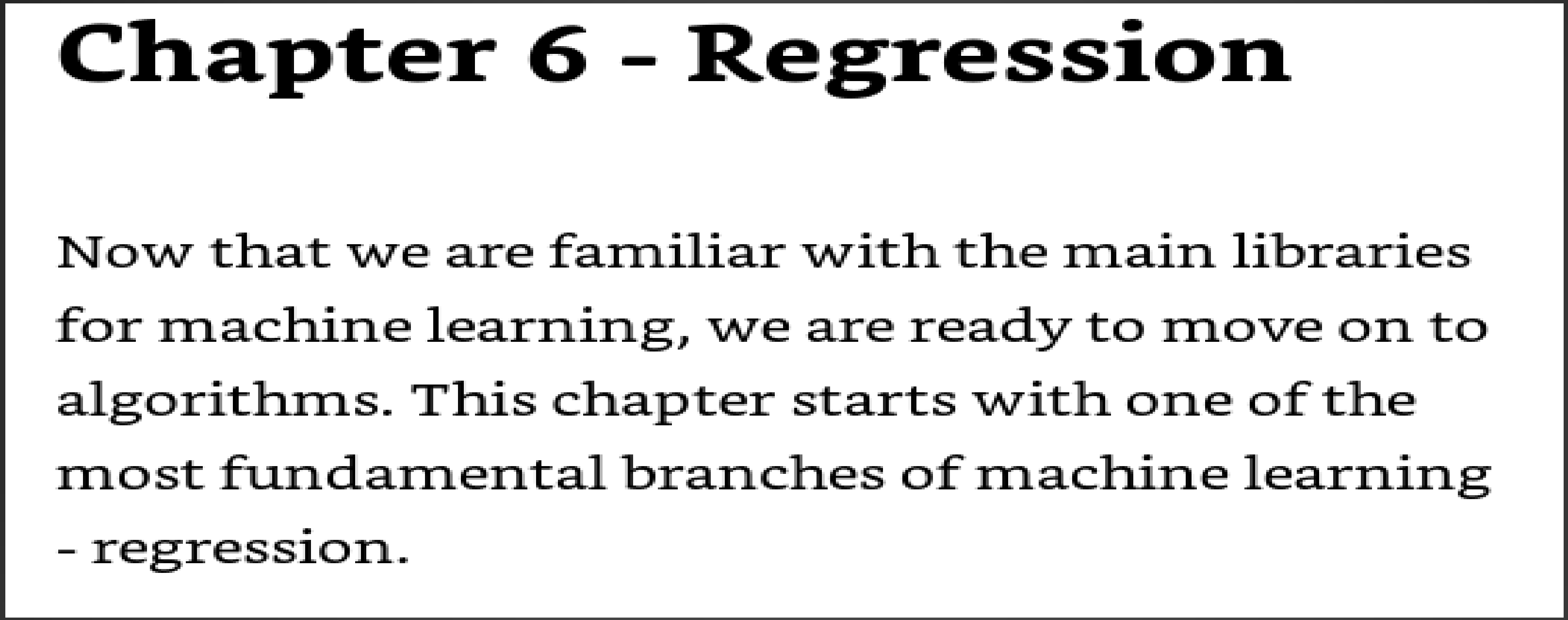
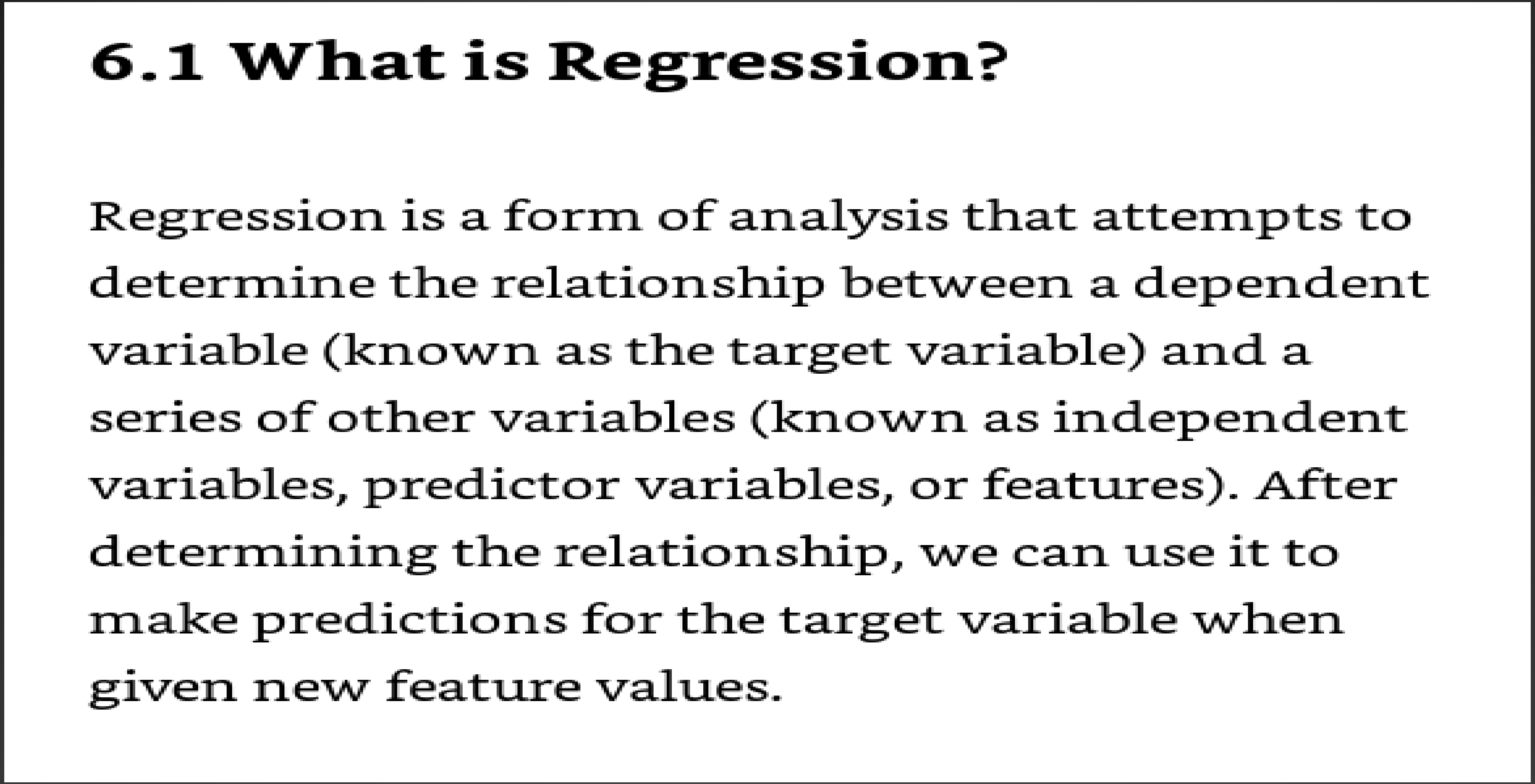
Linear Regression with Scikit-Learn

Sagara Samarawickrama



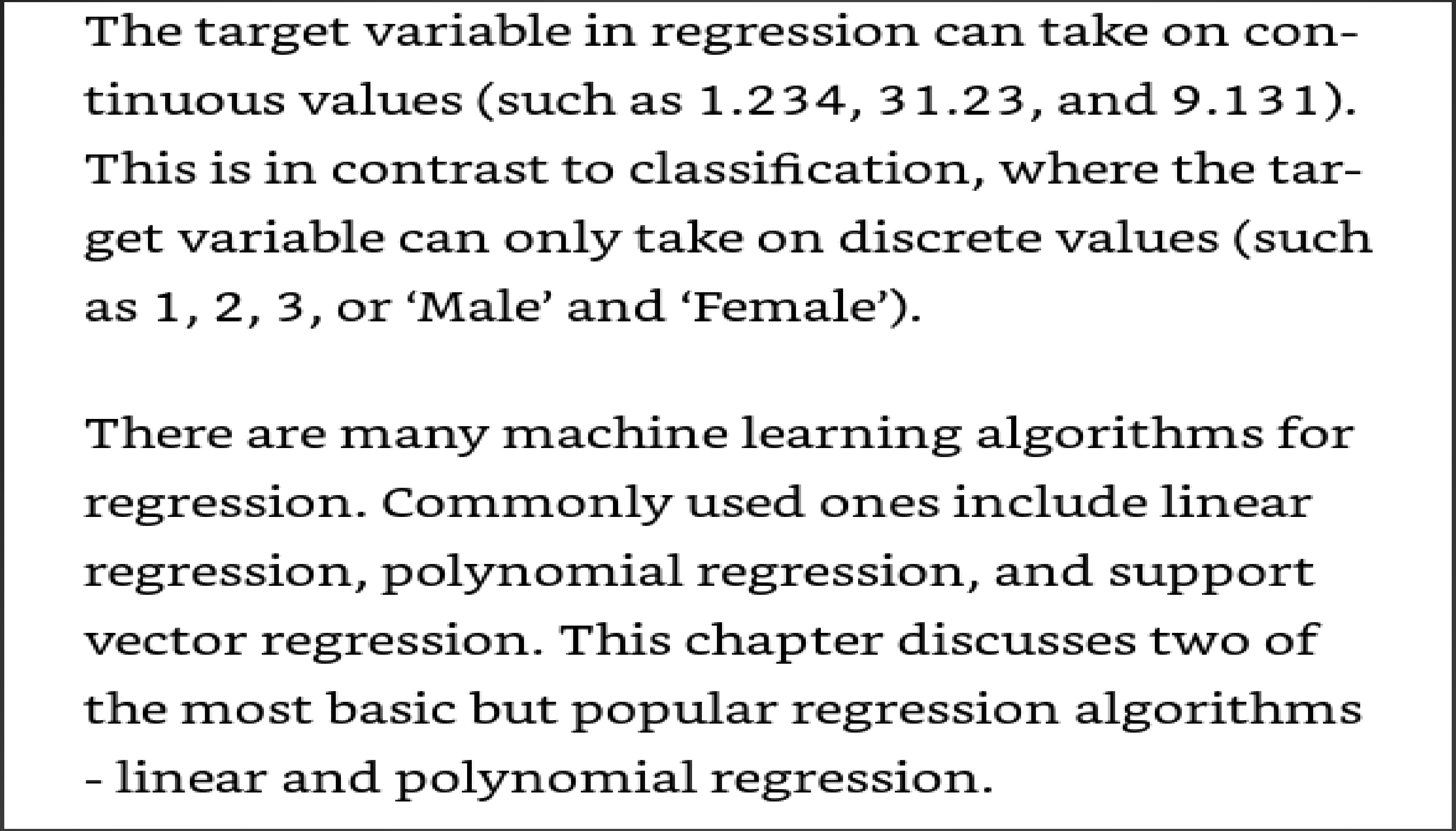
Chapter 6 - Regression

Now that we are familiar with the main libraries for machine learning, we are ready to move on to algorithms. This chapter starts with one of the most fundamental branches of machine learning - regression.

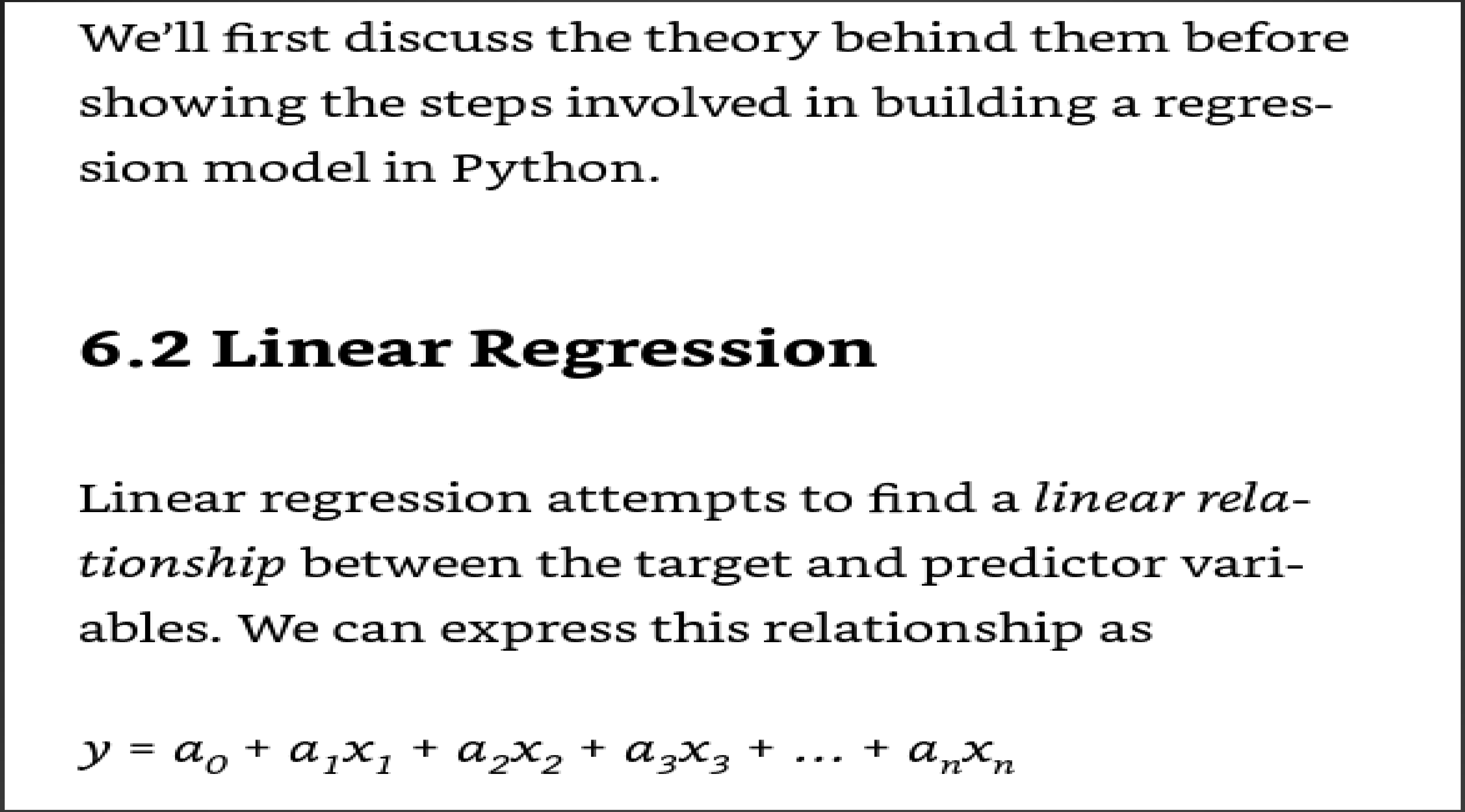


6.1 What is Regression?

Regression is a form of analysis that attempts to determine the relationship between a dependent variable (known as the target variable) and a series of other variables (known as independent variables, predictor variables, or features). After determining the relationship, we can use it to make predictions for the target variable when given new feature values.


The target variable in regression can take on continuous values (such as 1.234, 31.23, and 9.131). This is in contrast to classification, where the target variable can only take on discrete values (such as 1, 2, 3, or ‘Male’ and ‘Female’).

There are many machine learning algorithms for regression. Commonly used ones include linear regression, polynomial regression, and support vector regression. This chapter discusses two of the most basic but popular regression algorithms - linear and polynomial regression.

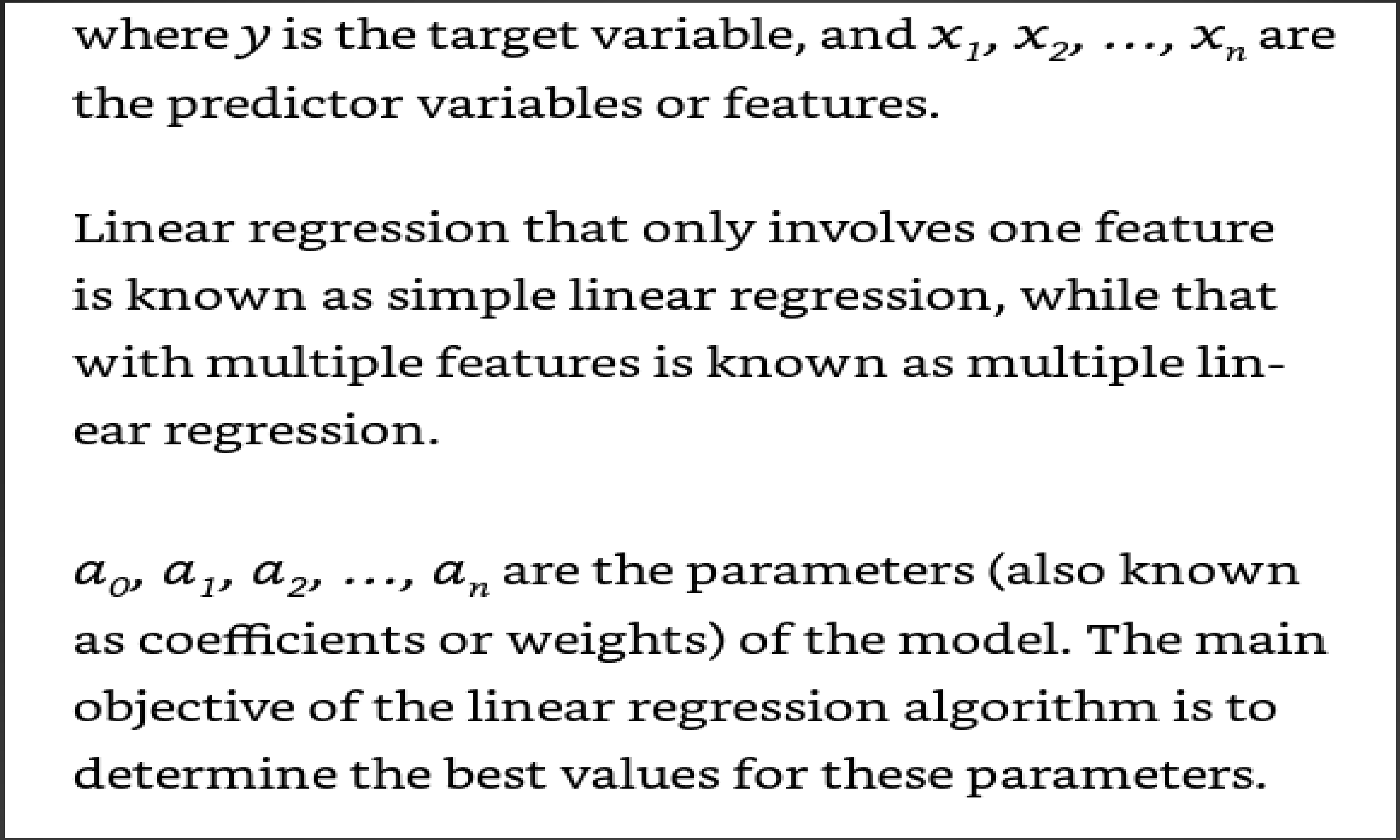


We'll first discuss the theory behind them before showing the steps involved in building a regression model in Python.

6.2 Linear Regression

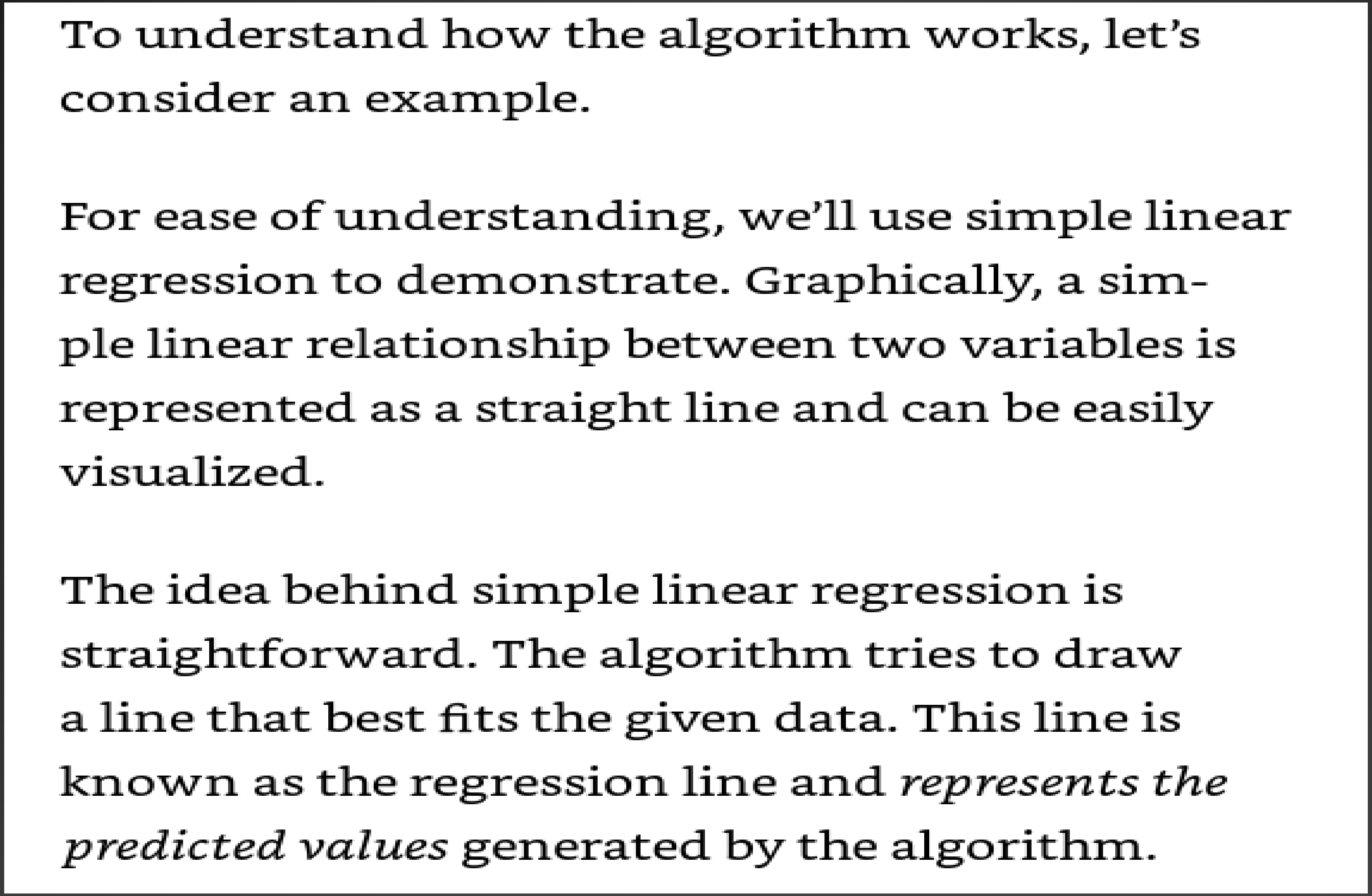
Linear regression attempts to find a *linear relationship* between the target and predictor variables. We can express this relationship as

$$y = a_0 + a_1x_1 + a_2x_2 + a_3x_3 + \ldots + a_nx_n$$


where y is the target variable, and x_1, x_2, ..., x_n are the predictor variables or features.

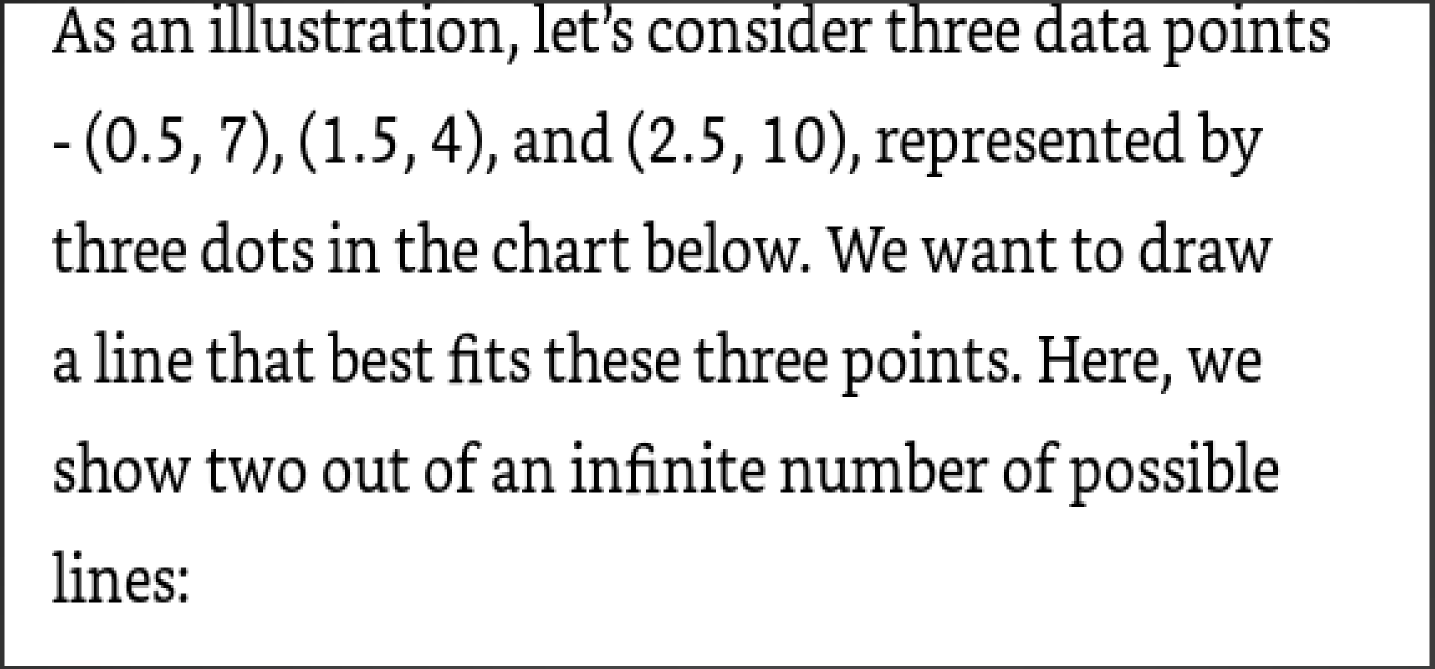
Linear regression that only involves one feature is known as simple linear regression, while that with multiple features is known as multiple linear regression.

a_0, a_1, a_2, ..., a_n are the parameters (also known as coefficients or weights) of the model. The main objective of the linear regression algorithm is to determine the best values for these parameters.


To understand how the algorithm works, let's consider an example.

For ease of understanding, we'll use simple linear regression to demonstrate. Graphically, a sim-ple linear relationship between two variables is represented as a straight line and can be easily visualized.

The idea behind simple linear regression is straightforward. The algorithm tries to draw a line that best fits the given data. This line is known as the regression line and *represents the predicted values* generated by the algorithm.


As an illustration, let's consider three data points - (0.5, 7), (1.5, 4), and (2.5, 10), represented by three dots in the chart below. We want to draw a line that best fits these three points. Here, we show two out of an infinite number of possible lines:

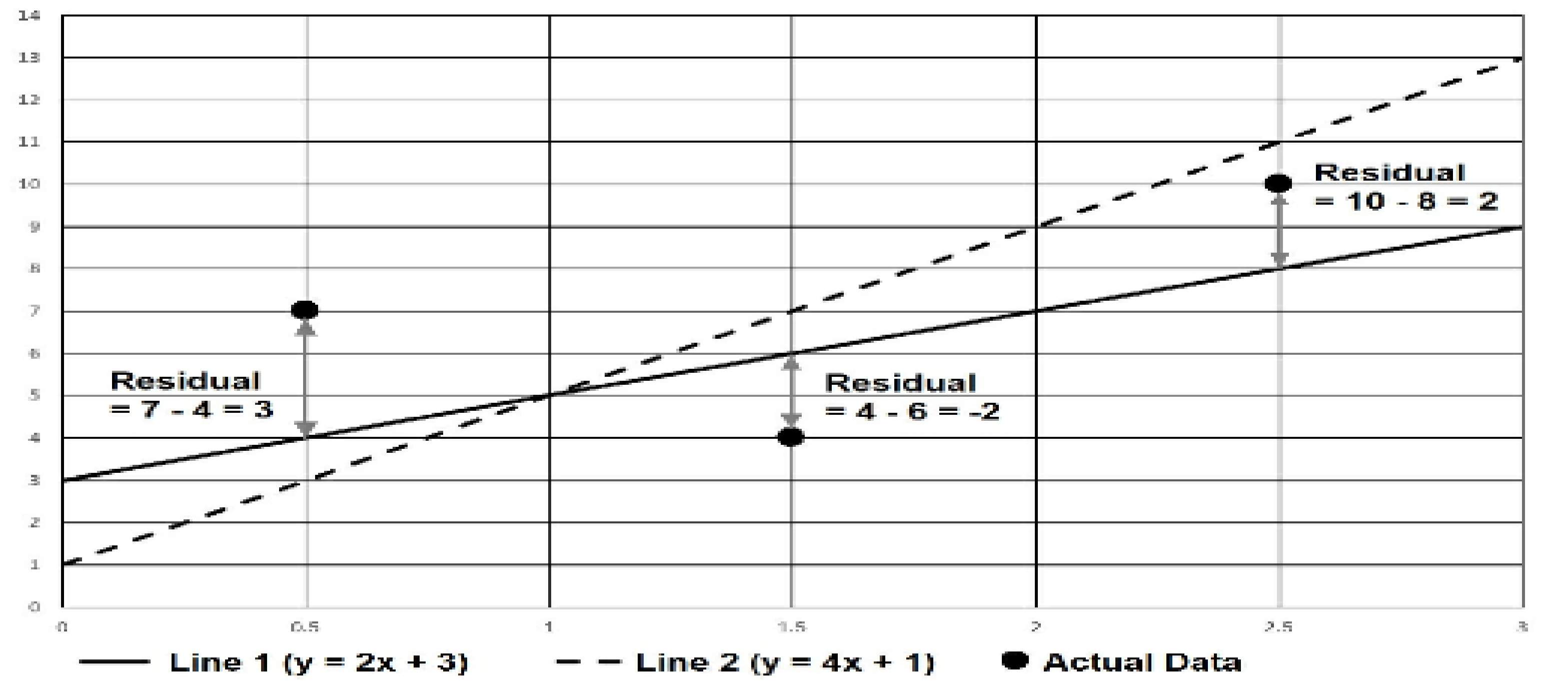
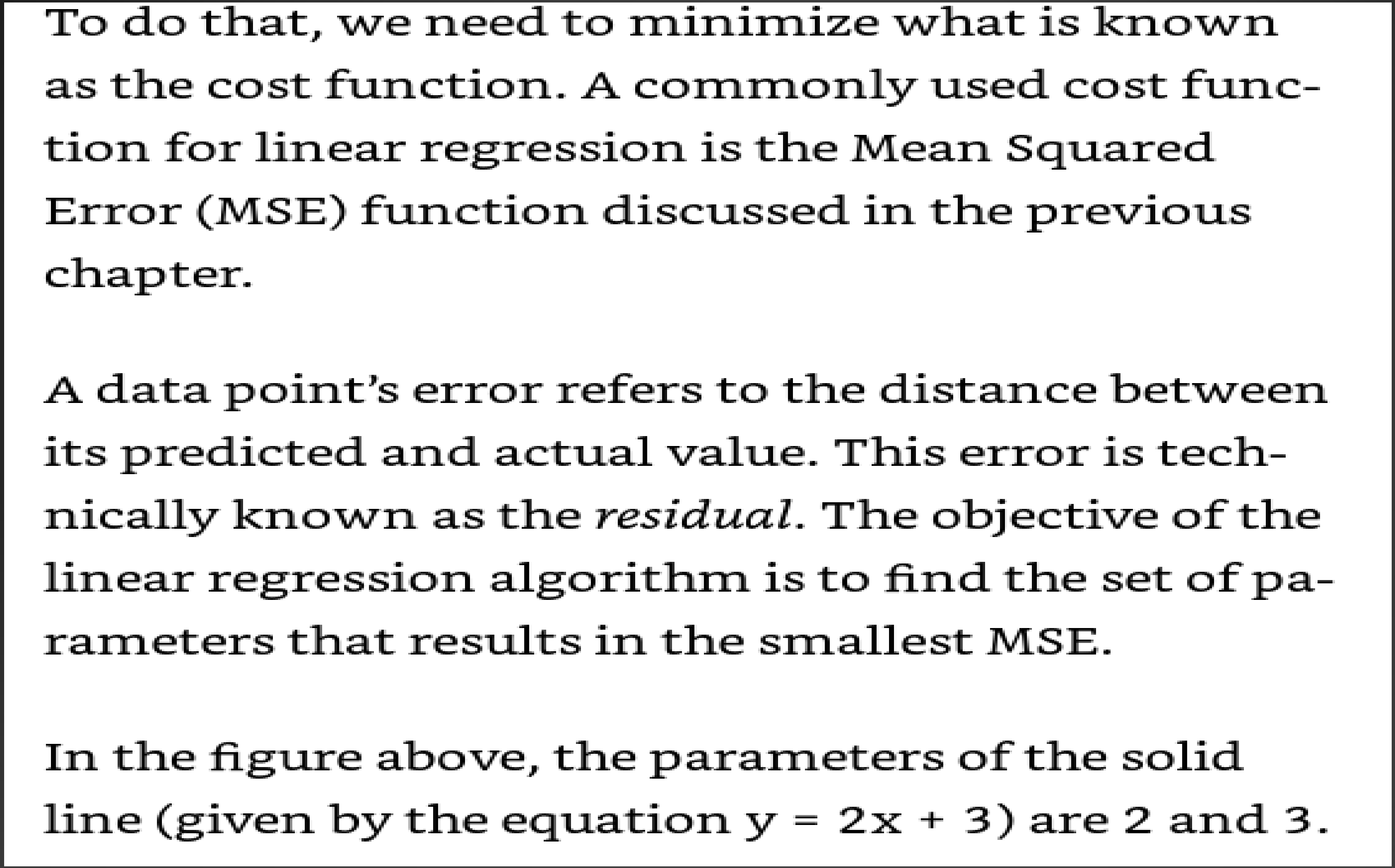


Figure 6.1: Comparing two regression lines

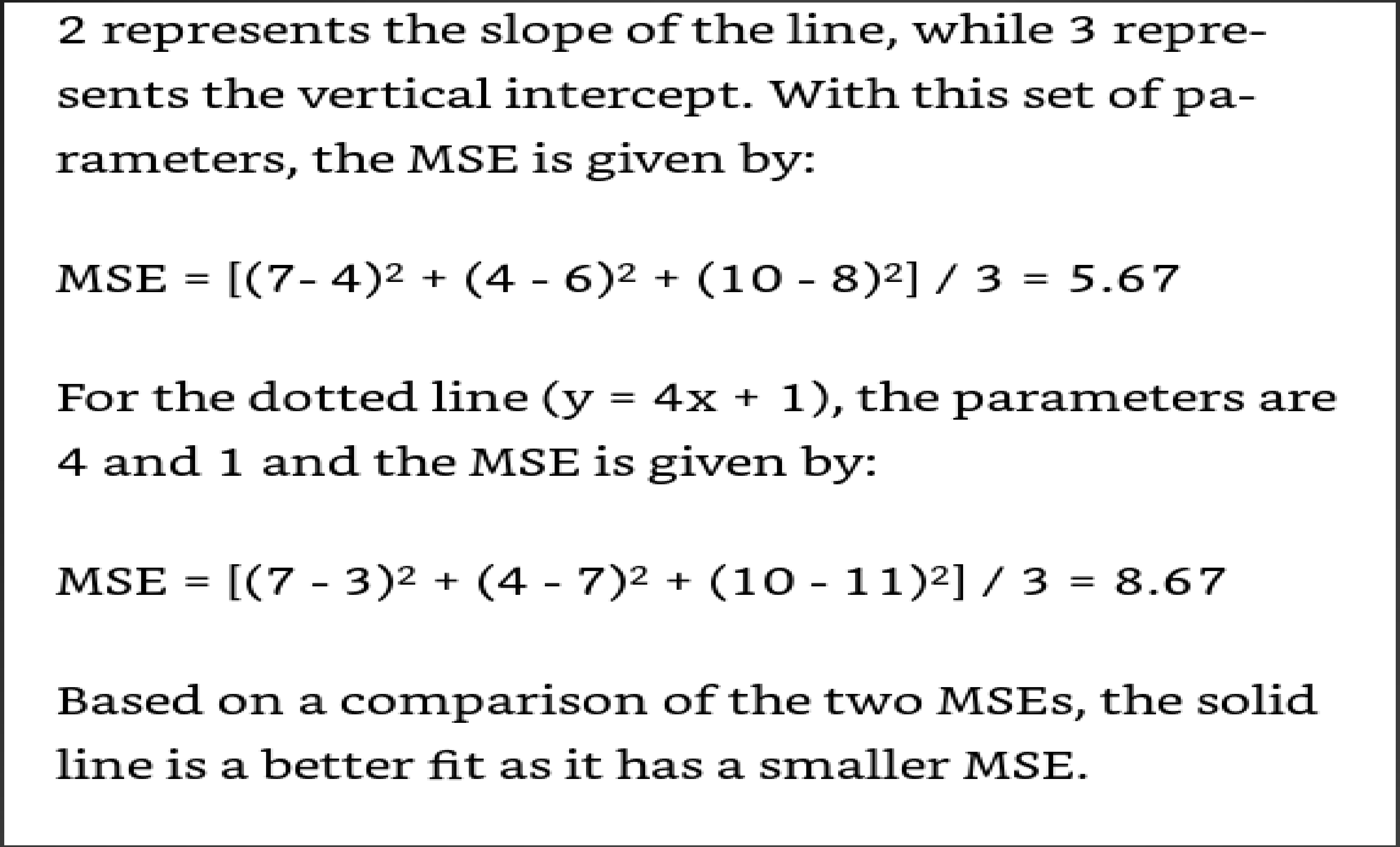
How do we choose the best line?



To do that, we need to minimize what is known as the cost function. A commonly used cost function for linear regression is the Mean Squared Error (MSE) function discussed in the previous chapter.

A data point's error refers to the distance between its predicted and actual value. This error is technically known as the *residual*. The objective of the linear regression algorithm is to find the set of parameters that results in the smallest MSE.

In the figure above, the parameters of the solid line (given by the equation $y = 2x + 3$) are 2 and 3.



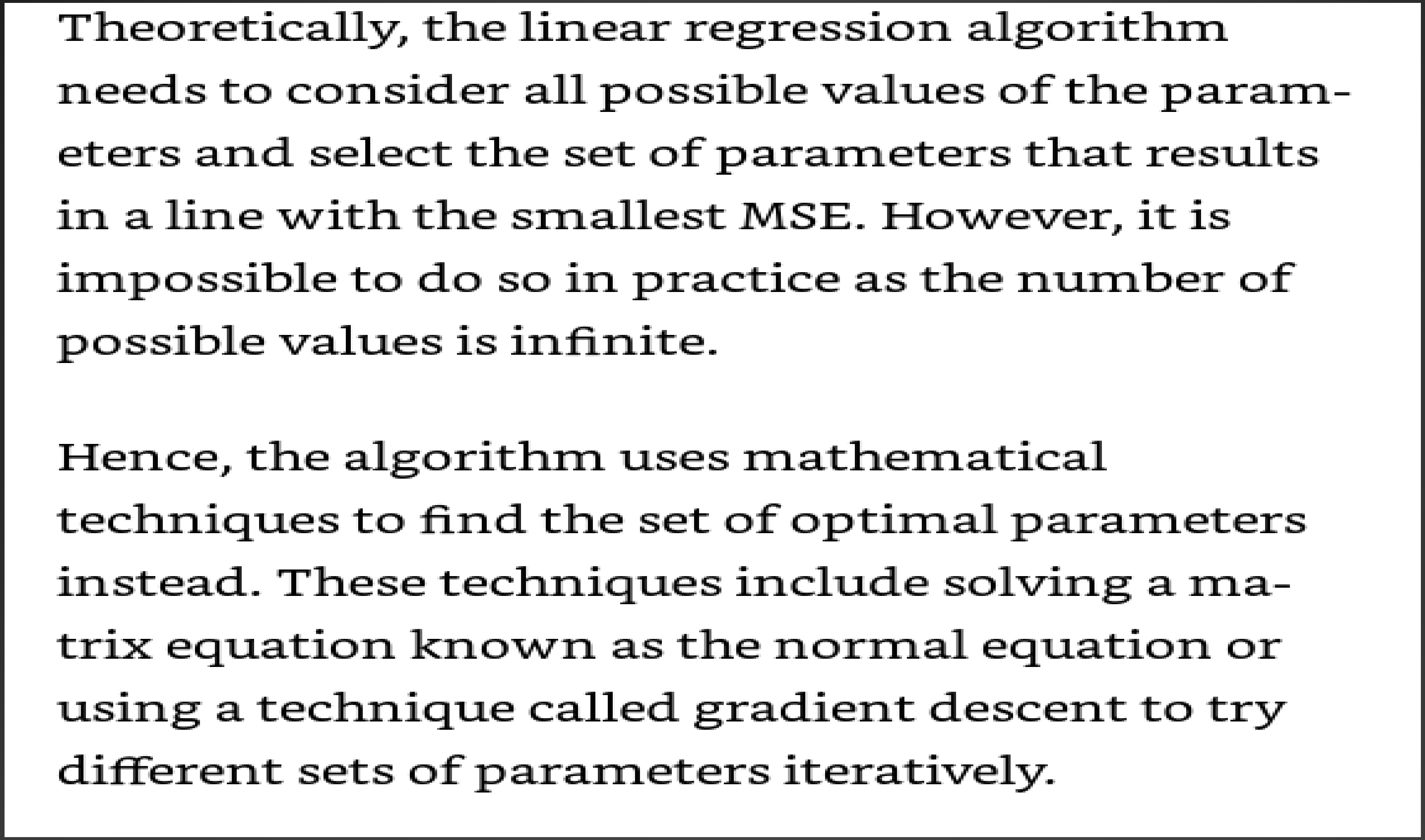
2 represents the slope of the line, while 3 represents the vertical intercept. With this set of parameters, the MSE is given by:

$$MSE = [(7 - 4)^2 + (4 - 6)^2 + (10 - 8)^2] / 3 = 5.67$$

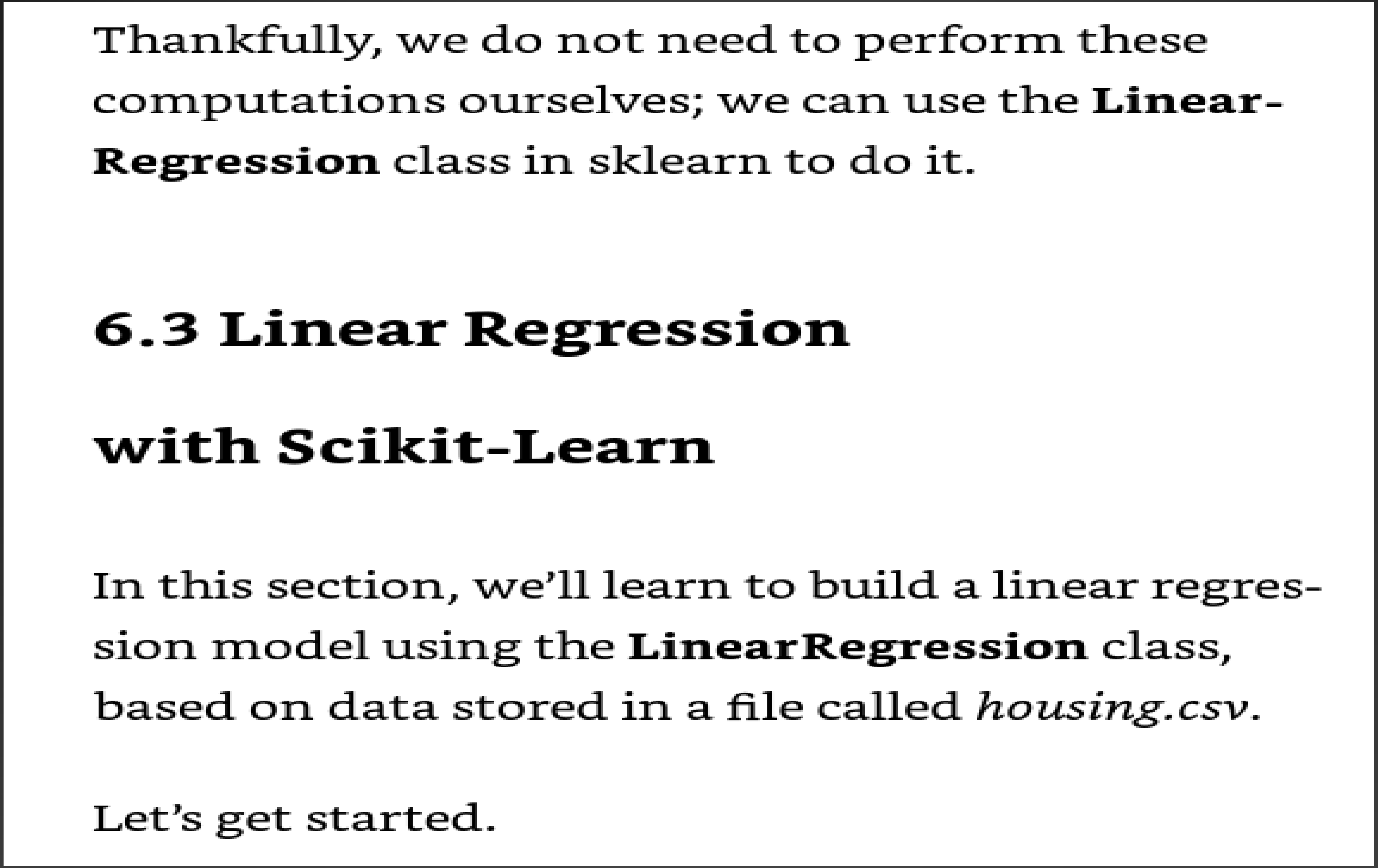
For the dotted line ($y = 4x + 1$), the parameters are 4 and 1 and the MSE is given by:

$$MSE = [(7 - 3)^2 + (4 - 7)^2 + (10 - 11)^2] / 3 = 8.67$$

Based on a comparison of the two MSEs, the solid line is a better fit as it has a smaller MSE.


Theoretically, the linear regression algorithm needs to consider all possible values of the parameters and select the set of parameters that results in a line with the smallest MSE. However, it is impossible to do so in practice as the number of possible values is infinite.

Hence, the algorithm uses mathematical techniques to find the set of optimal parameters instead. These techniques include solving a matrix equation known as the normal equation or using a technique called gradient descent to try different sets of parameters iteratively.

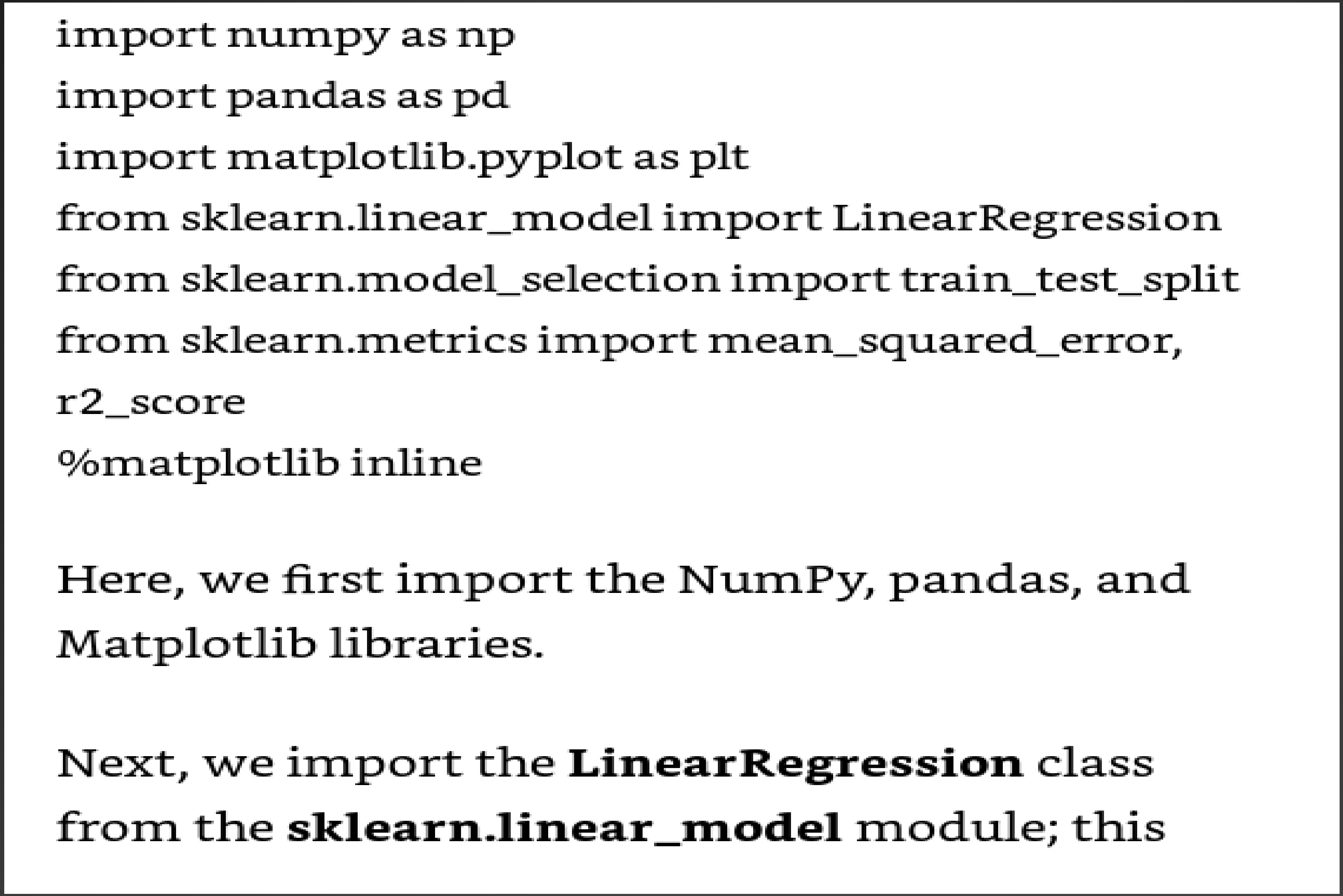


Thankfully, we do not need to perform these computations ourselves; we can use the **Linear-Regression** class in sklearn to do it.

6.3 Linear Regression with Scikit-Learn

In this section, we'll learn to build a linear regression model using the **LinearRegression** class, based on data stored in a file called *housing.csv*.

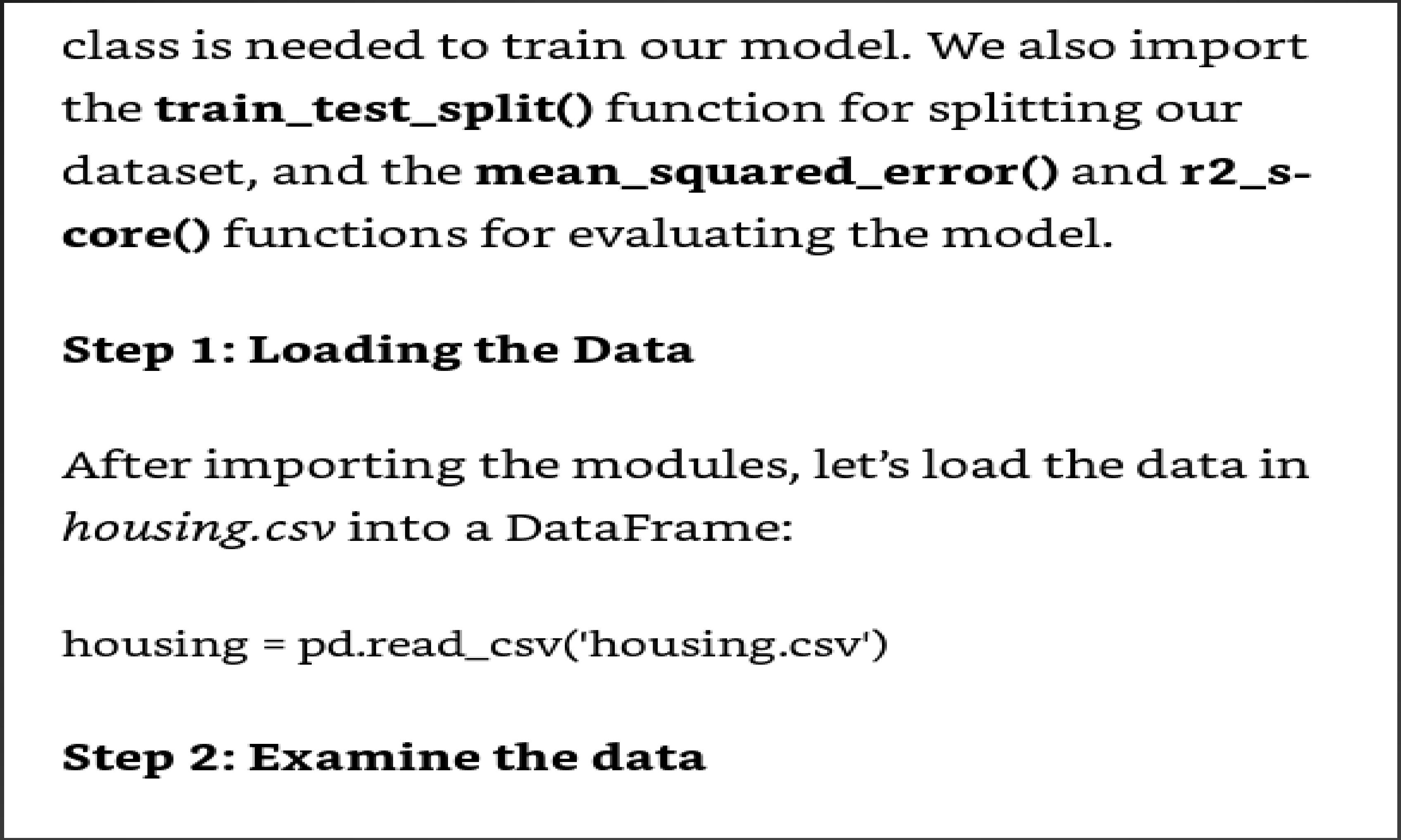
Let's get started.



```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error,
r2_score
%matplotlib inline
```

Here, we first import the NumPy, pandas, and Matplotlib libraries.

Next, we import the **LinearRegression** class from the **sklearn.linear_model** module; this

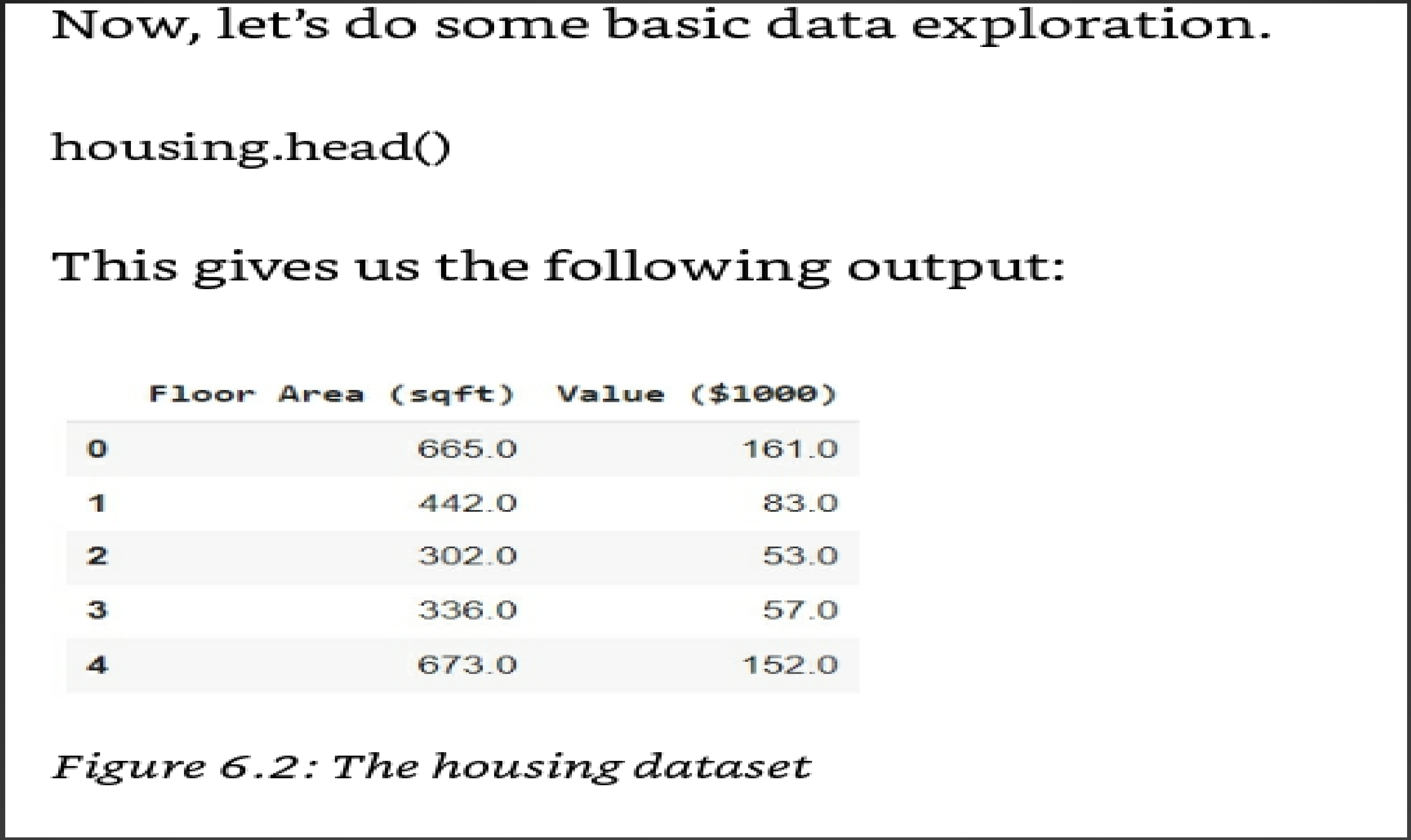

class is needed to train our model. We also import the **train_test_split()** function for splitting our dataset, and the **mean_squared_error()** and **r2_s-core()** functions for evaluating the model.

Step 1: Loading the Data

After importing the modules, let’s load the data in *housing.csv* into a DataFrame:

```
housing = pd.read_csv('housing.csv')
```

Step 2: Examine the data



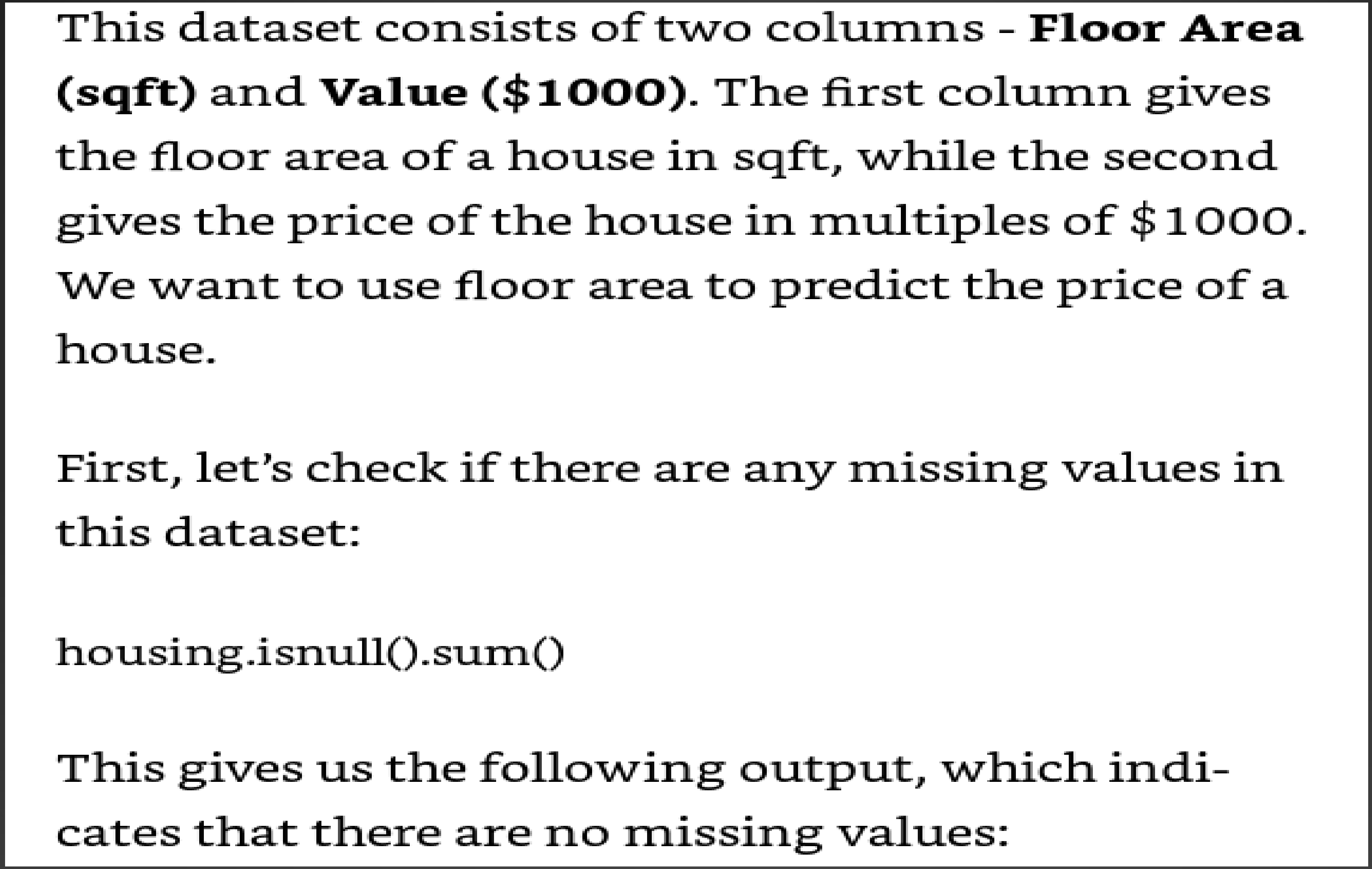
Now, let’s do some basic data exploration.

```
housing.head()
```

This gives us the following output:

	Floor Area (sqft)	Value ($1000)
0	665.0	161.0
1	442.0	83.0
2	302.0	53.0
3	336.0	57.0
4	673.0	152.0

Figure 6.2: The housing dataset

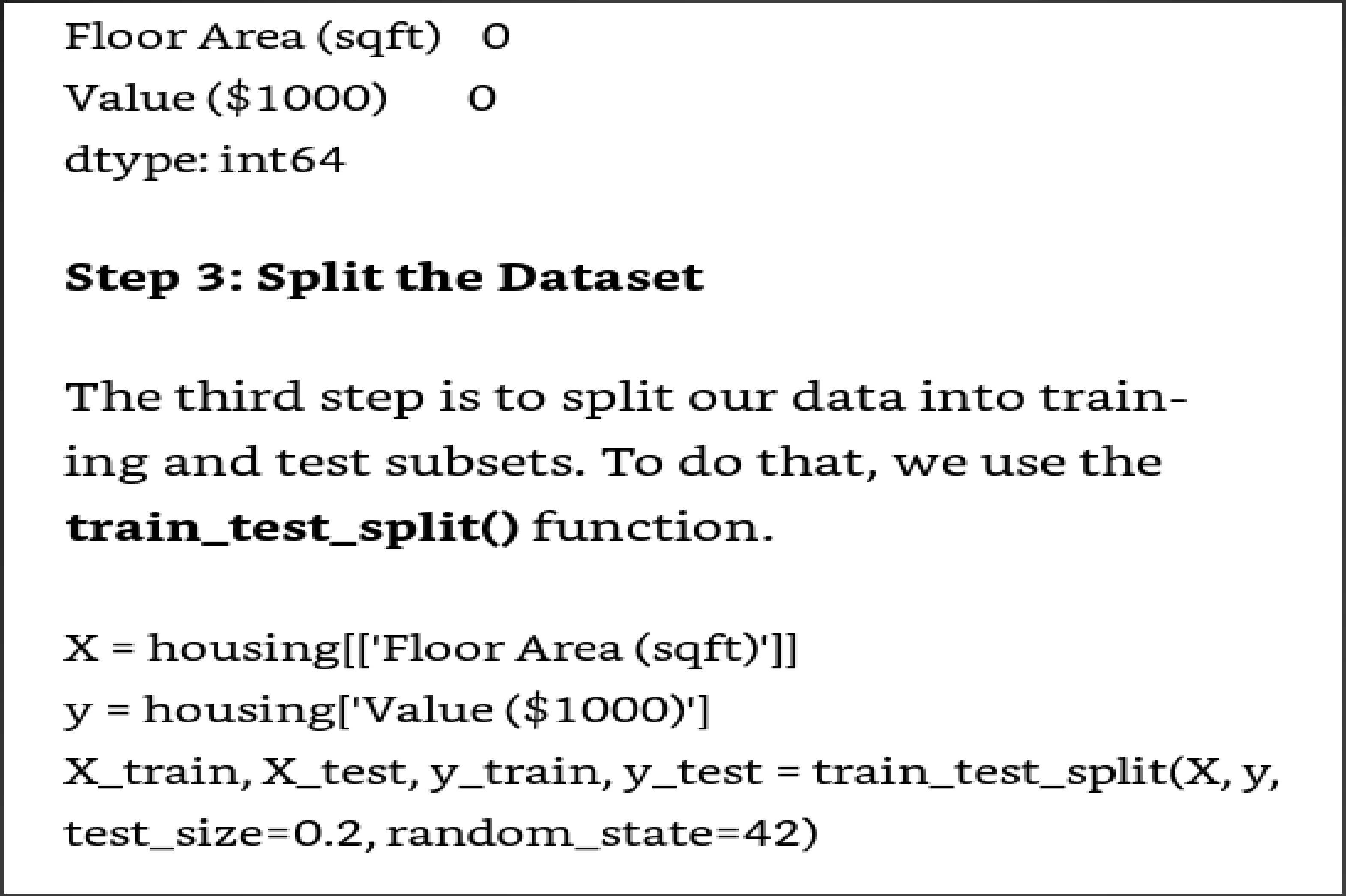


This dataset consists of two columns - **Floor Area (sqft)** and **Value ($1000)**. The first column gives the floor area of a house in sqft, while the second gives the price of the house in multiples of $1000. We want to use floor area to predict the price of a house.

First, let's check if there are any missing values in this dataset:

```
housing.isnull().sum()
```

This gives us the following output, which indi-cates that there are no missing values:

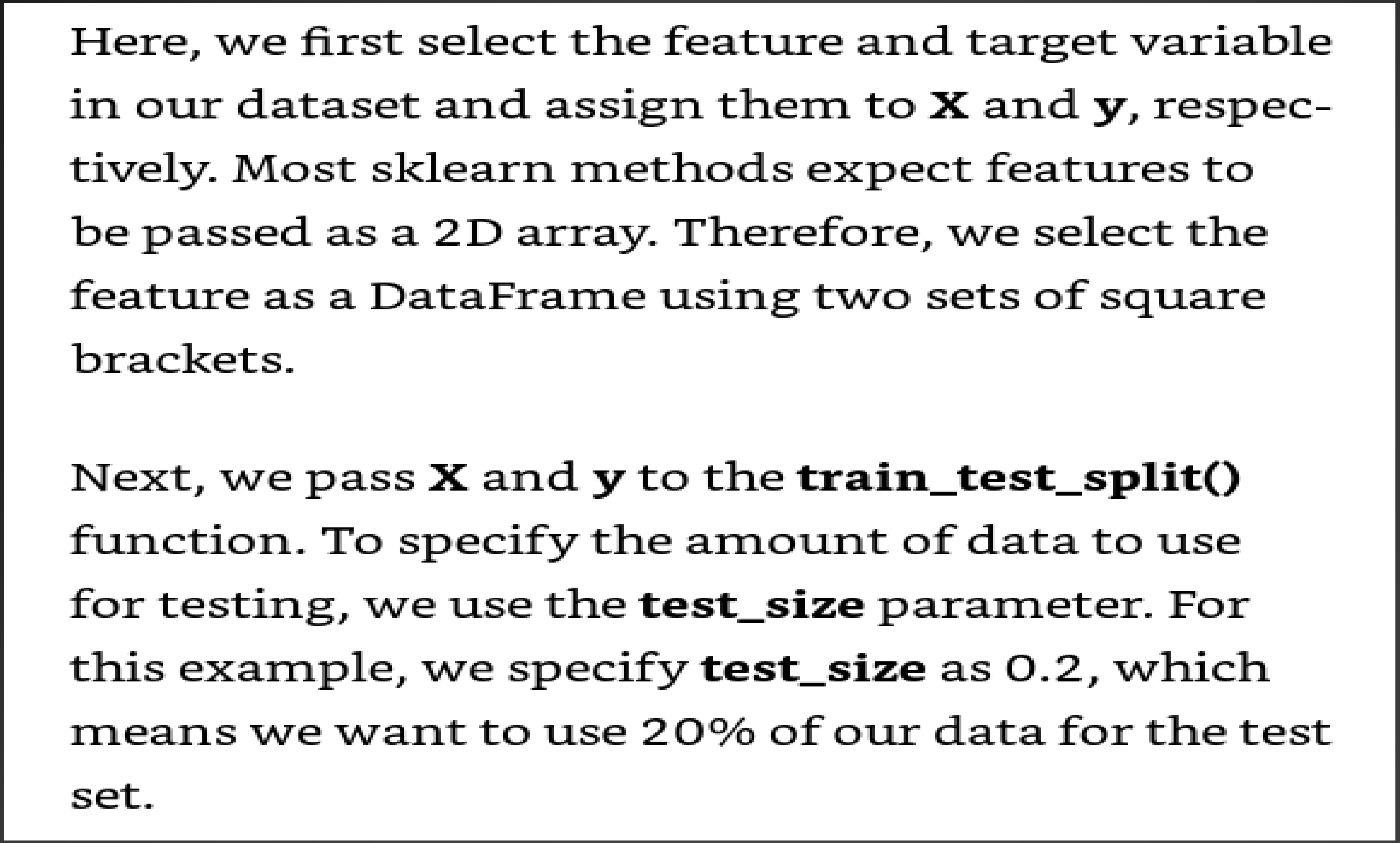


```
Floor Area (sqft)   0
Value ($1000)       0
dtype: int64
```

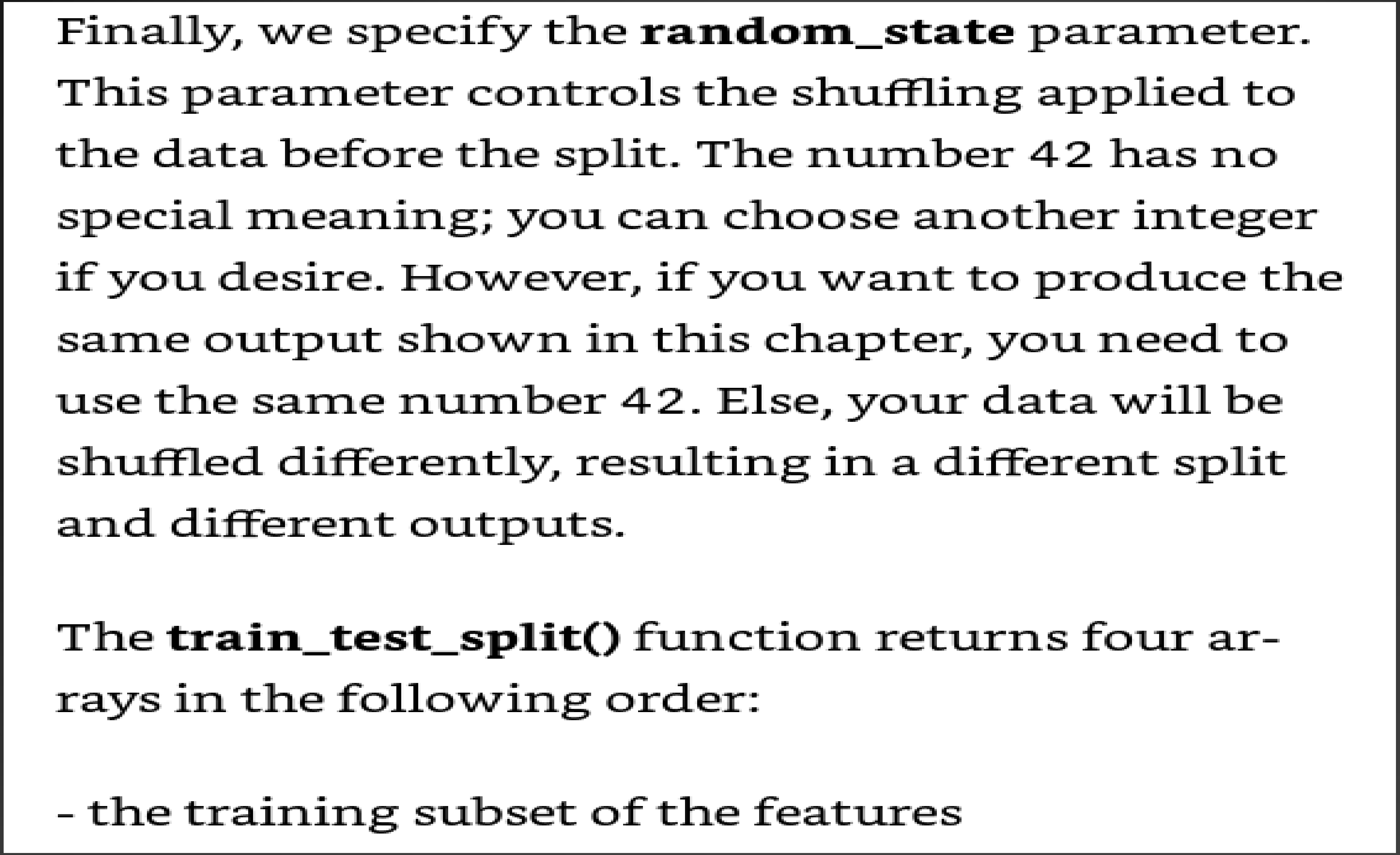
Step 3: Split the Dataset

The third step is to split our data into training and test subsets. To do that, we use the **train_test_split()** function.

```
X = housing[['Floor Area (sqft)']]
y = housing['Value ($1000)']
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

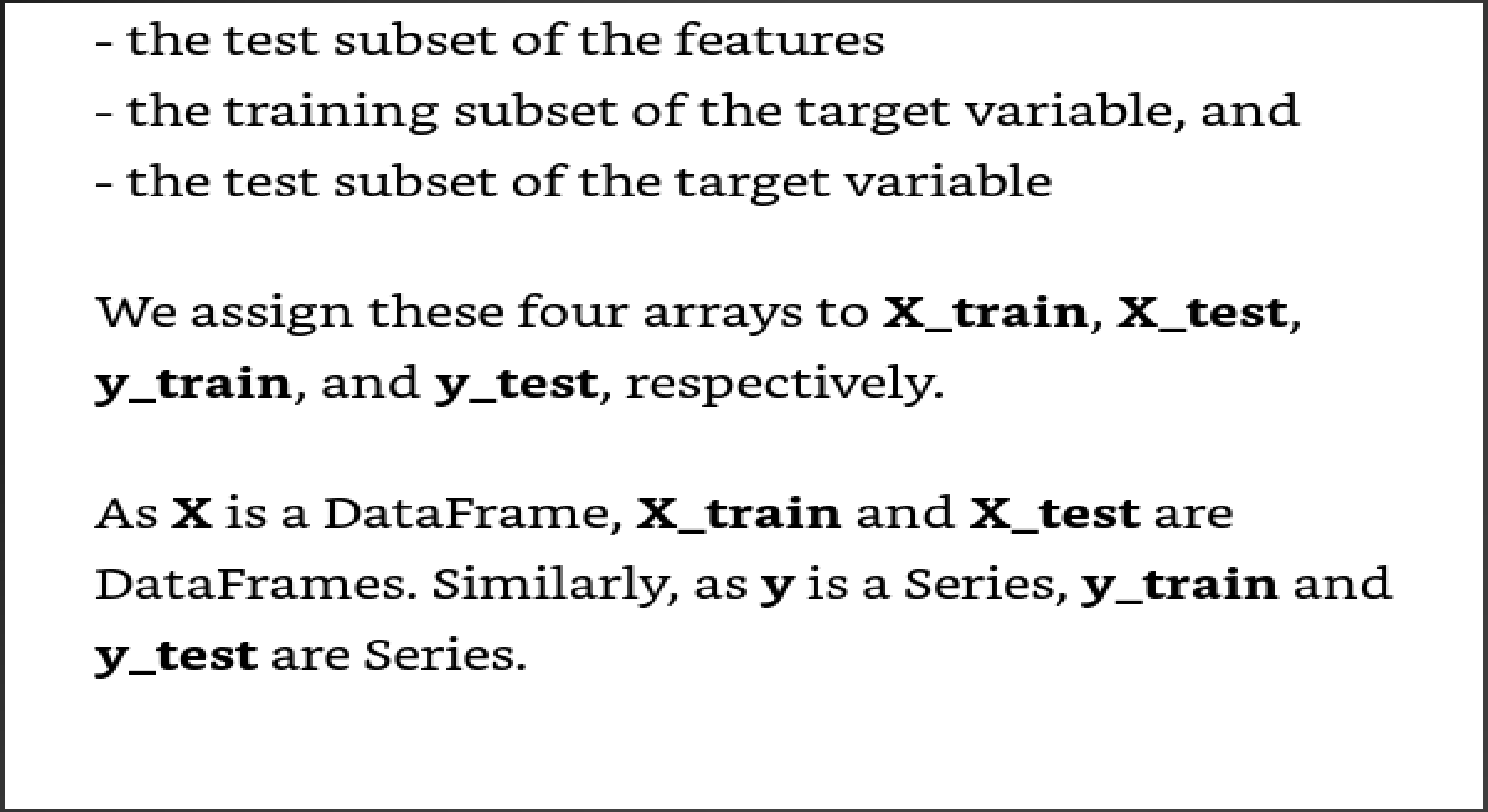

Here, we first select the feature and target variable in our dataset and assign them to **X** and **y**, respectively. Most sklearn methods expect features to be passed as a 2D array. Therefore, we select the feature as a DataFrame using two sets of square brackets.

Next, we pass **X** and **y** to the **train_test_split()** function. To specify the amount of data to use for testing, we use the **test_size** parameter. For this example, we specify **test_size** as 0.2, which means we want to use 20% of our data for the test set.


Finally, we specify the **random_state** parameter. This parameter controls the shuffling applied to the data before the split. The number 42 has no special meaning; you can choose another integer if you desire. However, if you want to produce the same output shown in this chapter, you need to use the same number 42. Else, your data will be shuffled differently, resulting in a different split and different outputs.

The **train_test_split()** function returns four arrays in the following order:

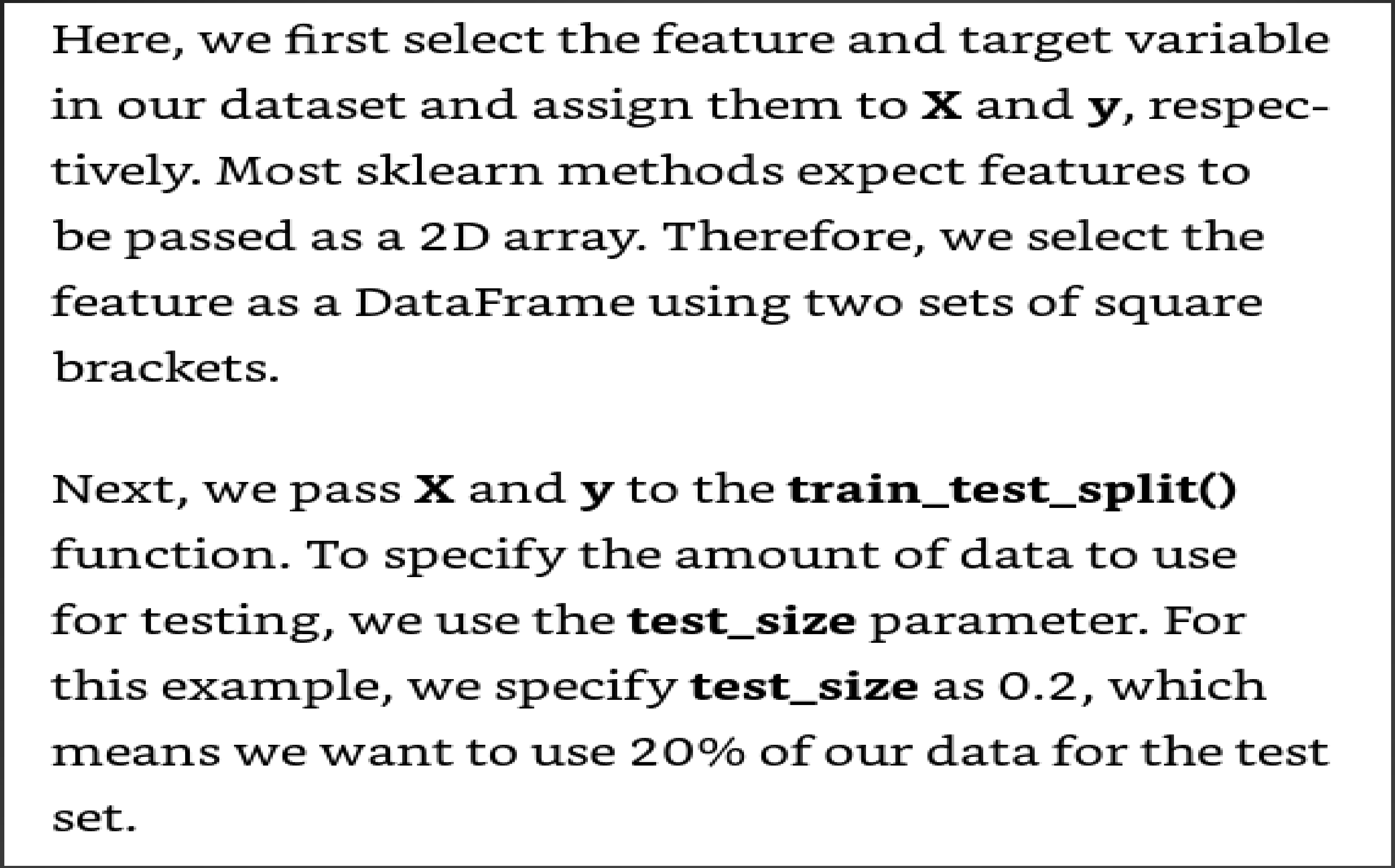
- the training subset of the features



- the test subset of the features
- the training subset of the target variable, and
- the test subset of the target variable

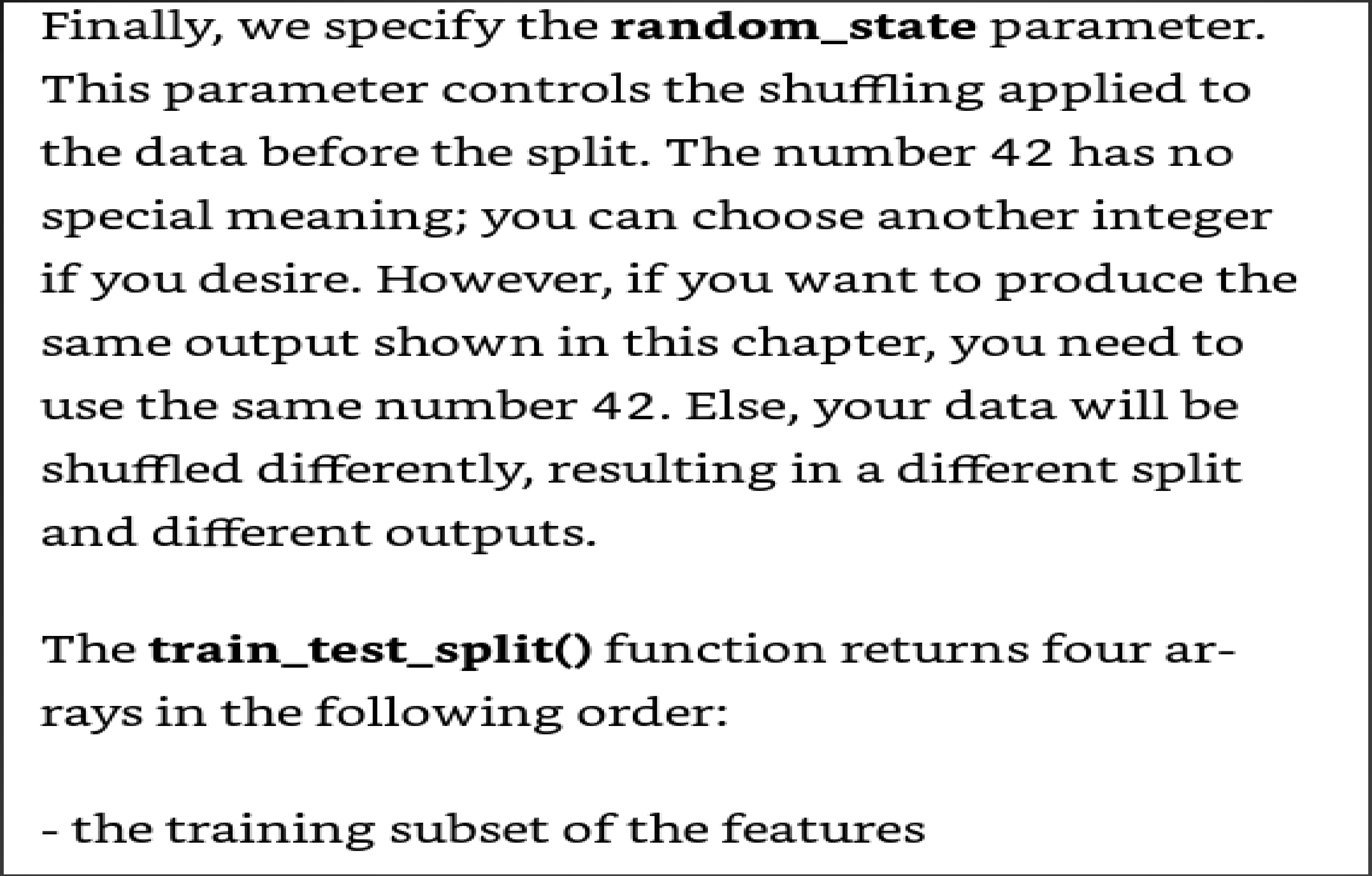
We assign these four arrays to **X_train**, **X_test**, **y_train**, and **y_test**, respectively.

As **X** is a DataFrame, **X_train** and **X_test** are DataFrames. Similarly, as **y** is a Series, **y_train** and **y_test** are Series.



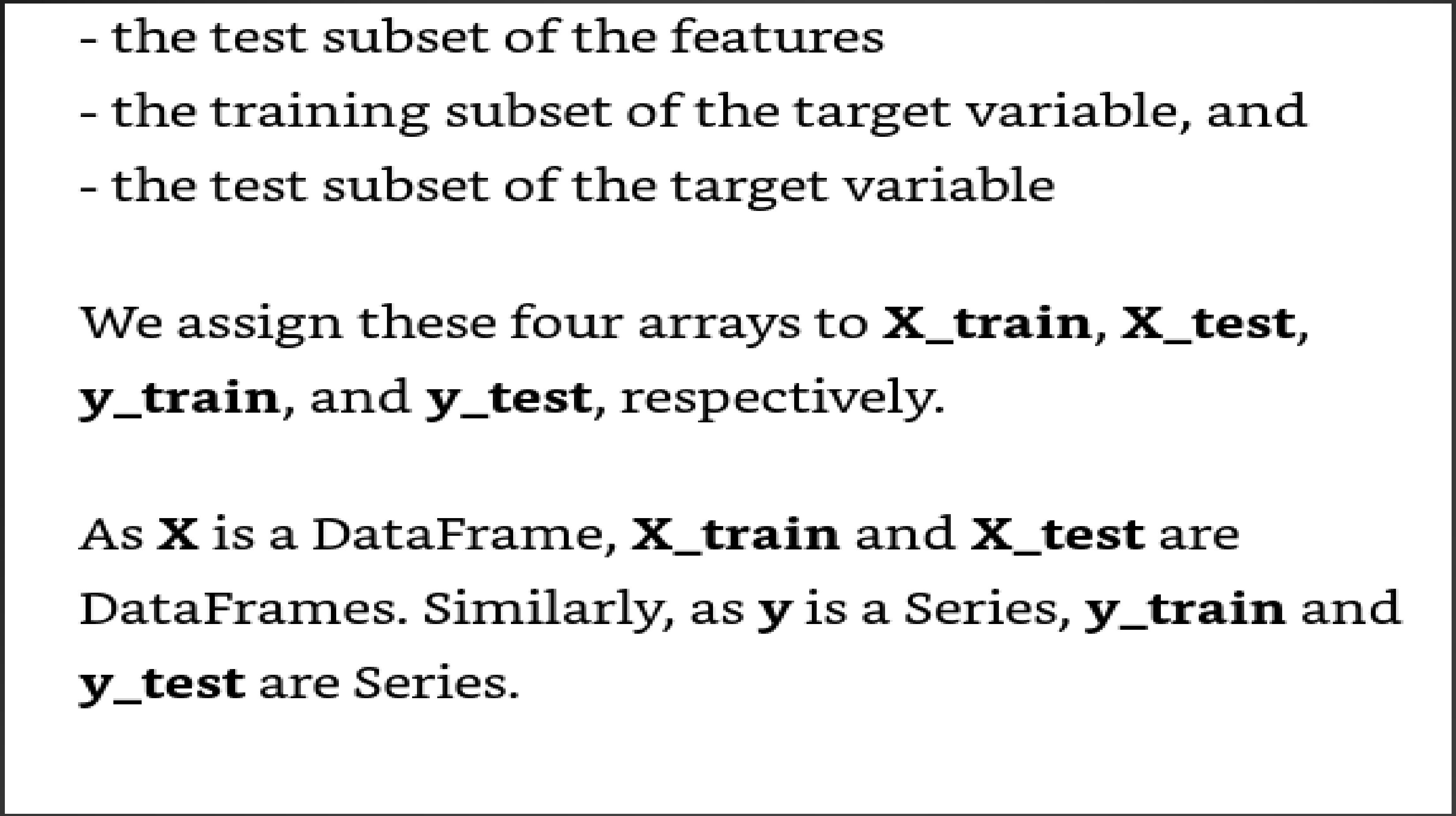
Here, we first select the feature and target variable in our dataset and assign them to **X** and **y**, respectively. Most sklearn methods expect features to be passed as a 2D array. Therefore, we select the feature as a DataFrame using two sets of square brackets.

Next, we pass **X** and **y** to the **train_test_split()** function. To specify the amount of data to use for testing, we use the **test_size** parameter. For this example, we specify **test_size** as 0.2, which means we want to use 20% of our data for the test set.


Finally, we specify the **random_state** parameter. This parameter controls the shuffling applied to the data before the split. The number 42 has no special meaning; you can choose another integer if you desire. However, if you want to produce the same output shown in this chapter, you need to use the same number 42. Else, your data will be shuffled differently, resulting in a different split and different outputs.

The **train_test_split()** function returns four arrays in the following order:

- the training subset of the features


- the test subset of the features
- the training subset of the target variable, and
- the test subset of the target variable

We assign these four arrays to **X_train**, **X_test**, **y_train**, and **y_test**, respectively.

As **X** is a DataFrame, **X_train** and **X_test** are DataFrames. Similarly, as **y** is a Series, **y_train** and **y_test** are Series.

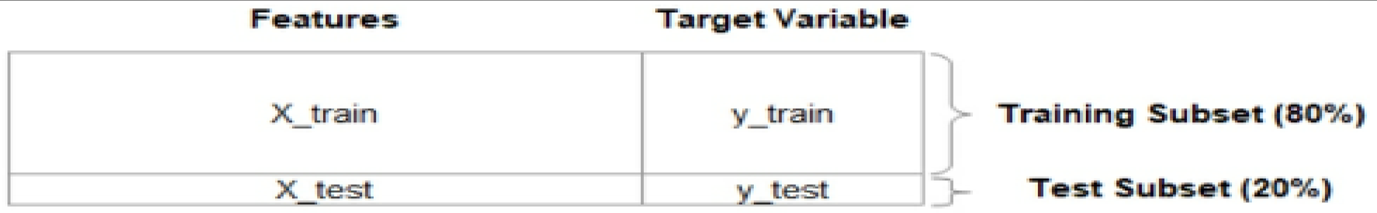


Figure 6.3: Training and test subsets

Step 4: Visualizing the Data

Now, let's do a simple scatter plot of the training subset to explore the relationship between housing price and floor area:

```
plt.scatter(X_train['Floor Area (sqft)'], y_train, s=10)
plt.title('Housing Price vs Floor Area')
plt.xlabel('Floor Area (sqft)')
plt.ylabel('Price in $1000s')
```

This gives us the following chart.

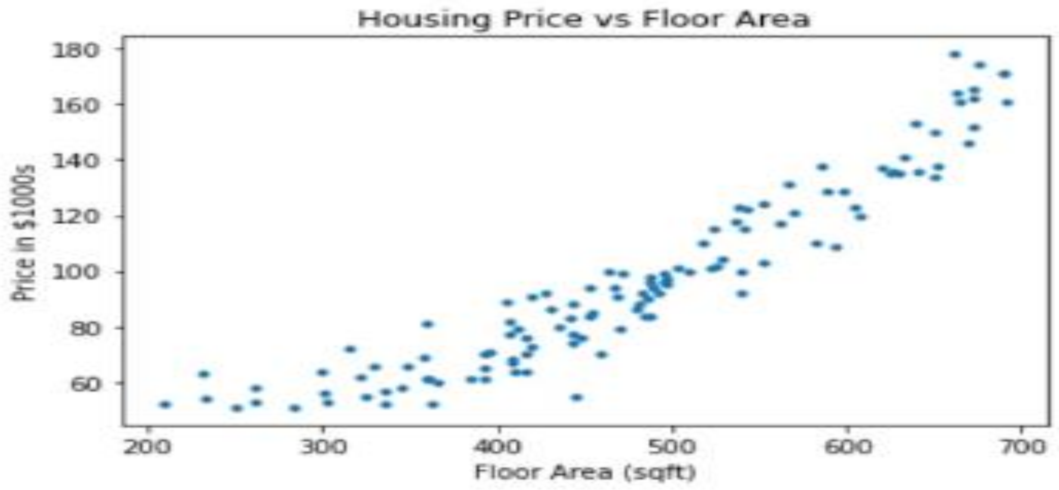
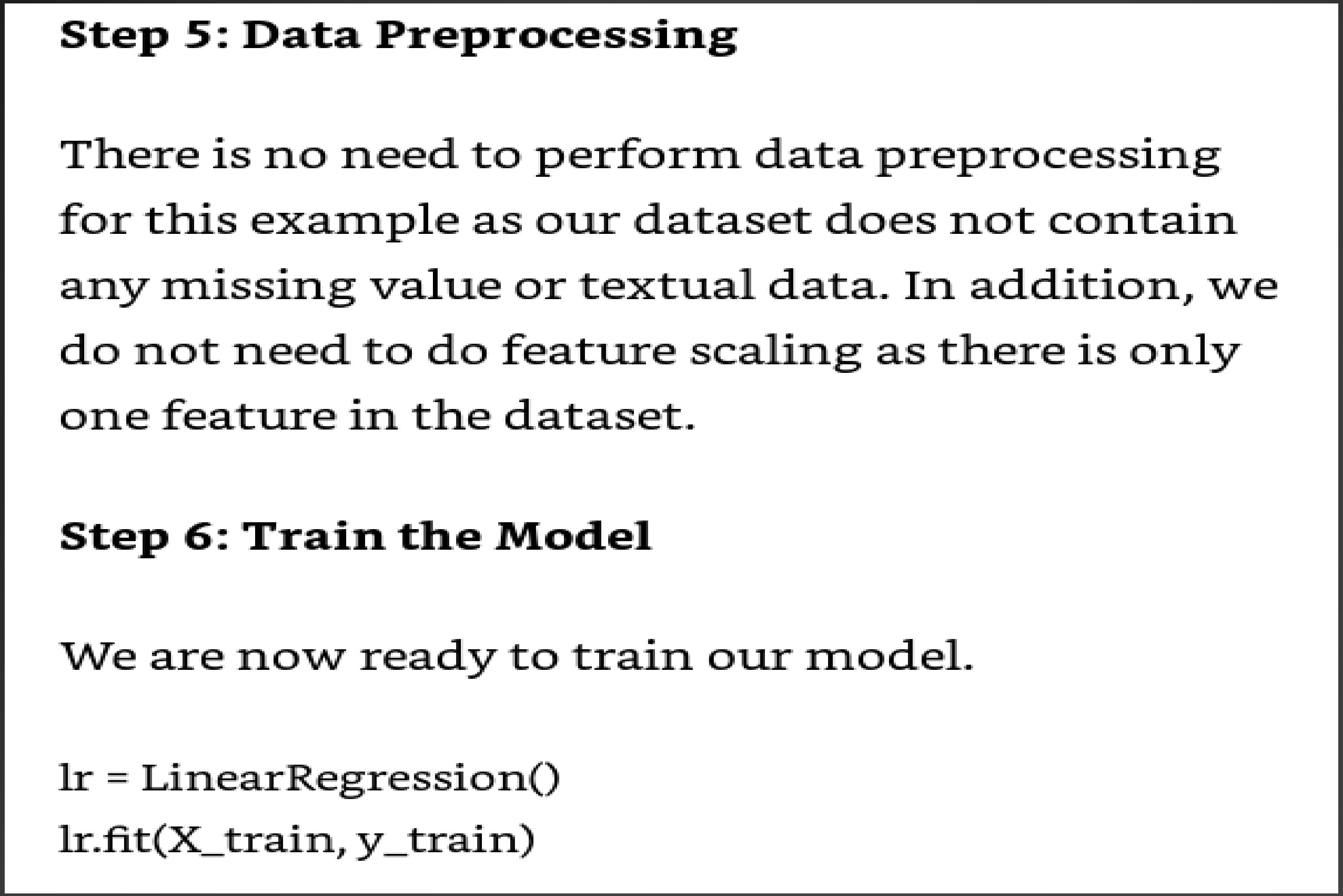


Figure 6.4: Scatter plot of the training set



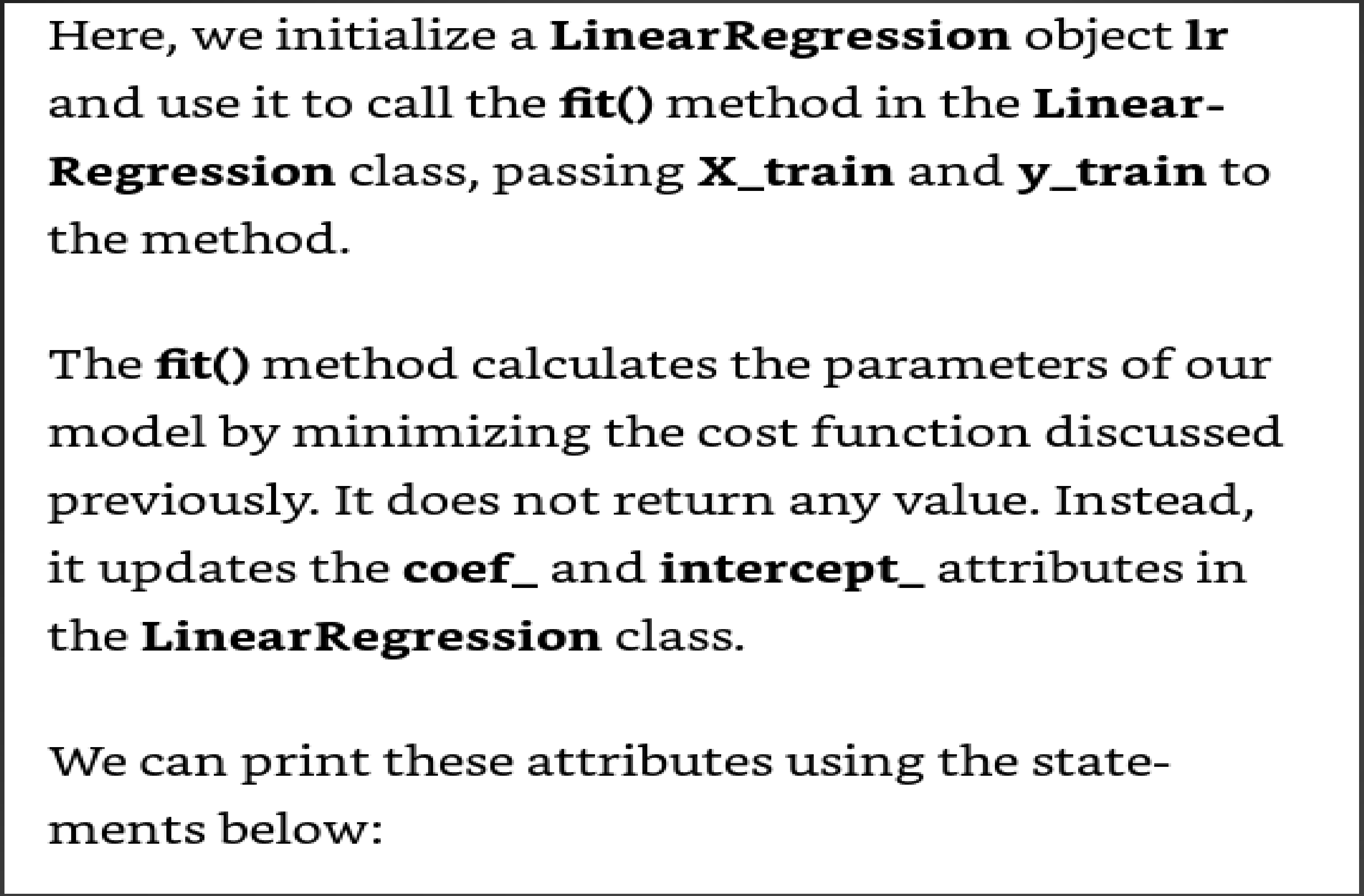
Step 5: Data Preprocessing

There is no need to perform data preprocessing for this example as our dataset does not contain any missing value or textual data. In addition, we do not need to do feature scaling as there is only one feature in the dataset.

Step 6: Train the Model

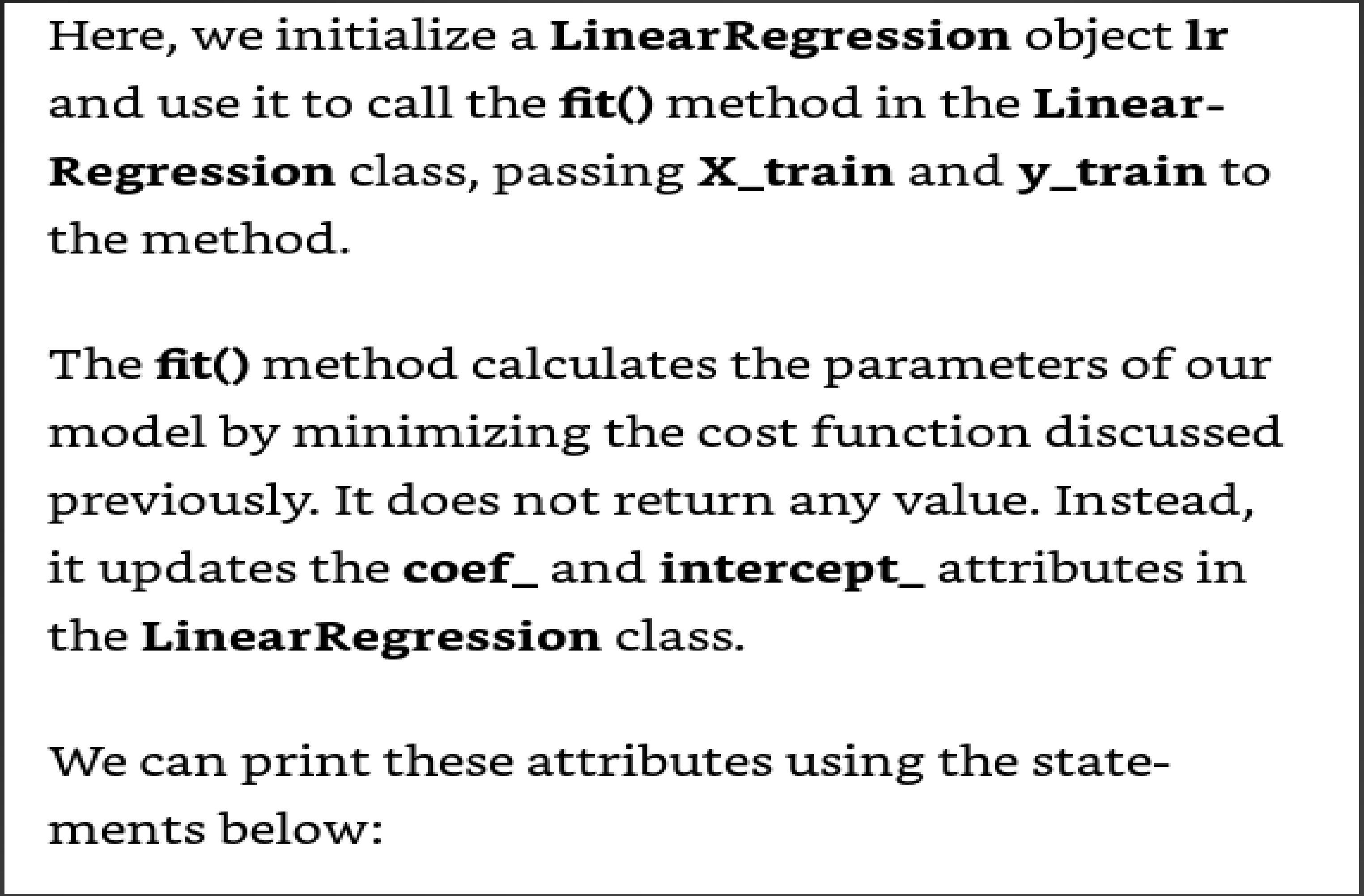
We are now ready to train our model.

```
lr = LinearRegression()
lr.fit(X_train, y_train)
```


Here, we initialize a **LinearRegression** object **lr** and use it to call the **fit()** method in the **Linear-Regression** class, passing **X_train** and **y_train** to the method.

The **fit()** method calculates the parameters of our model by minimizing the cost function discussed previously. It does not return any value. Instead, it updates the **coef_** and **intercept_** attributes in the **LinearRegression** class.

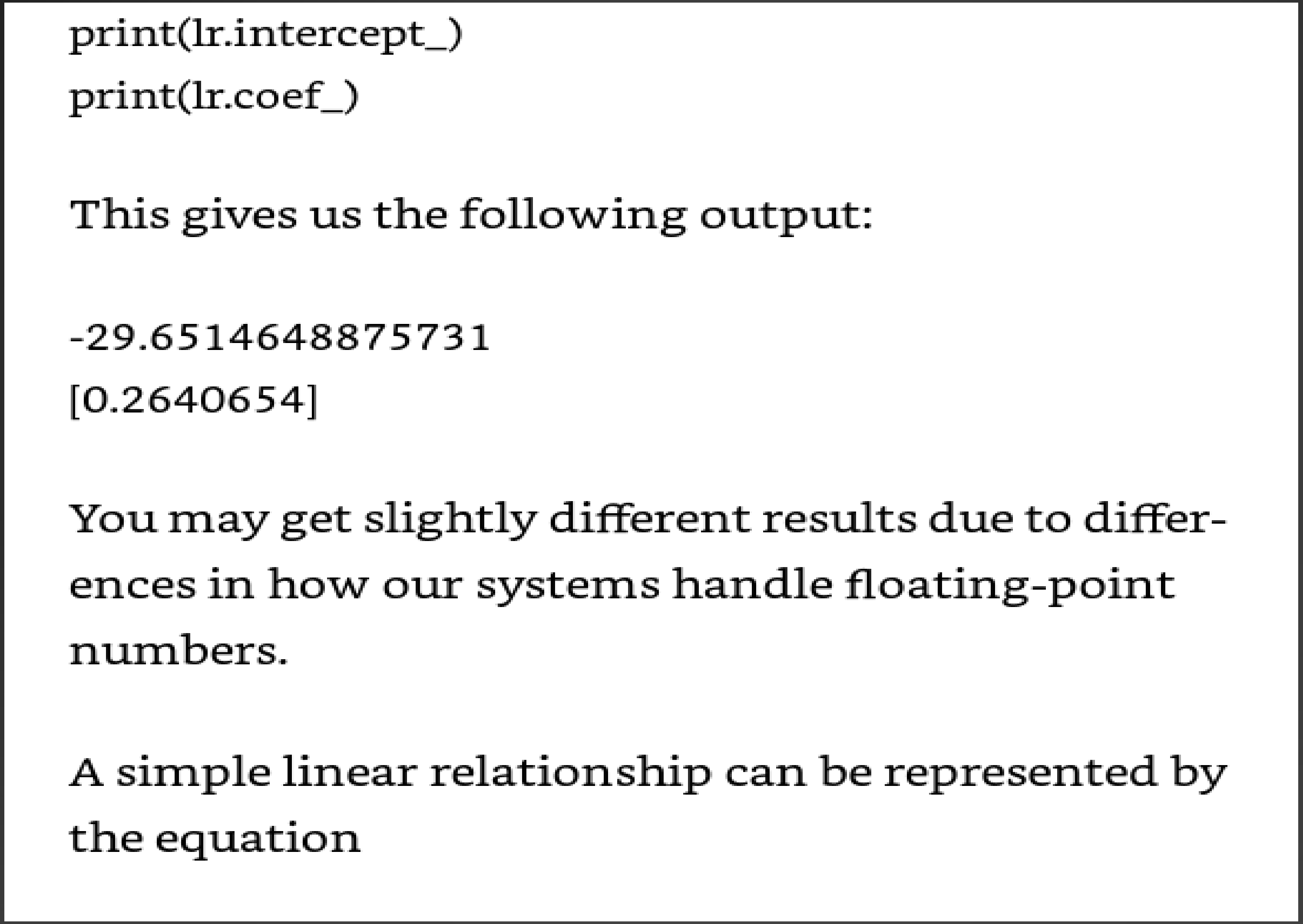
We can print these attributes using the statements below:



Here, we initialize a **LinearRegression** object **lr** and use it to call the **fit()** method in the **LinearRegression** class, passing **X_train** and **y_train** to the method.

The **fit()** method calculates the parameters of our model by minimizing the cost function discussed previously. It does not return any value. Instead, it updates the **coef_** and **intercept_** attributes in the **LinearRegression** class.

We can print these attributes using the statements below:



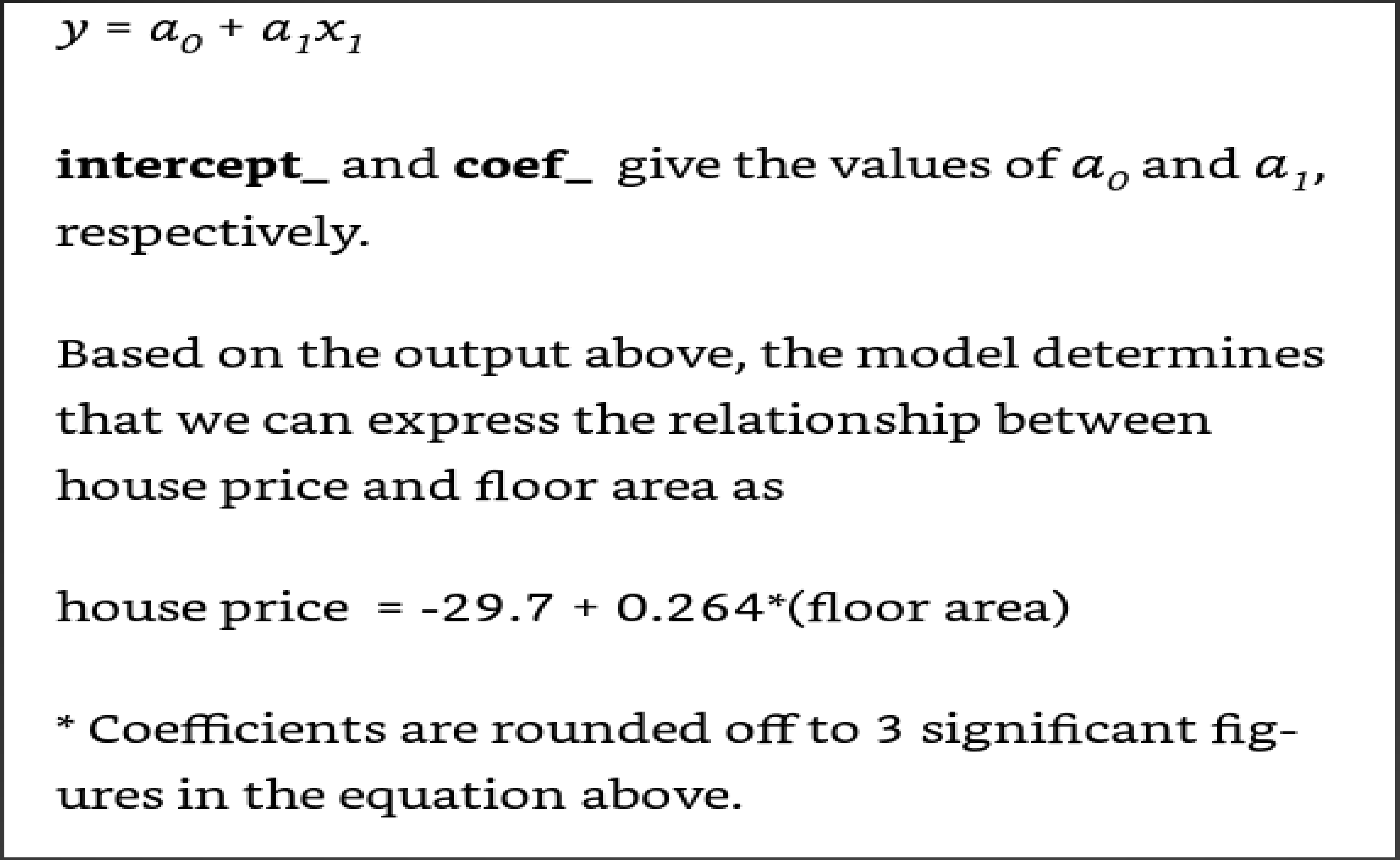
```
print(lr.intercept_)
print(lr.coef_)
```

This gives us the following output:

```
-29.6514648875731
[0.2640654]
```

You may get slightly different results due to differ-ences in how our systems handle floating-point numbers.

A simple linear relationship can be represented by the equation



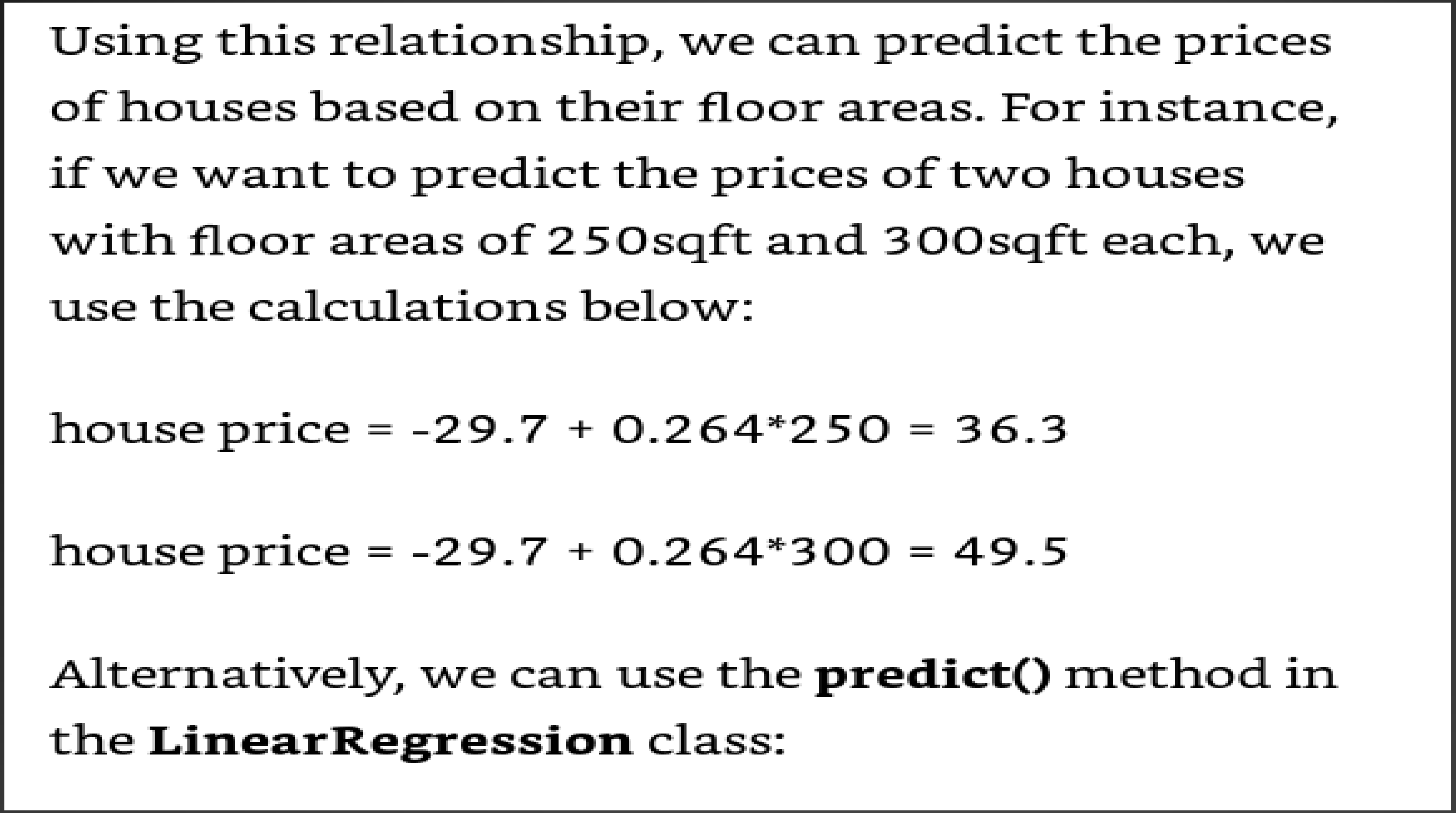
$y = a_0 + a_1x_1$

intercept_ and **coef_** give the values of a_0 and a_1, respectively.

Based on the output above, the model determines that we can express the relationship between house price and floor area as

house price = -29.7 + 0.264*(floor area)

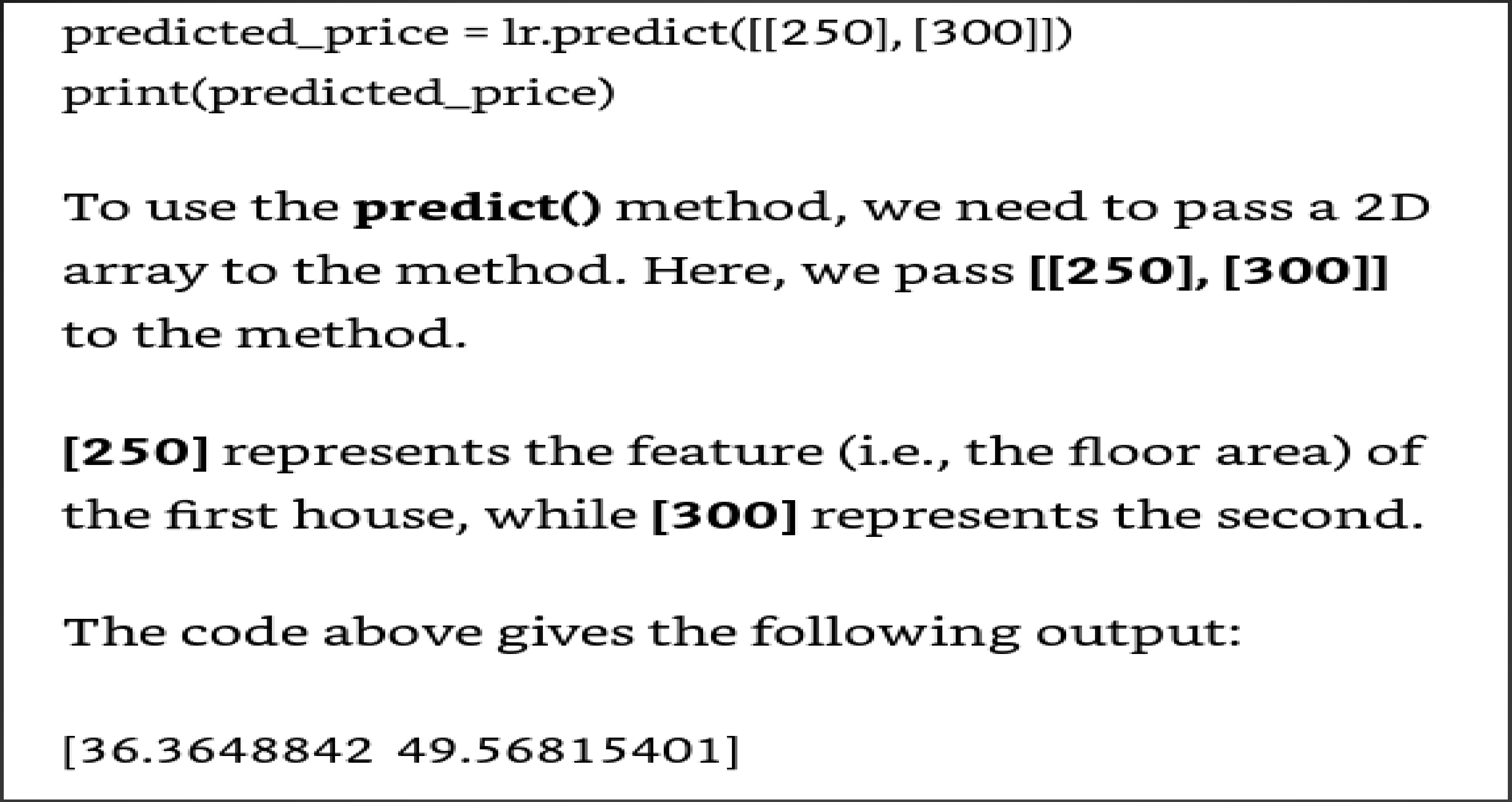
* Coefficients are rounded off to 3 significant figures in the equation above.


Using this relationship, we can predict the prices of houses based on their floor areas. For instance, if we want to predict the prices of two houses with floor areas of 250sqft and 300sqft each, we use the calculations below:

house price = -29.7 + 0.264*250 = 36.3

house price = -29.7 + 0.264*300 = 49.5

Alternatively, we can use the **predict()** method in the **LinearRegression** class:



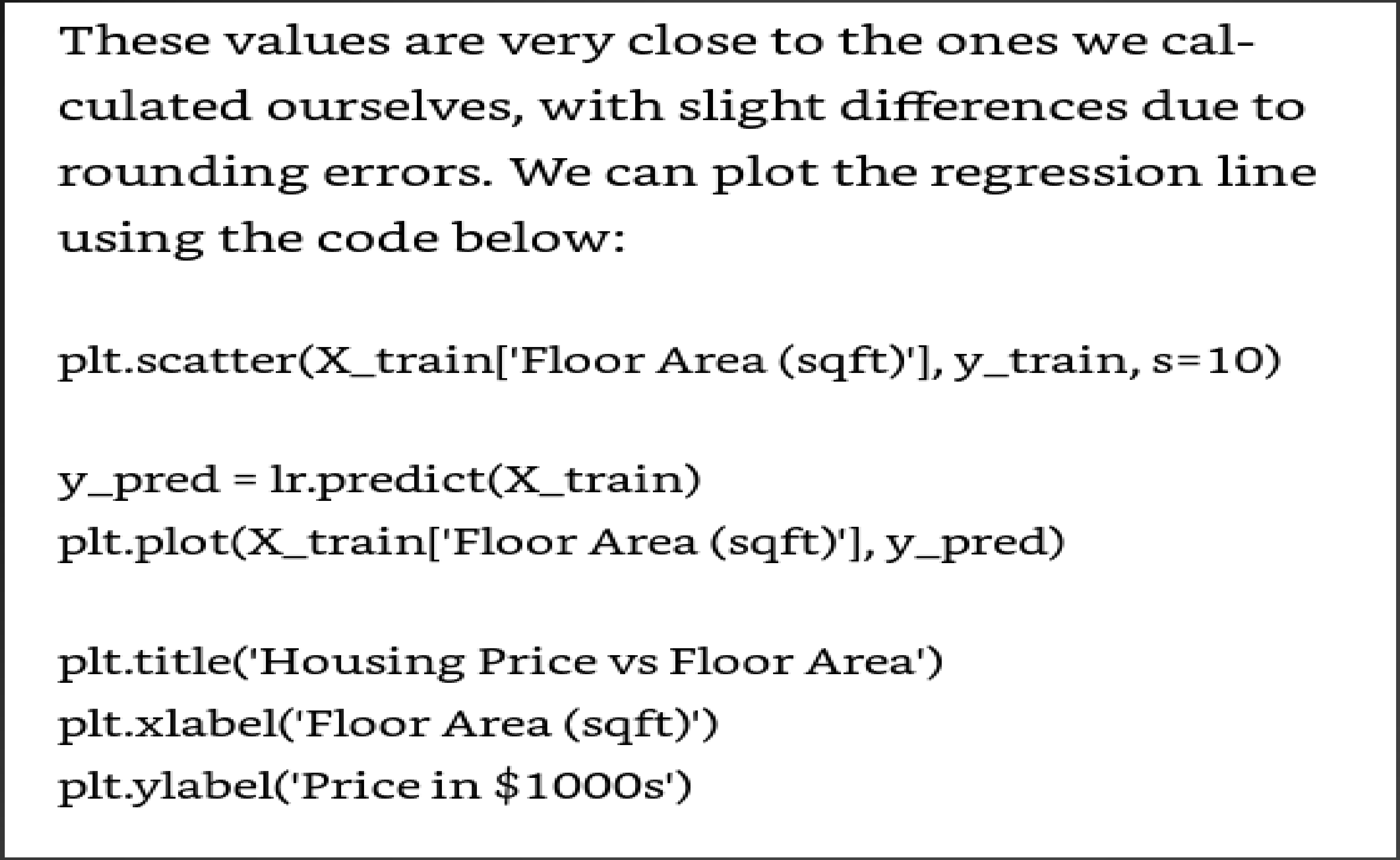
```
predicted_price = lr.predict([[250], [300]])
print(predicted_price)
```

To use the **predict()** method, we need to pass a 2D array to the method. Here, we pass **[[250], [300]]** to the method.

[250] represents the feature (i.e., the floor area) of the first house, while **[300]** represents the second.

The code above gives the following output:

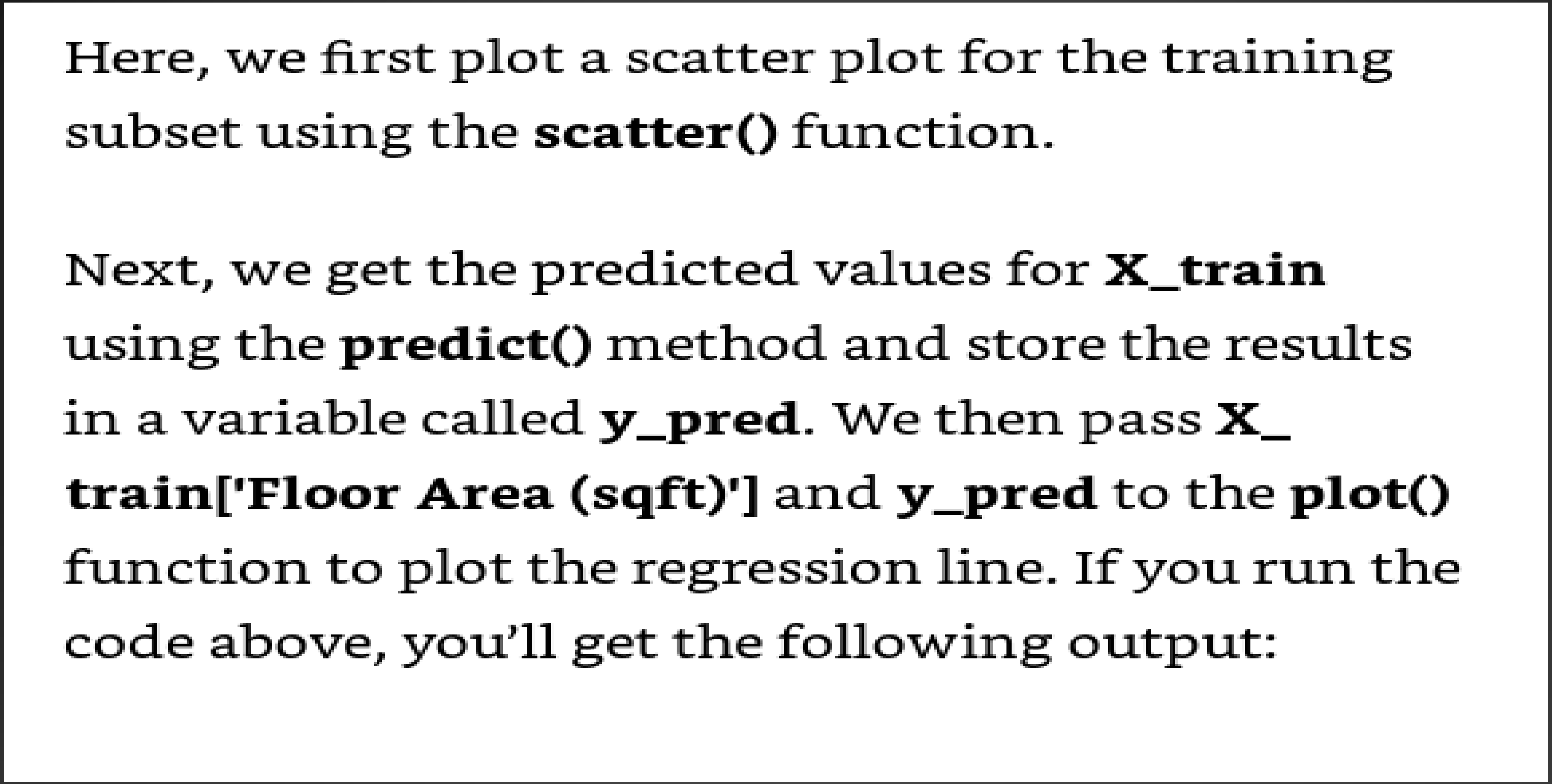
```
[36.3648842  49.56815401]
```


These values are very close to the ones we cal-
culated ourselves, with slight differences due to
rounding errors. We can plot the regression line
using the code below:

```
plt.scatter(X_train['Floor Area (sqft)'], y_train, s=10)

y_pred = lr.predict(X_train)
plt.plot(X_train['Floor Area (sqft)'], y_pred)

plt.title('Housing Price vs Floor Area')
plt.xlabel('Floor Area (sqft)')
plt.ylabel('Price in $1000s')
```


Here, we first plot a scatter plot for the training subset using the **scatter()** function.

Next, we get the predicted values for **X_train** using the **predict()** method and store the results in a variable called **y_pred**. We then pass **X_train['Floor Area (sqft)']** and **y_pred** to the **plot()** function to plot the regression line. If you run the code above, you'll get the following output:

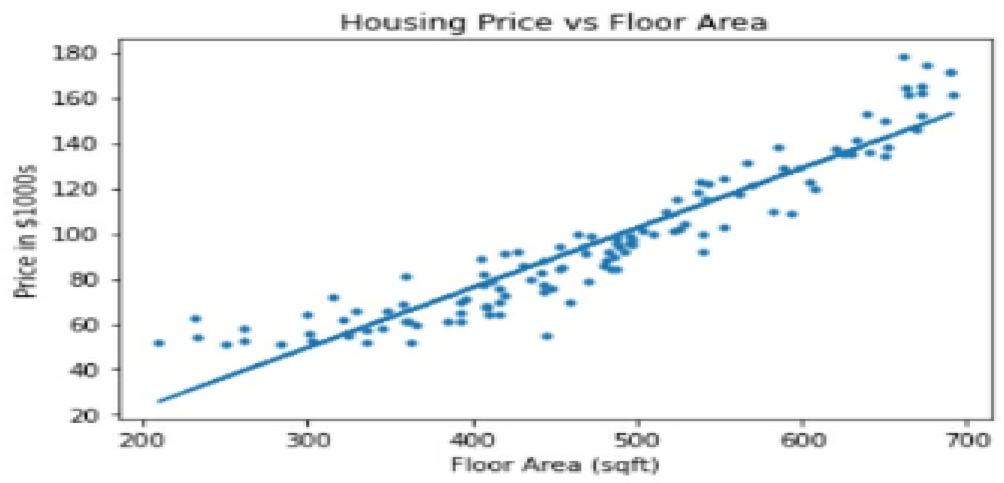
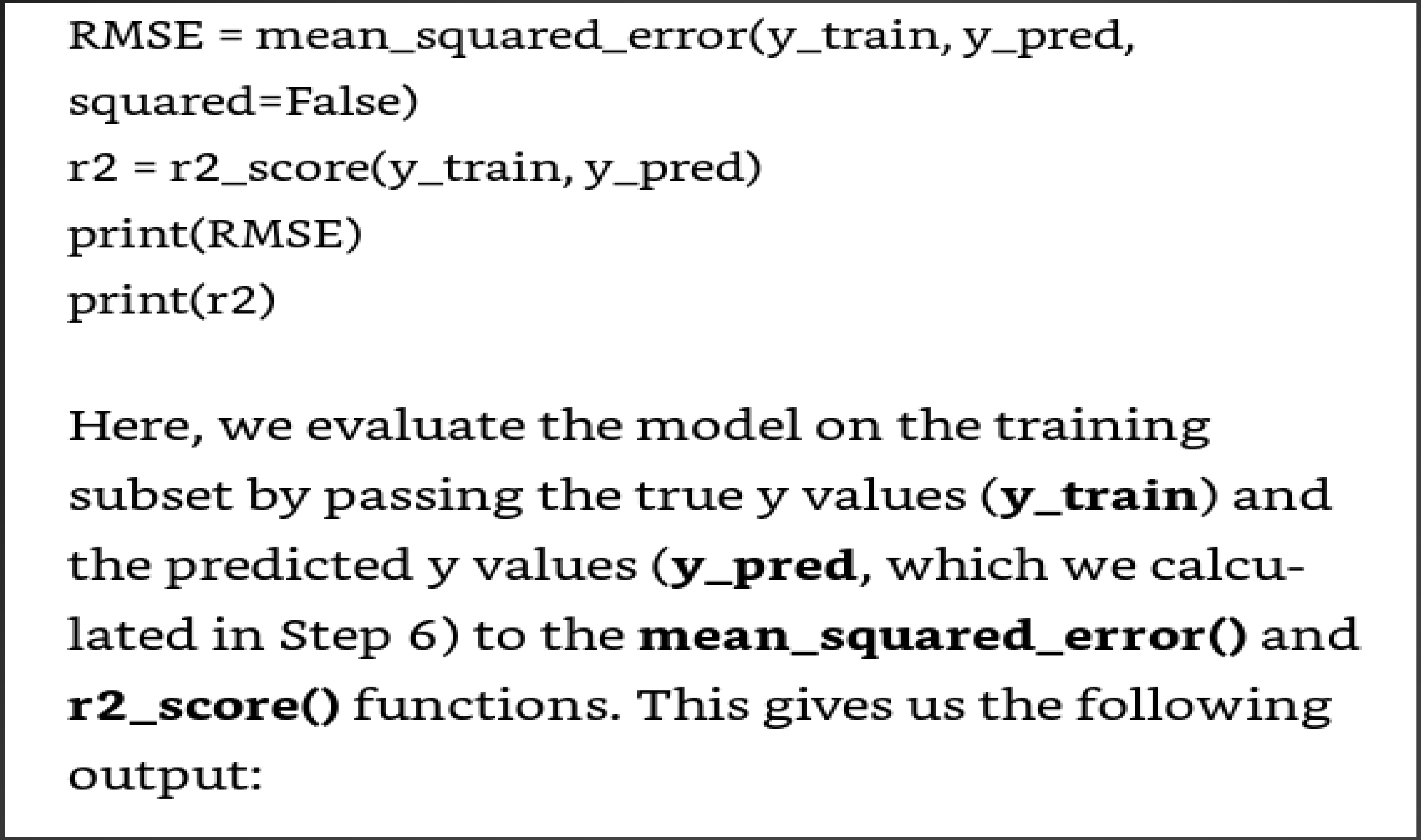


Figure 6.5: Plotting the regression line

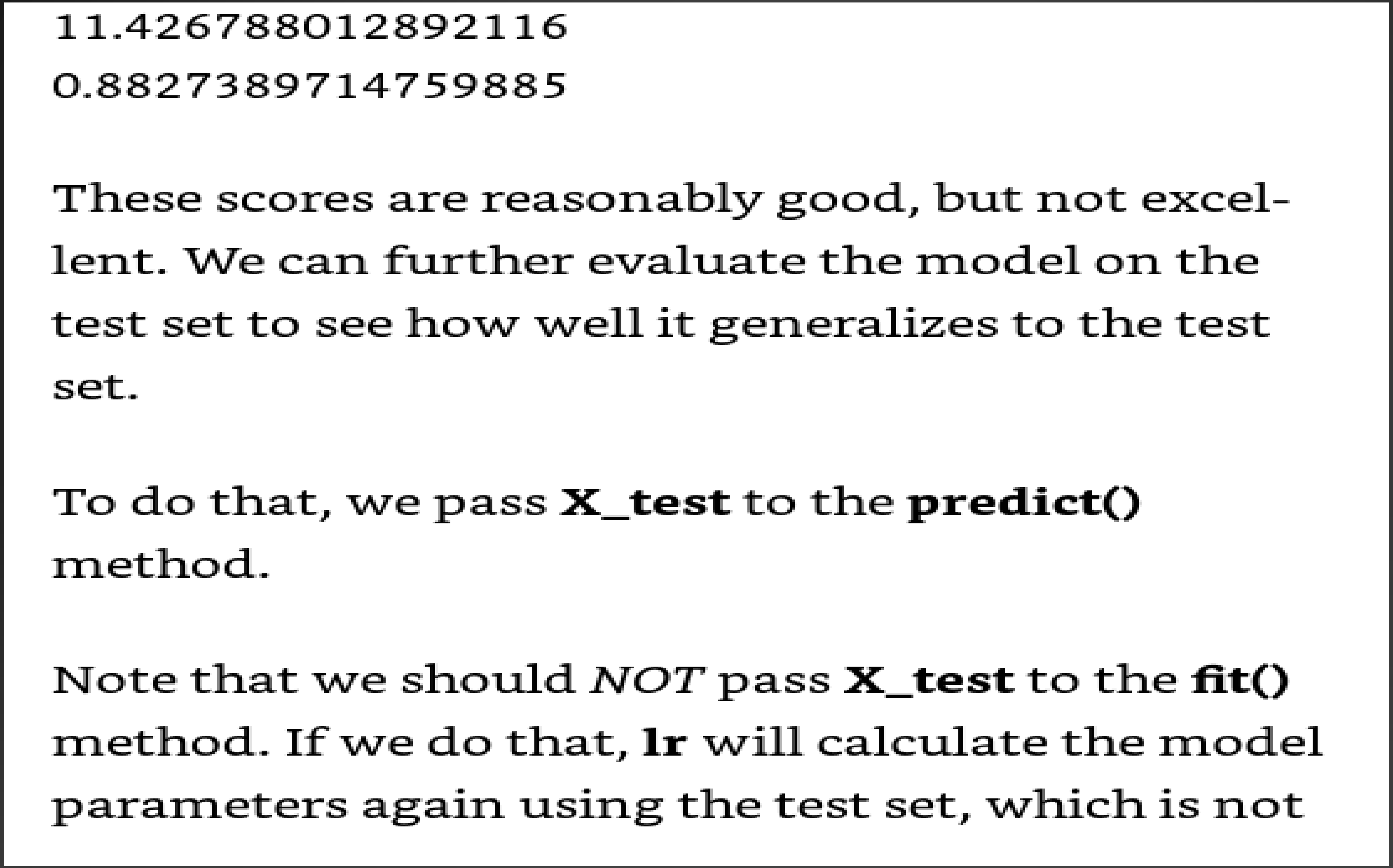
Step 7: Evaluate the Model

After we get the parameters of our model, we need to evaluate the model's performance:



```
RMSE = mean_squared_error(y_train, y_pred,
squared=False)
r2 = r2_score(y_train, y_pred)
print(RMSE)
print(r2)
```

Here, we evaluate the model on the training subset by passing the true y values (**y_train**) and the predicted y values (**y_pred**, which we calculated in Step 6) to the **mean_squared_error()** and **r2_score()** functions. This gives us the following output:

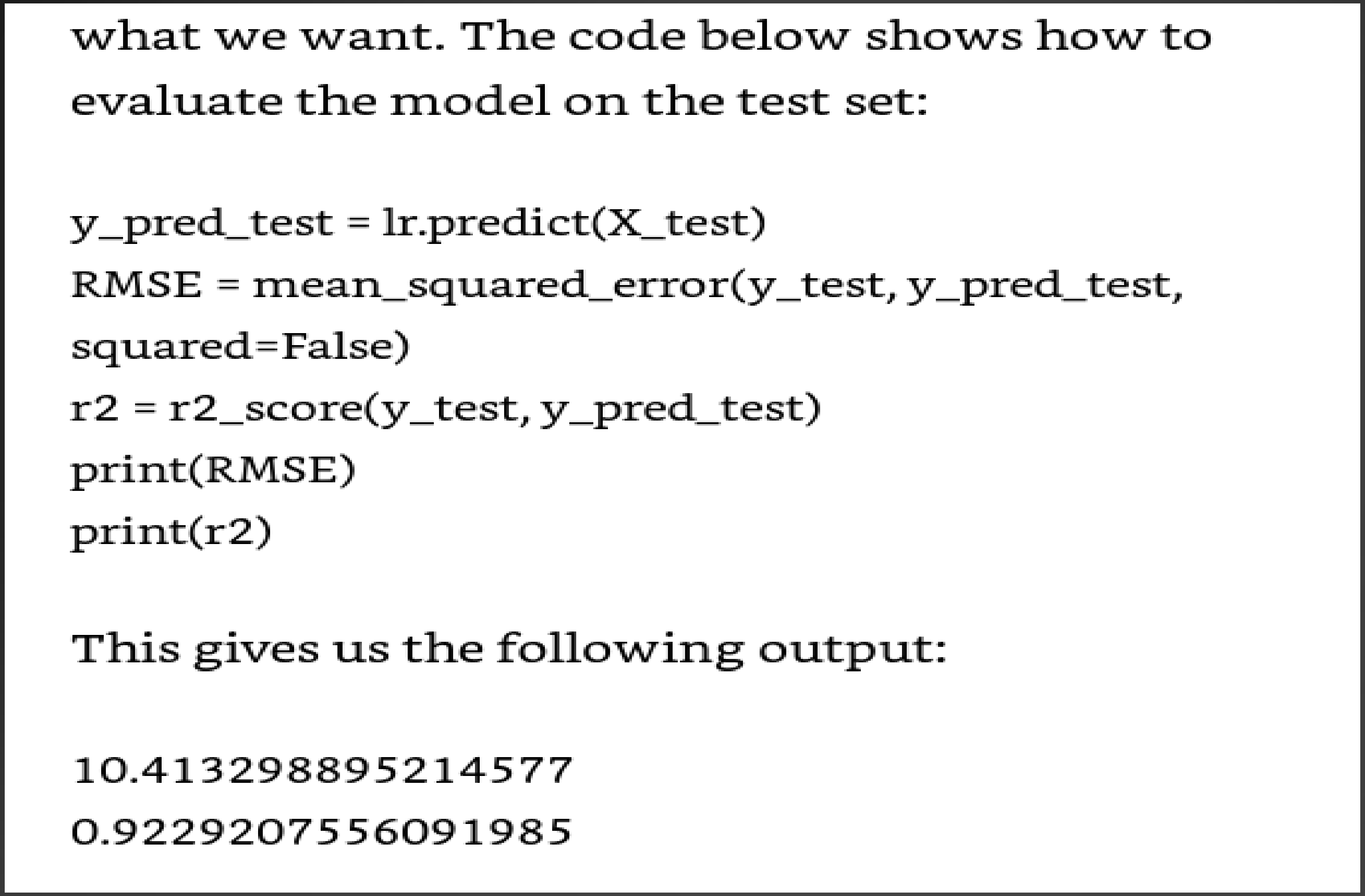


```
11.4267880012892116
0.8827389714759885
```

These scores are reasonably good, but not excel-
lent. We can further evaluate the model on the
test set to see how well it generalizes to the test
set.

To do that, we pass **X_test** to the **predict()**
method.

Note that we should *NOT* pass **X_test** to the **fit()**
method. If we do that, **lr** will calculate the model
parameters again using the test set, which is not

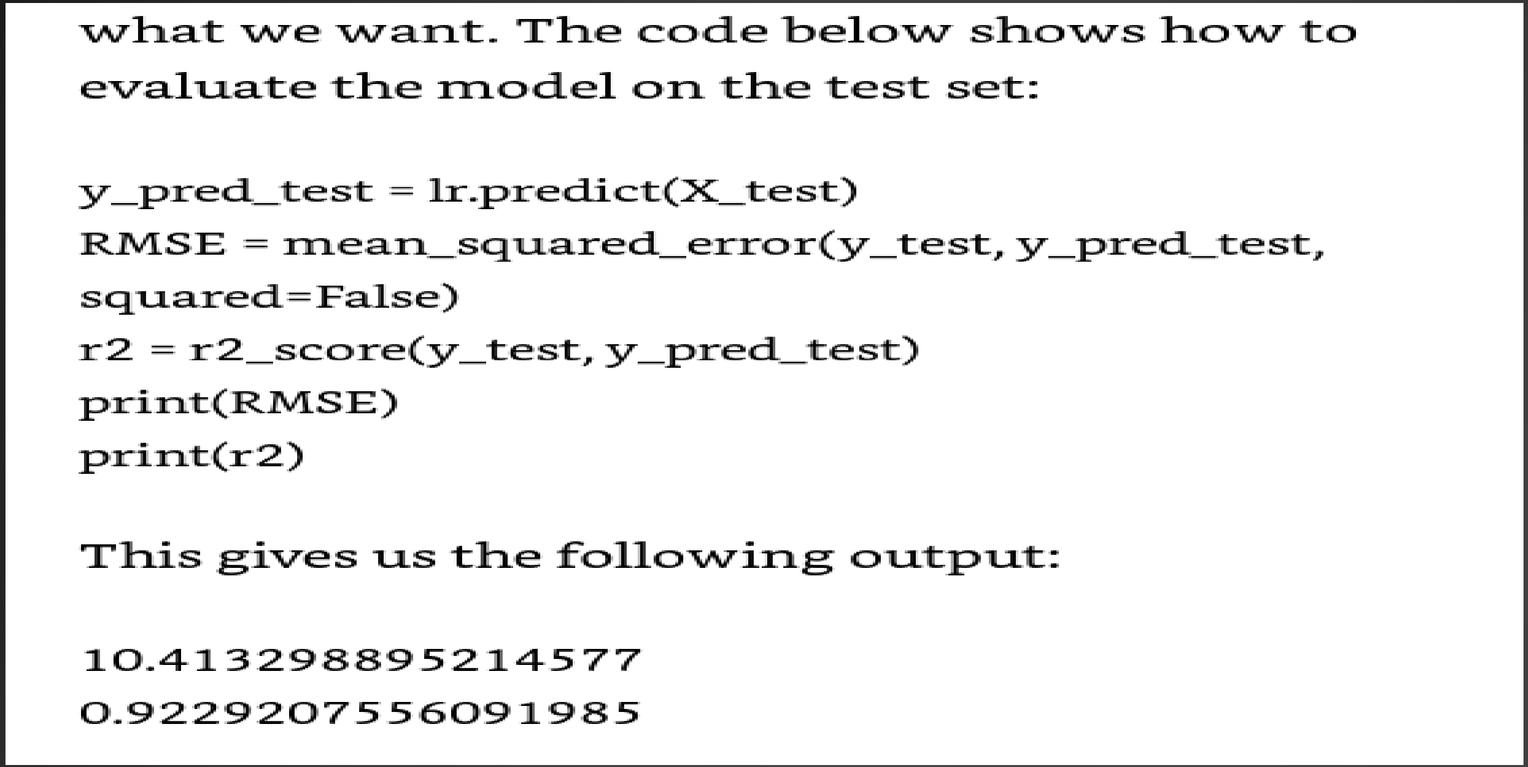


what we want. The code below shows how to evaluate the model on the test set:

```
y_pred_test = lr.predict(X_test)
RMSE = mean_squared_error(y_test, y_pred_test,
squared=False)
r2 = r2_score(y_test, y_pred_test)
print(RMSE)
print(r2)
```

This gives us the following output:

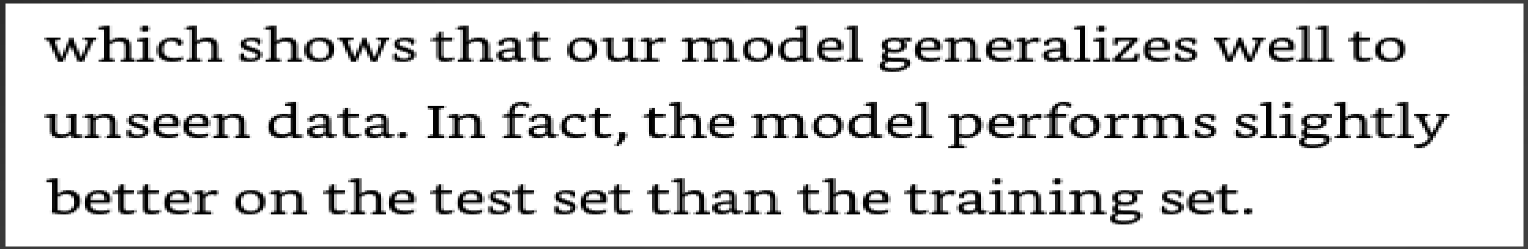
```
10.413298895214577
0.9229207556091985
```


what we want. The code below shows how to evaluate the model on the test set:

```
y_pred_test = lr.predict(X_test)
RMSE = mean_squared_error(y_test, y_pred_test,
squared=False)
r2 = r2_score(y_test, y_pred_test)
print(RMSE)
print(r2)
```

This gives us the following output:

```
10.413298895214577
0.9229207556091985
```


which shows that our model generalizes well to unseen data. In fact, the model performs slightly better on the test set than the training set.