, Amherst. Morgan
Kaufmann.
   - many more! (see http://archive.ics.uci.edu/ml/datasets/
Housing)
```

The prices of houses is the information we are seeking, and it can be accessed via the `target` property:

```
print(dataset.target)
```

You will see the following:

```
[ 24.   21.6  34.7  33.4  36.2  28.7  22.9  27.1  16.5  18.9  15.   18.9
  21.7  20.4  18.2  19.9  23.1  17.5  20.2  18.2  13.6  19.6  15.2  14.5
  15.6  13.9  16.6  14.8  18.4  21.   12.7  14.5  13.2  13.1  13.5  18.9
  20.   21.   24.7  30.8  34.9  26.6  25.3  24.7  21.2  19.3  20.   16.6
  14.4  19.4  19.7  20.5  25.   23.4  18.9  35.4  24.7  31.6  23.3  19.6
  18.7  16.   22.2  25.   33.   23.5  19.4  22.   17.4  20.9  24.2  21.7
  22.8  23.4  24.1  21.4  20.   20.8  21.2  20.3  28.   23.9  24.8  22.9
  23.9  26.6  22.5  22.2  23.6  28.7  22.6  22.   22.9  25.   20.6  28.4
  21.4  38.7  43.8  33.2  27.5  26.5  18.6  19.3  20.1  19.5  19.5  20.4
  19.8  19.4  21.7  22.8  18.8  18.7  18.5  18.3  21.2  19.2  20.4  19.3
  22.   20.3  20.5  17.3  18.8  21.4  15.7  16.2  18.   14.3  19.2  19.6
  23.   18.4  15.6  18.1  17.4  17.1  13.3  17.8  14.   14.4  13.4  15.6
  11.8  13.8  15.6  14.6  17.8  15.4  21.5  19.6  15.3  19.4  17.   15.6
  13.1  41.3  24.3  23.3  27.   50.   50.   50.   22.7  25.   50.   23.8
  23.8  22.3  17.4  19.1  23.1  23.6  22.6  29.4  23.2  24.6  29.9  37.2
  39.8  36.2  37.9  32.5  26.4  29.6  50.   32.   29.8  34.9  37.   30.5
  36.4  31.1  29.1  50.   33.3  30.3  34.6  34.9  32.9  24.1  42.3  48.5
  50.   22.6  24.4  22.5  24.4  20.   21.7  19.3  22.4  28.1  23.7  25.
  23.3  28.7  21.5  23.   26.7  21.7  27.5  30.1  44.8  50.   37.6  31.6
  46.7  31.5  24.3  31.7  41.7  48.3  29.   24.   25.1  31.5  23.7  23.3
  22.   20.1  22.2  23.7  17.6  18.5  24.3  20.5  24.5  26.2  24.4  24.8
  29.6  42.8  21.9  20.9  44.   50.   36.   30.1  33.8  43.1  48.8  31.
  36.5  22.8  30.7  50.   43.5  20.7  21.1  25.2  24.4  35.2  32.4  32.
  33.2  33.1  29.1  35.1  45.4  35.4  46.   50.   32.2  22.   20.1  23.2
  22.3  24.8  28.5  37.3  27.9  23.9  21.7  28.6  27.1  20.3  22.5  29.
  24.8  22.   26.4  33.1  36.1  28.4  33.4  28.2  22.8  20.3  16.1  22.1
  19.4  21.6  23.8  16.2  17.8  19.8  23.1  21.   23.8  23.1  20.4  18.5
  25.   24.6  23.   22.2  19.3  22.6  19.8  17.1  19.4  22.2  20.7  21.1
  19.5  18.5  20.6  19.   18.7  32.7  16.5  23.9  31.2  17.5  17.2  23.1
  24.5  26.6  22.9  24.1  18.6  30.1  18.2  20.6  17.8  21.7  22.7  22.6
  25.   19.9  20.8  16.8  21.9  27.5  21.9  23.1  50.   50.   50.   50.
  50.   13.8  13.8  15.   13.9  13.3  13.1  10.2  10.4  10.9  11.3  12.3
   8.8   7.2  10.5   7.4  10.2  11.5  15.1  23.2   9.7  13.8  12.7  13.1
  12.5   8.5   5.    6.3   5.6   7.2  12.1   8.3   8.5   5.   11.9  27.9
  17.2  27.5  15.   17.2  17.9  16.3   7.    7.2   7.5  10.4   8.8   8.4
```

```
 16.7  14.2  20.8  13.4  11.7   8.3  10.2  10.9  11.    9.5  14.5  14.1
 16.1  14.3  11.7  13.4   9.6   8.7   8.4  12.8  10.5  17.1  18.4  15.4
 10.8  11.8  14.9  12.6  14.1  13.   13.4  15.2  16.1  17.8  14.9  14.1
 12.7  13.5  14.9  20.   16.4  17.7  19.5  20.2  21.4  19.9  19.   19.1
 19.1  20.1  19.9  19.6  23.2  29.8  13.8  13.3  16.7  12.   14.6  21.4
 23.   23.7  25.   21.8  20.6  21.2  19.1  20.6  15.2   7.    8.1  13.6
 20.1  21.8  24.5  23.1  19.7  18.3  21.2  17.5  16.8  22.4  20.6  23.9
 22.   11.9]
```

Now let's load the data into a Pandas DataFrame:

```
df = pd.DataFrame(dataset.data, columns=dataset.feature_names)
df.head()
```

The DataFrame would look like the one shown in Figure 6.2.

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33

Figure 6.2: The DataFrame containing all of the features

You would also want to add the prices of the houses to the DataFrame, so let's add a new column to the DataFrame and call it `MEDV`:

```
df['MEDV'] = dataset.target
df.head()
```

Figure 6.3 shows the complete DataFrame with the features and label.

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

Figure 6.3: The DataFrame containing all of the features and the label

Data Cleansing

The next step would be to clean the data and perform any conversion if necessary. First, use the `info()` function to check the data type of each field:

```
df.info()
```

You should see the following:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
CRIM       506 non-null float64
ZN         506 non-null float64
INDUS      506 non-null float64
CHAS       506 non-null float64
NOX        506 non-null float64
RM         506 non-null float64
AGE        506 non-null float64
DIS        506 non-null float64
RAD        506 non-null float64
TAX        506 non-null float64
PTRATIO    506 non-null float64
B          506 non-null float64
LSTAT      506 non-null float64
MEDV       506 non-null float64
dtypes: float64(14)
memory usage: 55.4 KB
```

As Scikit-learn only works with fields that are numeric, you need to encode string values into numeric values. Fortunately, the dataset contains all numerical values, and so no encoding is necessary.

Next, we need to check to see if there are any missing values. To do so, use the `isnull()` function:

```
print(df.isnull().sum())
```

Again, the dataset is good, as it does not have any missing values:

```
CRIM       0
ZN         0
INDUS      0
CHAS       0
NOX        0
RM         0
AGE        0
DIS        0
RAD        0
TAX        0
PTRATIO    0
B          0
```

```
LSTAT      0
MEDV       0
dtype: int64
```

Feature Selection

Now that the data is good to go, we are ready to move on to the next step of the process. As there are 13 features in the dataset, we do not want to use all of these features for training our model, because not all of them are relevant. Instead, we want to choose those features that directly influence the result (that is, prices of houses) to train the model. For this, we can use the `corr()` function. The `corr()` function computes the pairwise correlation of columns:

```
corr = df.corr()
print(corr)
```

You will see the following:

```
CRIM         ZN     INDUS      CHAS       NOX        RM       AGE  \
CRIM     1.000000 -0.199458  0.404471 -0.055295  0.417521 -0.219940
0.350784
ZN      -0.199458  1.000000 -0.533828 -0.042697 -0.516604  0.311991
-0.569537
INDUS    0.404471 -0.533828  1.000000  0.062938  0.763651 -0.391676
0.644779
CHAS    -0.055295 -0.042697  0.062938  1.000000  0.091203  0.091251
0.086518
NOX      0.417521 -0.516604  0.763651  0.091203  1.000000 -0.302188
0.731470
RM      -0.219940  0.311991 -0.391676  0.091251 -0.302188  1.000000
-0.240265
AGE      0.350784 -0.569537  0.644779  0.086518  0.731470 -0.240265
1.000000
DIS     -0.377904  0.664408 -0.708027 -0.099176 -0.769230  0.205246
-0.747881
RAD      0.622029 -0.311948  0.595129 -0.007368  0.611441 -0.209847
0.456022
TAX      0.579564 -0.314563  0.720760 -0.035587  0.668023 -0.292048
0.506456
PTRATIO  0.288250 -0.391679  0.383248 -0.121515  0.188933 -0.355501
0.261515
B       -0.377365  0.175520 -0.356977  0.048788 -0.380051  0.128069
-0.273534
LSTAT    0.452220 -0.412995  0.603800 -0.053929  0.590879 -0.613808
0.602339
MEDV    -0.385832  0.360445 -0.483725  0.175260 -0.427321  0.695360
-0.376955
```

```
              DIS       RAD       TAX   PTRATIO         B     LSTAT
MEDV
CRIM    -0.377904  0.622029  0.579564  0.288250 -0.377365  0.452220
-0.385832
ZN       0.664408 -0.311948 -0.314563 -0.391679  0.175520 -0.412995
0.360445
INDUS   -0.708027  0.595129  0.720760  0.383248 -0.356977  0.603800
-0.483725
CHAS    -0.099176 -0.007368 -0.035587 -0.121515  0.048788 -0.053929
0.175260
NOX     -0.769230  0.611441  0.668023  0.188933 -0.380051  0.590879
-0.427321
RM       0.205246 -0.209847 -0.292048 -0.355501  0.128069 -0.613808
0.695360
AGE     -0.747881  0.456022  0.506456  0.261515 -0.273534  0.602339
-0.376955
DIS      1.000000 -0.494588 -0.534432 -0.232471  0.291512 -0.496996
0.249929
RAD     -0.494588  1.000000  0.910228  0.464741 -0.444413  0.488676
-0.381626
TAX     -0.534432  0.910228  1.000000  0.460853 -0.441808  0.543993
-0.468536
PTRATIO -0.232471  0.464741  0.460853  1.000000 -0.177383  0.374044
-0.507787
B        0.291512 -0.444413 -0.441808 -0.177383  1.000000 -0.366087
0.333461
LSTAT   -0.496996  0.488676  0.543993  0.374044 -0.366087  1.000000
-0.737663
MEDV     0.249929 -0.381626 -0.468536 -0.507787  0.333461 -0.737663
1.000000
```

A *positive correlation* is a relationship between two variables in which both variables move in tandem. A positive correlation exists when one variable decreases as the other variable decreases, or one variable increases while the other variable increases. Similarly, a *negative correlation* is a relationship between two variables in which one variable increases as the other decreases. A perfect negative correlation is represented by the value –1.00: a 0.00 indicates no correlation and a +1.00 indicates a perfect positive correlation.

From the `MEDV` column in the output, you can see that the `RM` and `LSTAT` features have high correlation factors (positive and negative correlations) with the `MEDV`:

```
MEDV
CRIM    -0.385832
ZN       0.360445
INDUS   -0.483725
CHAS     0.175260
NOX     -0.427321
RM       0.695360
AGE     -0.376955
```

```
DIS      0.249929
RAD     -0.381626
TAX     -0.468536
PTRATIO -0.507787
B        0.333461
LSTAT   -0.737663
MEDV     1.000000
```

This means that as LSTAT ("% of lower status of the population") increases, the prices of houses go down. When LSTAT decreases, the prices go up. Similarly, as RM ("average number of rooms per dwelling") increases, so will the price. And when RM goes down, the prices go down as well.

Instead of visually finding the top two features with the highest correlation factors, we can do it programmatically as follows:

```
#---get the top 3 features that has the highest correlation---
print(df.corr().abs().nlargest(3, 'MEDV').index)

#---print the top 3 correlation values---
print(df.corr().abs().nlargest(3, 'MEDV').values[:,13])
```

The result confirms our findings:

```
Index(['MEDV', 'LSTAT', 'RM'], dtype='object')
[ 1.          0.73766273  0.69535995]
```

TIP **We will ignore the first result, as MEDV definitely has a perfect correlation with itself!**

Since RM and LSTAT have high correlation values, we will use these two features to train our model.

Multiple Regression

In the previous chapter, you saw how to perform a simple linear regression using a single feature and a label. Often, you might want to train your model using more than one independent variable and a label. This is known as *multiple regression*. In multiple regression, two or more independent variables are used to predict the value of a dependent variable (label).

Now let's plot a scatter plot showing the relationship between the LSTAT feature and the MEDV label:

```
plt.scatter(df['LSTAT'], df['MEDV'], marker='o')
plt.xlabel('LSTAT')
plt.ylabel('MEDV')
```

Figure 6.4 shows the scatter plot. It appears that there is a linear correlation between the two.

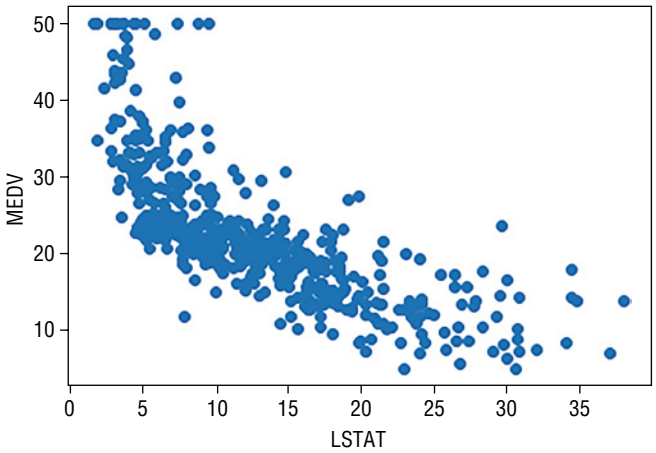

Figure 6.4: Scatter plot showing the relationship between LSTAT and MEDV

Let's also plot a scatter plot showing the relationship between the `RM` feature and the `MEDV` label:

```
plt.scatter(df['RM'], df['MEDV'], marker='o')
plt.xlabel('RM')
plt.ylabel('MEDV')
```

Figure 6.5 shows the scatter plot. Again, it appears that there is a linear correlation between the two, albeit with some outliers.

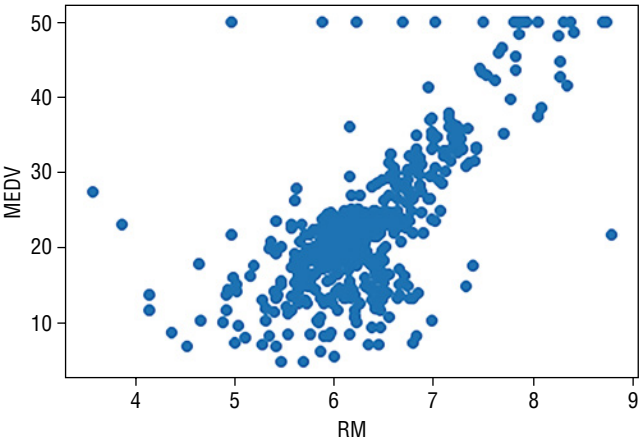

Figure 6.5: Scatter plot showing the relationship between RM and MEDV

Better still, let's plot the two features and the label on a 3D chart:

```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(18,15))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(df['LSTAT'],
           df['RM'],
           df['MEDV'],
           c='b')

ax.set_xlabel("LSTAT")
ax.set_ylabel("RM")
ax.set_zlabel("MEDV")
plt.show()
```

Figure 6.6 shows the 3D chart of `LSTAT` and `RM` plotted against `MEDV`.

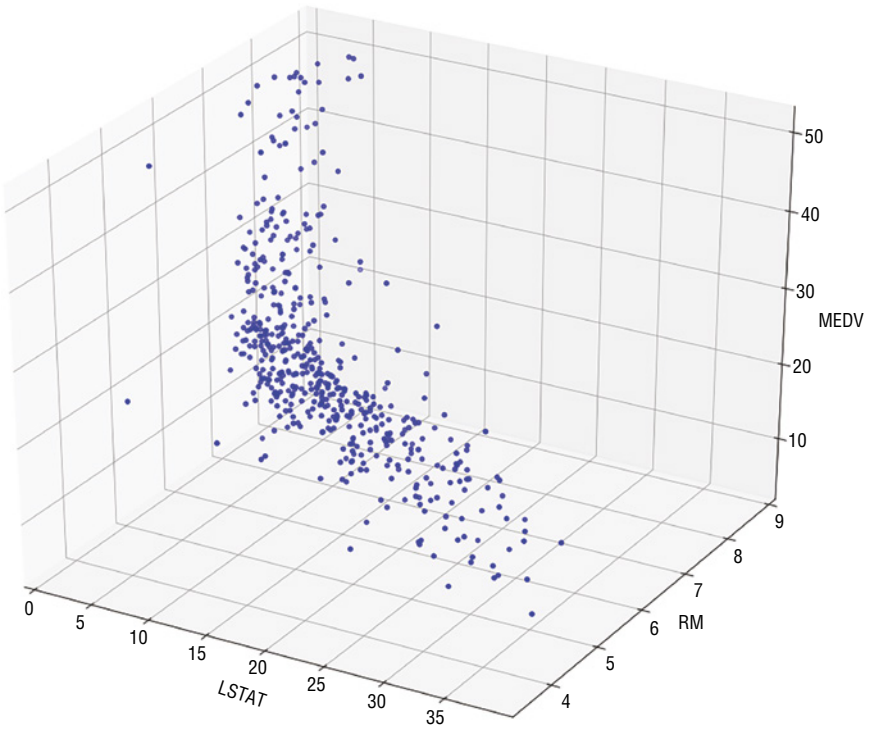

Figure 6.6: The 3D scatter plot showing the relationship between LSTAT, RM, and MEDV

Training the Model

We can now train the model. First, create two DataFrames: `x` and `Y`. The `x` DataFrame will contain the combination of the `LSTAT` and `RM` features, while the `Y` DataFrame will contain the `MEDV` label:

```
x = pd.DataFrame(np.c_[df['LSTAT'], df['RM']], columns = ['LSTAT','RM'])
Y = df['MEDV']
```

We will split the dataset into 70 percent for training and 30 percent for testing:

```
from sklearn.model_selection import train_test_split
x_train, x_test, Y_train, Y_test = train_test_split(x, Y, test_size = 0.3,
                                                    random_state=5)
```

TIP **Chapter 7, "Supervised Learning—Classification Using Logistic Regression," will discuss more about the `train_test_split()` function.**

After the split, let's print out the shape of the training sets:

```
print(x_train.shape)
print(Y_train.shape)
```

You will see the following:

```
(354, 2)
(354,)
```

This means that the `x` training set now has 354 rows and 2 columns, while the `Y` training set (which contains the label) has 354 rows and 1 column.

Let's also print out the testing set:

```
print(x_test.shape)
print(Y_test.shape)
```

This time, the testing set has 152 rows:

```
(152, 2)
(152,)
```

We are now ready to begin the training. As you learned from the previous chapter, you can use the `LinearRegression` class to perform linear regression. In this case, we will use it to train our model:

```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(x_train, Y_train)
```

Once the model is trained, we will use the testing set to perform some predictions:

```
price_pred = model.predict(x_test)
```

To learn how well our model performed, we use the R-Squared method that you learned in the previous chapter. The R-Squared method lets you know how close the test data fits the regression line. A value of 1.0 means a perfect fit. So, you aim for a value of R-Squared that is close to 1:

```
print('R-Squared: %.4f' % model.score(x_test,
                                      Y_test))
```

For our model, it returns an R-Squared value as follows:

```
R-Squared: 0.6162
```

We will also plot a scatter plot showing the actual price vs. the predicted price:

```
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(Y_test, price_pred)
print(mse)

plt.scatter(Y_test, price_pred)
plt.xlabel("Actual Prices")
plt.ylabel("Predicted prices")
plt.title("Actual prices vs Predicted prices")
```

Figure 6.7 shows the plot. Ideally, it should be a straight line, but for now it is good enough.

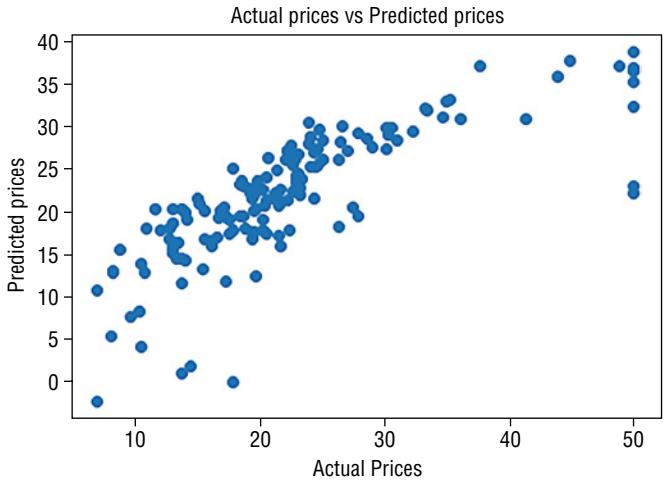

Figure 6.7: A scatter plot showing the predicted prices vs. the actual prices

Getting the Intercept and Coefficients

The formula for multiple regression is as follows:

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

where Y is the dependent variable, β_0 is the intercept, and β_1 and β_2 are the coefficient of the two features x_1 and x_2, respectively.

With the model trained, we can obtain the intercept as well as the coefficients of the features:

```
print(model.intercept_)
print(model.coef_)
```

You should see the following:

```
0.3843793678034899
[-0.65957972  4.83197581]
```

We can use the model to make a prediction for the house price when `LSTAT` is 30 and `RM` is 5:

```
print(model.predict([[30,5]]))
```

You should see the following:

```
[4.75686695]
```

You can verify the predicted value by using the formula that was given earlier:

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

$$Y = 0.3843793678034899 + 30\left(-0.65957972\right) + 5\left(4.83197581\right)$$

$$Y = 4.7568$$

Plotting the 3D Hyperplane

Let's plot a 3D regression hyperplane showing the predictions:

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

from sklearn.datasets import load_boston
dataset = load_boston()

df = pd.DataFrame(dataset.data, columns=dataset.feature_names)
df['MEDV'] = dataset.target
```

```
x = pd.DataFrame(np.c_[df['LSTAT'], df['RM']], columns = ['LSTAT','RM'])
Y = df['MEDV']

fig = plt.figure(figsize=(18,15))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(x['LSTAT'],
           x['RM'],
           Y,
           c='b')

ax.set_xlabel("LSTAT")
ax.set_ylabel("RM")
ax.set_zlabel("MEDV")

#---create a meshgrid of all the values for LSTAT and RM---
x_surf = np.arange(0, 40, 1)   #---for LSTAT---
y_surf = np.arange(0, 10, 1)   #---for RM---
x_surf, y_surf = np.meshgrid(x_surf, y_surf)

from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(x, Y)

#---calculate z(MEDC) based on the model---
z = lambda x,y: (model.intercept_ + model.coef_[0] * x + model.coef_[1] * y)

ax.plot_surface(x_surf, y_surf, z(x_surf,y_surf),
                rstride=1,
                cstride=1,
                color='None',
                alpha = 0.4)

plt.show()
```

Here, we are training the model using the entire dataset. We then make predictions by passing a combination of values for `LSTAT` (`x _ surf`) and `RM` (`y_surf`) and calculating the predicted values using the model's intercept and coefficients. The hyperplane is then plotted using the `plot_surface()` function. The end result is shown in Figure 6.8.

As the chart shown in Jupyter Notebook is static, save the preceding code snippet in a file named `boston.py` and run it in Terminal, like this:

```
$ python boston.py
```

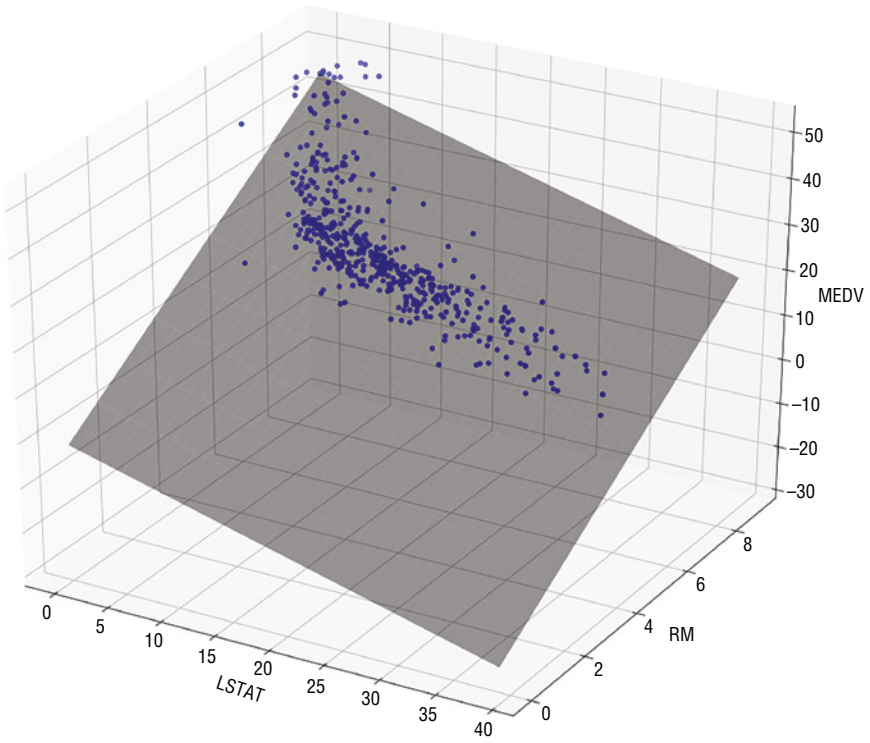

Figure 6.8: The hyperplane showing the predictions for the two features—LSTAT and RM

You will now be able to rotate the chart and move it around to have a better perspective, as shown in Figure 6.9.

Polynomial Regression

In the previous section, you saw how to apply linear regression to predict the prices of houses in the Boston area. While the result is somewhat acceptable, it is not very accurate. This is because sometimes a linear regression line might not be the best solution to capture the relationships between the features and label accurately. In some cases, a curved line might do better.

Consider the series of points shown in Figure 6.10.

The series of points are stored in a file named `polynomial.csv`:

```
x,y
1.5,1.5
2,2.5
3,4
4,4
5,4.5
6,5
```

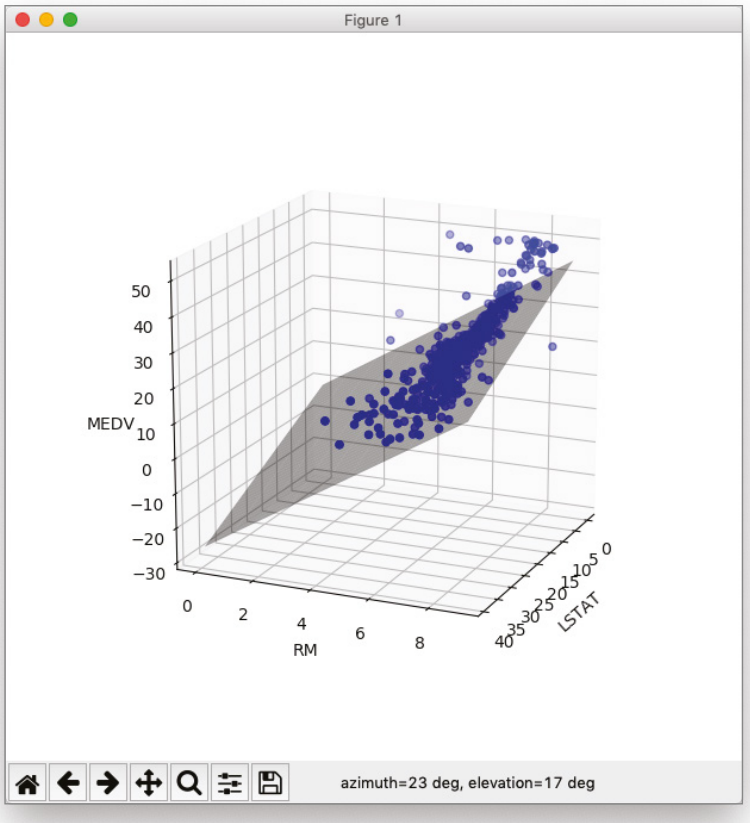

Figure 6.9: Rotating the chart to have a better view of the hyperplane

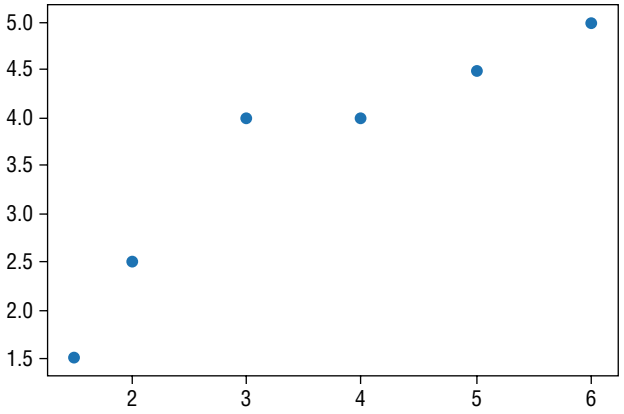

Figure 6.10: A scatter plot of points

And plotted using a scatter plot:

```
df = pd.read_csv('polynomial.csv')
plt.scatter(df.x,df.y)
```

Using linear regression, you can try to plot a straight line cutting through most of the points:

```
model = LinearRegression()

x = df.x[0:6, np.newaxis] #---convert to 2D array---
y = df.y[0:6, np.newaxis] #---convert to 2D array---

model.fit(x,y)

#---perform prediction---
y_pred = model.predict(x)

#---plot the training points---
plt.scatter(x, y, s=10, color='b')

#---plot the straight line---
plt.plot(x, y_pred, color='r')
plt.show()

#---calculate R-Squared---
print('R-Squared for training set: %.4f' % model.score(x,y))
```

You will see the straight regression line, as shown in Figure 6.11.

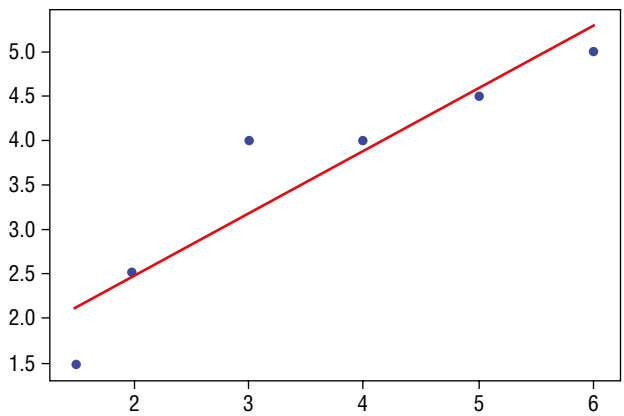

Figure 6.11: The regression line fitting the points

The R-Squared value for the training set is:

```
R-Squared for training set: 0.8658
```

We want to see if there is a more accurate way to fit the points. For instance, instead of a straight line, we want to investigate the possibility of a curved line. This is where *polynomial regression* comes in.

Formula for Polynomial Regression

Polynomial regression is an attempt to create a polynomial function that fits a set of data points.

A polynomial function of degree 1 has the following form:

$$Y = \beta_0 + \beta_1 x$$

This is the simple linear regression that we have seen in the previous chapter. *Quadratic regression* is a degree 2 polynomial:

$$Y = \beta_0 + \beta_1 x + \beta_2 x^2$$

For a polynomial of degree 3, the formula is as follows:

$$Y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$$

In general, a polynomial of degree *n* has the formula of:

$$Y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \ldots + \beta_n x^n$$

The idea behind polynomial regression is simple—find the coefficients of the polynomial function that best fits the data.

Polynomial Regression in Scikit-learn

The Scikit-learn library contains a number of classes and functions for solving polynomial regression. The `PolynomialFeatures` class takes in a number specifying the degree of the polynomial features. In the following code snippet, we are creating a quadratic equation (polynomial function of degree 2):

```
from sklearn.preprocessing import PolynomialFeatures
degree = 2
polynomial_features = PolynomialFeatures(degree = degree)
```

Using this `PolynomialFeatures` object, you can generate a new feature matrix consisting of all polynomial combinations of the features with a degree of less than or equal to the specified degree:

```
x_poly = polynomial_features.fit_transform(x)
print(x_poly)
```

You should see the following:

```
[[ 1.    1.5   2.25]
 [ 1.    2.    4.  ]
 [ 1.    3.    9.  ]
 [ 1.    4.   16.  ]
 [ 1.    5.   25.  ]
 [ 1.    6.   36.  ]]
```

The matrix that you see is generated as follows:

- The first column is always 1.
- The second column is the value of x.
- The third column is the value of x^2.

This can be verified using the `get_feature_names()` function:

```
print(polynomial_features.get_feature_names('x'))
```

It prints out the following:

```
['1', 'x', 'x^2']
```

TIP **The math behind finding the coefficients of a polynomial function is beyond the scope of this book. For those who are interested, however, check out the following link on the math behind polynomial regression:** `http://polynomialregression.drque.net/math.html`**.**

You will now use this generated matrix with the `LinearRegression` class to train your model:

```
model = LinearRegression()
model.fit(x_poly, y)
y_poly_pred = model.predict(x_poly)

#---plot the points---
plt.scatter(x, y, s=10)

#---plot the regression line---
plt.plot(x, y_poly_pred)
plt.show()
```

Figure 6.12 now shows the regression line, a nice curve trying to fit the points. You can print out the intercept and coefficients of the polynomial function:

```
print(model.intercept_)
print(model.coef_)
```

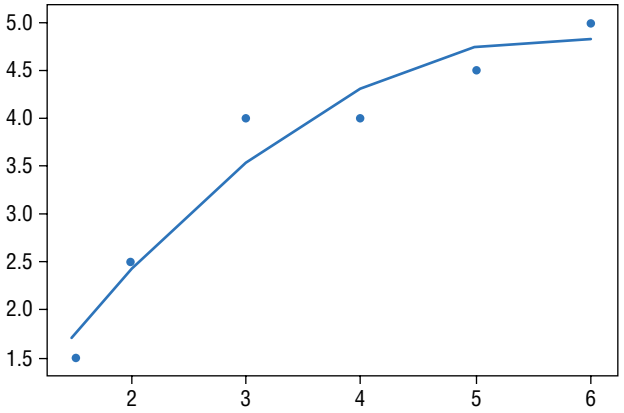

Figure 6.12: A curved line trying to fit the points

You should see the following:

```
[-0.87153912]
[[ 0.          1.98293207 -0.17239897]]
```

By plugging these numbers $Y = -0.87153912 + 1.98293207\,x + (-0.17239897)\,x^2$ into the formula $Y = \beta_0 + \beta_1 x + \beta_2 x^2$, you can make predictions using the preceding formula.

If you evaluate the regression by printing its R-Squared value,

```
print('R-Squared for training set: %.4f' % model.score(x_poly,y))
```

you should get a value of 0.9474:

```
R-Squared for training set: 0.9474
```

Can the R-Squared value be improved? Let's try a degree 3 polynomial. Using the same code and changing `degree` to `3`, you should get the curve shown in Figure 6.13 and a value of 0.9889 for R-Squared.

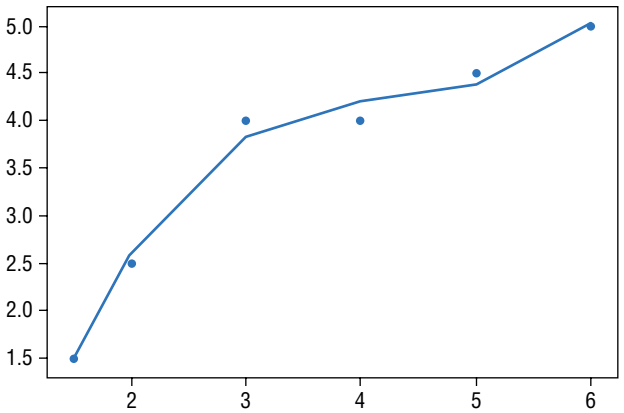

Figure 6.13: A curved line trying to fit most of the points

You now see a curve that more closely fits the points and a much-improved R-Squared value. Moreover, since raising the polynomial degree by 1 improves the R-Squared value, you might be tempted to increase it further. In fact, Figure 6.14 shows the curve when the `degree` is set to `4`. It fits all the points perfectly.

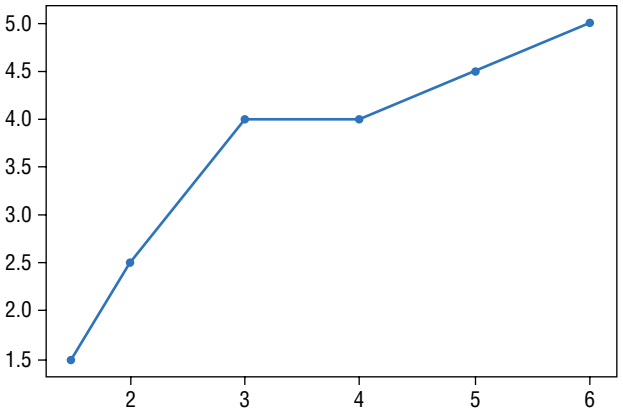

Figure 6.14: The line now fits the points perfectly

And guess what? You get an R-Squared value of 1! However, before you celebrate your success in finding the perfect algorithm in your prediction, you need to realize that while your algorithm may fit the training data perfectly, it is unlikely to perform well with new data. This is a known as *overfitting*, and the next section will discuss this topic in more detail.

Understanding Bias and Variance

The inability for a machine learning algorithm to capture the true relationship between the variables and the outcome is known as the *bias*. Figure 6.15 shows a straight line trying to fit all the points. Because it doesn't cut through all of the points, it has a high bias.

The curvy line in Figure 6.16, however, is able to fit all of the points and thus has a low bias.

While the straight line can't fit through all of the points and has high bias, when it comes to applying unseen observations, it gives a pretty good estimate. Figure 6.17 shows the testing points (in pink). The RSS (Residual Sum of Squares), which is the sum of the errors of prediction, is pretty low compared to that of the curvy line when using the same test points (see Figure 6.18).

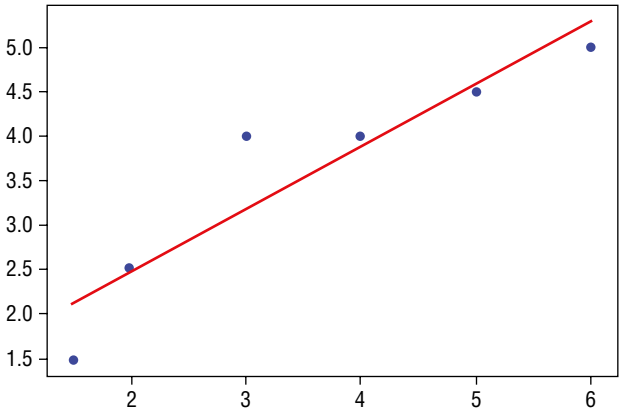

Figure 6.15: The straight line can't fit all of the points, so the bias is high

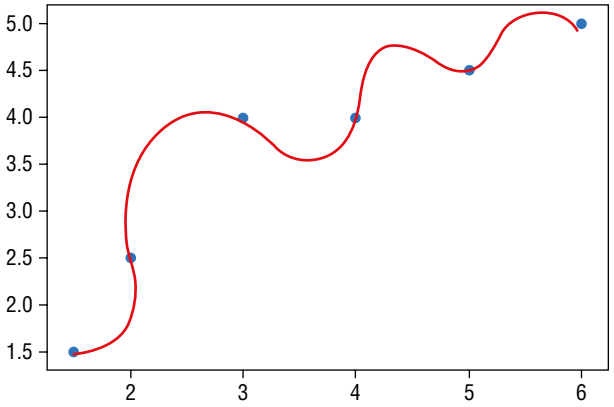

Figure 6.16: The curvy line fits all of the points, so the bias is low

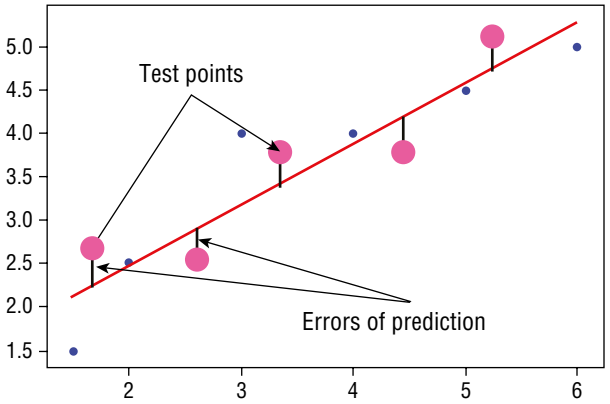

Figure 6.17: The straight line works well with unseen data, and its result does not vary much with different datasets. Hence, it has low variance.

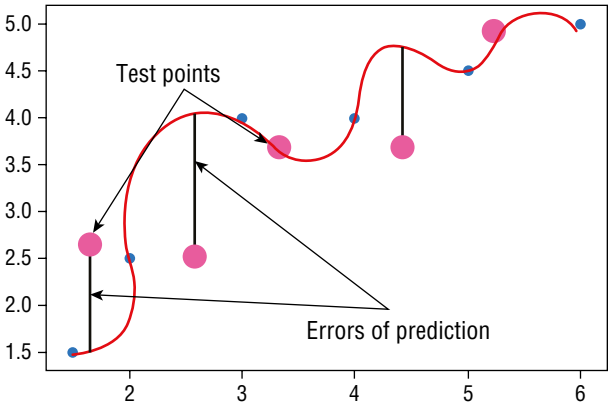

Figure 6.18: The curvy line does not work well with unseen data, and its result varies with different datasets. Hence, it has high variance.

In machine learning, the fit between the datasets is known as *variance*. In this example, the curvy line has *high variance* because it will result in vastly different RSS for different datasets. That is, you can't really predict how well it will perform with future datasets—sometimes it will do well with certain datasets and at other times it may fail badly. On the other hand, the straight line has a *low variance*, as the RSS is similar for different datasets.

TIP **In machine learning, when we try to find a curve that tries to fit all of the points perfectly, it is known as *overfitting*. On the other hand, if we have a line that does not fit most points, it is known as *underfitting*.**

Ideally, we should find a line that accurately expresses the relationships between the independent variables and that of the outcome. Expressed in terms of bias and variance, the ideal algorithm should have the following:

High bias, with the line hugging as many points as possible

Low variance, with the line resulting in consistent predictions using different datasets

Figure 6.19 shows such an ideal curve—high bias and low variance.

To strike a balance between finding a simple model and a complex model, you can use techniques such as *Regularization*, *Bagging*, and *Boosting*:

- *Regularization* is a technique that automatically penalizes the extra features you used in your modeling.
- *Bagging* (or *bootstrap aggregation*) is a specific type of machine learning process that uses *ensemble learning* to evolve machine learning models. Bagging uses a subset of the data and each sample trains a weaker learner. The weak learners can then be combined (through averaging or max vote) to create a strong learner that can make accurate predictions.

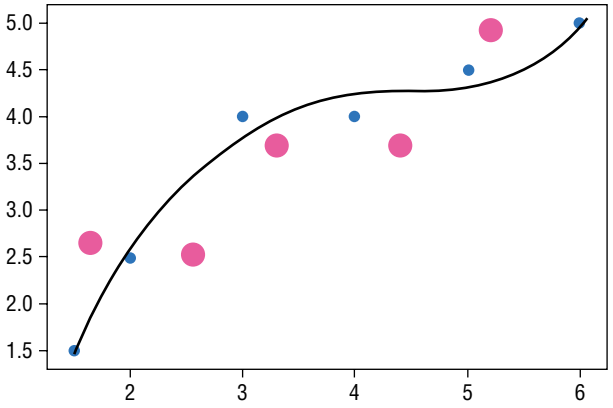

Figure 6.19: You should aim for a line that has high bias and low variance

- *Boosting* is also similar to Bagging, except that it uses all of the data to train each learner, but data points that were misclassified by previous learners are given more weight so that subsequent learners will give more focus to them during training.

TIP ***Ensemble learning*** **is a technique where you use several models working together on a single dataset and then combine its result.**

Using Polynomial Multiple Regression on the Boston Dataset

Earlier in this chapter, you used multiple linear regression and trained a model based on the Boston dataset. After learning about the polynomial regression in the previous section, now let's try to apply it to the Boston dataset and see if we can improve the model.

As usual, let's load the data and split the dataset into training and testing sets:

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_boston

dataset = load_boston()

df = pd.DataFrame(dataset.data, columns=dataset.feature_names)
df['MEDV'] = dataset.target

x = pd.DataFrame(np.c_[df['LSTAT'], df['RM']], columns = ['LSTAT','RM'])
Y = df['MEDV']
```

```
from sklearn.model_selection import train_test_split
x_train, x_test, Y_train, Y_test = train_test_split(x, Y, test_size = 0.3,
                                                    random_state=5)
```

You then use the polynomial function with degree 2:

```
#---use a polynomial function of degree 2---
degree = 2
polynomial_features= PolynomialFeatures(degree = degree)
x_train_poly = polynomial_features.fit_transform(x_train)
```

When using a polynomial function of degree 2 on two independent variables x_1 and x_2, the formula becomes:

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1^2 + \beta_4 x_1 x_2 + \beta_5 x_2^2$$

where Y is the dependent variable, β_0 is the intercept, and β_1, β_2, β_3, and β_4 are the coefficients of the various combinations of the two features x_1 and x_2, respectively.

You can verify this by printing out the feature names:

```
#---print out the formula---
print(polynomial_features.get_feature_names(['x','y']))
```

You should see the following, which coincides with the formula:

```
# ['1', 'x', 'y', 'x^2', 'x y', 'y^2']
```

TIP **Knowing the polynomial function formula is useful when plotting the 3D hyperplane, which you will do shortly.**

You can then train your model using the `LinearRegression` class:

```
model = LinearRegression()
model.fit(x_train_poly, Y_train)
```

Now let's evaluate the model using the testing set:

```
x_test_poly = polynomial_features.fit_transform(x_test)
print('R-Squared: %.4f' % model.score(x_test_poly,
                                      Y_test))
```

You will see the result as follows:

```
R-Squared: 0.7340
```

You can also print the intercept and coefficients:

```
print(model.intercept_)
print(model.coef_)
```

You should see the following:

```
26.9334305238
[  0.00000000e+00   1.47424550e+00  -6.70204730e+00   7.93570743e-04
  -3.66578385e-01   1.17188007e+00]
```

With these values, the formula now becomes:

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1^2 + \beta_4 x_1 x_2 + \beta_5 x_2^2$$

$$Y = 26.9334305238 + 1.47424550e+00\, x_1 + (-6.70204730e+00)\, x_2 + 7.93570743e-04\, x_1^2 + (-3.66578385e-01)\, x_1 x_2 + 1.17188007e+00\, x_2^2$$

Plotting the 3D Hyperplane

Since you know the intercept and coefficients of the polynomial multiple regression function, you can plot out the 3D hyperplane of function easily. Save the following code snippet as a file named `boston2.py`:

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

from mpl_toolkits.mplot3d import Axes3D
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_boston

dataset = load_boston()

df = pd.DataFrame(dataset.data, columns=dataset.feature_names)
df['MEDV'] = dataset.target

x = pd.DataFrame(np.c_[df['LSTAT'], df['RM']], columns = ['LSTAT','RM'])
Y = df['MEDV']

fig = plt.figure(figsize=(18,15))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(x['LSTAT'],
           x['RM'],
           Y,
           c='b')
```

```
ax.set_xlabel("LSTAT")
ax.set_ylabel("RM")
ax.set_zlabel("MEDV")

#---create a meshgrid of all the values for LSTAT and RM---
x_surf = np.arange(0, 40, 1)   #---for LSTAT---
y_surf = np.arange(0, 10, 1)   #---for RM---
x_surf, y_surf = np.meshgrid(x_surf, y_surf)

#---use a polynomial function of degree 2---
degree = 2
polynomial_features= PolynomialFeatures(degree = degree)
x_poly = polynomial_features.fit_transform(x)
print(polynomial_features.get_feature_names(['x','y']))

#---apply linear regression---
model = LinearRegression()
model.fit(x_poly, Y)

#---calculate z(MEDC) based on the model---
z = lambda x,y: (model.intercept_ +
                 (model.coef_[1] * x) +
                 (model.coef_[2] * y) +
                 (model.coef_[3] * x**2) +
                 (model.coef_[4] * x*y) +
                 (model.coef_[5] * y**2))

ax.plot_surface(x_surf, y_surf, z(x_surf,y_surf),
                rstride=1,
                cstride=1,
                color='None',
                alpha = 0.4)

plt.show()
```

To run the code, type the following in Terminal:

```
$ python boston2.py
```

You will see the 3D chart, as shown in Figure 6.20.

You can drag to rotate the chart. Figure 6.21 shows the different perspectives of the hyperplane.

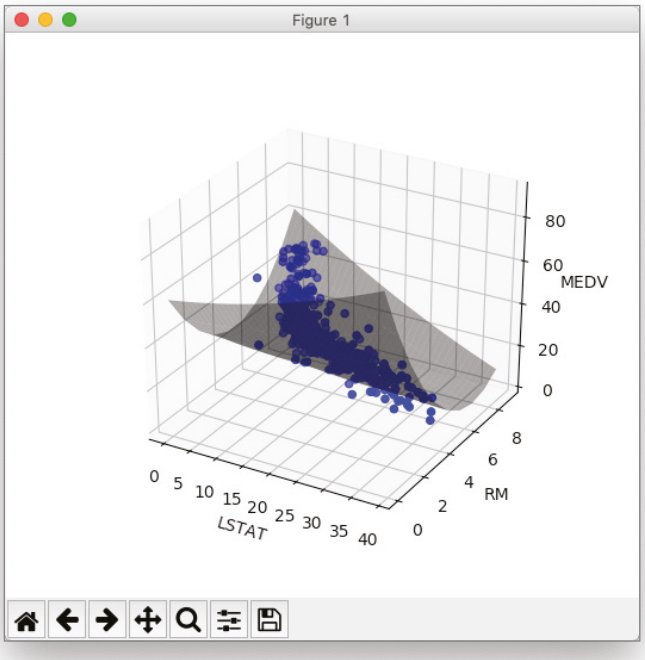

Figure 6.20: The hyperplane in the polynomial multiple regression

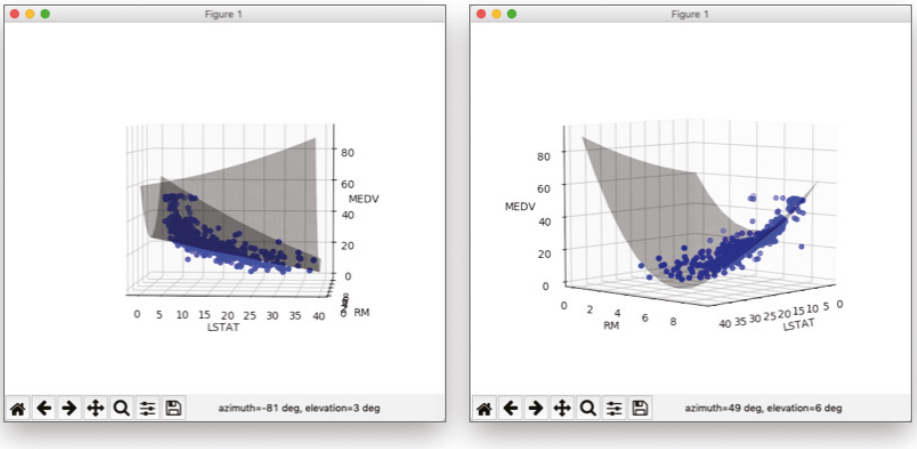

Figure 6.21: Rotate the chart to see the different perspectives of the hyperplane

Summary

In this chapter, you learned about the various types of linear regression. In particular, you learned about the following:

Multiple Regression Linear relationships between two or more independent variables and one dependent variable.

Polynomial Regression Modeling the relationship between one independent variable and one dependent variable using an n^{th} degree polynomial function.

Polynomial Multiple Regression Modeling the relationship between two or more independent variables and one dependent variable using an n^{th} degree polynomial function.

You also learned how to plot the hyperplane showing the relationships between two independent variables and the label.