2

FORECASTING

Let's look at some data science tools that can help you predict the future. In this chapter, we'll introduce a simple business scenario in which a company needs to forecast customer demand. We'll then talk about how tools from data science can be applied to make an accurate forecast and how that forecast can lead to better business decision-making.

We'll use linear regression for our forecasting, and we'll discuss both univariate and multivariate linear regression. Finally, we'll look at extrapolation of regression lines and how to evaluate various regression models to choose the best one.

Predicting Customer Demand

Imagine that you're running a car dealership in Quebec, Canada. You are using a standard business model for retail: you buy cars from a manufacturer at a low price and then sell those cars to individual customers at higher prices. Every month, you need to decide how many cars you'll order from the manufacturer. If you order too many cars, you'll be unable to sell them all quickly, resulting in high storage costs or cash flow problems. If you order too few cars, you won't be able to meet your customers' demands.

Ordering the right number of cars is important. But what is the right number? The answer depends on certain business considerations, such as the cash in your bank account and how much you want to grow—but in a typical month, the right number of cars to order is exactly the number of cars that customers will want to buy during the coming month. Since we can't see into the future, we need to forecast the demand and place an order based on our forecast.

We can choose from several proven quantitative methods to obtain a forecast of next month's demand. One of the best methods is *linear regression*. In the remainder of this chapter, we'll explain how to use linear regression for forecasting. We'll use past data to predict future data, to learn the number of cars we need to order. We'll start simply, just by reading in and looking at some data, and then proceed to the other steps of the forecasting process.

Cleaning Erroneous Data

The data we'll analyze to forecast the future is a record of the number of cars sold by dealerships in Quebec, Canada, for each of 108 consecutive months. This data was originally made available online by Rob Hyndman, a statistics professor and forecasting guru. You can download the data from *https://bradfordtuckfield.com/carsales.csv*.

This data is old; the most recent month recorded is December 1968. Therefore, for this scenario, we'll be imagining that we live in December 1968, and we'll make forecasts for January 1969. The forecasting principles we'll discuss will be evergreen, so if you can use data from 1968 to forecast results in 1969, you'll be able to use data from year n to forecast results from year $n + 1$, for $n = 2,023$ or $3,023$ or any other number.

Save this file into the same directory where you're running Python. Then we'll read our data by using Python's pandas package:

```
import pandas as pd
carsales=pd.read_csv('carsales.csv')
```

Here, we import pandas and give it the alias `pd`. We then use its `read_csv()` method to read our data into Python and store it in the variable `carsales`. The pandas package we import and use here is a powerful module that makes working with data in Python easier. The `carsales` object we create is a pandas dataframe, which is the standard pandas format for storing data in a Python session. Because the object is stored as a pandas

dataframe, we'll be able to use many helpful pandas methods to work with it, just as we did in Chapter 1. Let's start by using the head() method that enables us to inspect pandas dataframes:

```
>>> print(carsales.head())
     Month  Monthly car sales in Quebec 1960-1968
0  1960-01                                 6550.0
1  1960-02                                 8728.0
2  1960-03                                12026.0
3  1960-04                                14395.0
4  1960-05                                14587.0
```

By looking at these rows, we can notice a few important points. First, we can see the column names. The column names in this dataset are Month and Monthly car sales in Quebec 1960-1968. The second column name will be easier to work with if we shorten it. We can do this easily in Python:

```
carsales.columns= ['month','sales']
```

In this snippet, we access the columns of our dataframe and redefine them to have shorter names (month and sales, respectively).

Just as the head() method prints the top five rows of a dataset, the tail() method prints the bottom five rows. If you run print(carsales.tail()), you'll see the following output:

```
>>> print(carsales.tail())
                                  month    sales
104                             1968-09  14385.0
105                             1968-10  21342.0
106                             1968-11  17180.0
107                             1968-12  14577.0
108  Monthly car sales in Quebec 1960-1968     NaN
```

We can see that the column names are shorter now and easier to read. But we also see that the very last row doesn't contain car sales data. Instead, its first entry is a *tag*, or label, that tells us about the whole dataset. Its second entry is NaN, which stands for *not a number*, meaning that the entry contains no data or undefined data. We don't need the label entry or the empty (NaN) entry, so let's remove the entire last row (row 108):

```
carsales=carsales.loc[0:107,:].copy()
```

Here, we use the pandas loc() method to specify a selection of rows that we want to keep: in this case, all the rows between row 0 and row 107, inclusive. We use the colon (:) after the comma to indicate that we want to keep both of the dataset's columns. We store the result in our carsales variable, thereby removing the superfluous row 108. If you run **print(carsales.head())** again, you'll see that that row has been removed.

Another thing we can see by looking at the head and tail of our data is the format of the month data. The first entry is 1960-01 (January of 1960), the second column is 1960-02 (February of 1960), and so on.

As data scientists, we're interested in doing numeric analyses using math, statistics, and other quantitative methods. Dates can present several tedious challenges that make it hard to do math and statistics the way we want to. The first challenge is that dates are sometimes not stored in a numeric data type. Here, the dates are stored as *strings*, or collections of characters.

To see why this is an issue, try `print(1960+1)` in the Python console; you'll notice that the result is `1961`. Python has seen that we're working with two numbers, and it's added them in the way we expect. Then, try `print('1960'+'1')` in the Python console; now you get `19601` as the result. Instead of adding two numbers, Python has seen that we've input strings and assumes that the + sign means that we want to do *concatenation*, simply fusing the strings together in a way that doesn't follow the rules of math.

Another challenge with dates is that even when they're in numeric form, they follow logic that is different from the logic of natural numbers. For example, if we add 1 to month 11, we get month 12, which follows the arithmetic rule that 11 + 1 = 12. But if we add 1 to month 12, we get month 1 again (since every December is followed by January of the next year), which is not consistent with the simple arithmetic of 12 + 1 = 13.

In this case, the simplest way to address the issues with the data type of our date data is to define a new variable called `period`. We can define it as follows:

```
carsales['period']=list(range(108))
```

Our new `period` variable is just a list of all the numbers from 0 to 107. We'll refer to January 1960 as period 0, February 1960 as period 1, and so on until December 1968, the last month in our data, which we'll call period 107. This new variable is numeric, so we can add to it, subtract from it, or do any other mathematical operation with it. Also, it will follow the rules of standard arithmetic, with period 13 coming after period 12, as we expect in numeric variables. This simple solution is possible because in this particular dataset, the rows are organized in chronological order, so we can be sure that each period number is being assigned to the correct month.

These simple tasks, like adding a numeric column for months, removing an extra row, and changing column names, are part of *data cleaning*. This is not a glamorous or particularly exciting process, but doing it right is extremely important because it lays a foundation for the more thrilling steps of the data science process.

Plotting Data to Find Trends

After these basic data-cleaning tasks, we should definitely plot the data. Plotting should be done early and often in every data science project. Let's use the Matplotlib module to create a simple plot of our data:

```
from matplotlib import pyplot as plt
plt.scatter(carsales['period'],carsales['sales'])
```

```
plt.title('Car Sales by Month')
plt.xlabel('Month')
plt.ylabel('Sales')
plt.show()
```

In this snippet, we import the Matplotlib `pyplot` module and give it the alias `plt`. Then, we use the `scatter()` method to create a scatterplot of all the sales numbers in our data, organized by period (month). We also use a few lines to add axis labels and a plot title and then show the plot. Figure 2-1 shows the result.
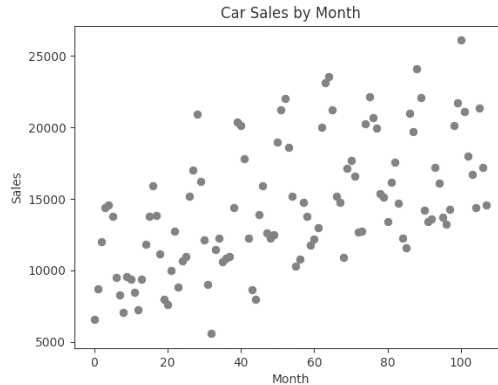

Figure 2-1: Car sales by month over nine years

This simple plot shows our `period` variable on the x-axis and `sales` on the y-axis. Each point represents one row of data or, in other words, the number of car sales for one particular month.

See what interesting information pops out at you in this plot. Probably the most obvious thing is the gradual upward trend from left to right: sales appear to be increasing gradually over time. Other than this trend, the data seems noisy and scattered, with huge variations between one month and another. The variations within a year or season look random, noisy, and unpredictable. The linear regression method that we'll implement next will attempt to capture the order and patterns in our data, and help us be less distracted by the randomness and noise.

So far, all we've done is read in the data and draw a simple plot. But already we're starting to see patterns that will be useful for making accurate forecasts. Let's go on to some more serious forecasting steps.

Performing Linear Regression

Now that we've cleaned the data, plotted it, and noticed some basic patterns, we're ready to do forecasting in earnest. We'll use linear regression for our forecasts. *Linear regression* is an essential part of every data scientist's toolkit: it finds a line that captures a noisy relationship between variables, and we can use that line to make predictions about things we've never seen.

Linear regression was invented more than a century before the term *machine learning* was coined, and it has historically been thought of as part of pure statistics. However, since it bears such a strong resemblance to many common machine learning methods, and since it shares some common theoretical foundations with machine learning, linear regression is sometimes considered part of the machine learning field. Like all the best scientific tools, it allows us to pull order from chaos.

In this case, we have the chaos of car sales in Quebec, where seasonal variations, time trends, and plain randomness mingle together in a noisy dataset. When we apply simple linear regression to this data, our output will be a straight line that captures an underlying structure that will help us make sound forecasts for the future. Figure 2-2 shows an example of a typical output of linear regression.
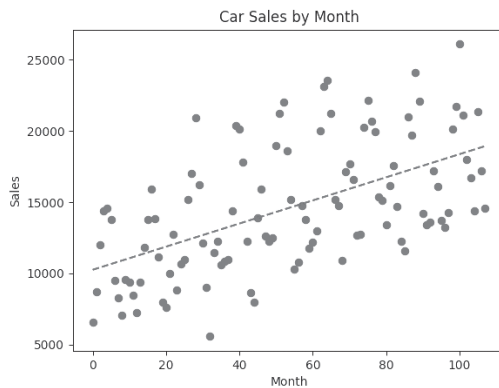

Figure 2-2: A dashed line showing a typical output of linear regression

In this plot, you can see the points representing the data, just as in Figure 2-1. Again, we see the chaos of our dataset: great variations occur between months across the whole dataset.

The dashed line that progresses slightly upward from left to right represents the output of a linear regression. It's called a *regression line*, and we often say that this regression line *fits* the data. In other words, it goes through what looks like roughly the center of the cloud constituted by all the points together. It gets close to many of the points of our data, and no data point is particularly far away from it. It's as if the line expresses or reveals the fundamental relationship between time and sales (a relationship of gradual growth). The notion of a line fitting a set of points is fundamental to linear regression. In fact, for reasons we'll discuss later, a regression line is sometimes called the *line of best fit* to a dataset.

Since our regression line is a straight line, it doesn't have the random variation of the real data. The line proceeds in a predictable way. By removing that randomness, the regression line shows us a clear representation of the underlying pattern of the data. In this case, the regression line shows us that the data has a general trend upward over time, and if we measure the regression line carefully, we can find exactly the slope and height of that trend.

We can interpret the value of the regression line for any particular month as the number of car sales expected in that month. Later we'll extrapolate our simple line forward into the future (by continuing to draw it with the same slope until it extends past the right edge of the plot) to generate forecasts for sales in future months.

Let's run the code that performs linear regression and outputs a regression line. We'll use methods for linear regression that are very particular about the *shape* of the data we use, meaning whether sales numbers are stored as 108 rows × 1 column or 108 columns × 1 row. In this case, our linear regression code will run more smoothly if our data is stored as 108 rows of 1 list each, where each list contains one number. To get our data in this shape, we'll use the pandas `reshape()` method as follows:

```
x = carsales['period'].values.reshape(-1,1)
y = carsales['sales'].values.reshape(-1,1)
```

If you run `print(x)` and `print(y)`, you can see the new shape of the data: 108 rows of one-element lists. Actually performing the linear regression takes little code. We can do the whole thing, including importing the relevant module, with three lines:

```
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(x, y)
```

Here, we import the linear regression capability from the scikit-learn package, which can be referred to by its standard abbreviation, sklearn. This package, which is extremely popular in the machine learning world, provides many useful machine learning capabilities, including linear regression. After importing sklearn, we define the variable `regressor`. A *regressor*, as its name tautologically suggests, is a Python object that we'll use to perform regression. After creating the regressor, we tell it to `fit` our `x` and `y` variables. We are telling it to calculate the line shown in Figure 2-2 that fits the data by matching its location and general trend.

A more quantitative way to describe what *fitting our regression* means is that it's determining precise, optimized values for two numbers: a coefficient and an intercept. After running the preceding snippet, we can look at both of these numbers as follows:

```
print(regressor.coef_)
print(regressor.intercept_)
```

This code prints out two numbers that are output by the regressor's `fit()` method: an intercept, which you should see is about 10,250.8; and a variable called `coef_`, which is short for *coefficients*, and should be equal to about 81.2. Together, these two numbers specify the exact position and trend of the dashed regression line you see in Figure 2-2. You'll see how they do this in the next section.

Applying Algebra to the Regression Line

To see how these two numbers specify the regression line, think back to your high school math classes. You may remember learning that every straight line can be expressed in a form like this:

$$y = m \cdot x + b$$

Here, m is a slope, or coefficient, and b is the intercept (technically a *y-intercept*—the exact place where the line crosses the plot's y-axis). In this case, the value of the coef_ variable we found, about 81.2, is the value of m, and the value of the intercept variable we found, about 10,250.8, is the value of b. So, what we have learned from our regression process is that the relationship between time period and car sales can be expressed, at least approximately, as follows:

$$car\ sales = 81.2 \cdot period + 10250.8$$

The chaos of the car sales dataset's apparently random variation (shown in Figure 2-1) is now reduced to the order of this simple equation. The line that this equation describes is the dashed line in Figure 2-2. We can think of every point on that line as a prediction of how many car sales are expected at each time period, ignoring the distracting randomness and noise.

The m and b values in our equation have useful interpretations. The interpretation of the line's slope, 81.2, is the monthly growth trend of car sales. Based on the data we've observed in the past, we conclude that car sales in Quebec grow by about 81.2 cars per month. Randomness and other variation remain, but a growth of 81.2 is what we approximately expect. The interpretation of the intercept variable, 10,250.8, is the *baseline* value of car sales: the expected car sales in month 0 after "removing" or ignoring the chaos of seasonal variation, the passage of time, and other influences.

The equation that linear regression finds can also be called a *model*, a quantitative description of how two or more variables relate to each other. So when we perform the preceding steps, we can say that we *fit a regression*, or equivalently we can say that we *trained a model*. Our regression, or equivalently our model, tells us that we expect to sell about 10,250.8 cars at the beginning of the time frame in our data, and we expect to sell about 81.2 more cars every month than we sold in the previous month.

It's natural to wonder how our regressor determined that 81.2 and 10,250.8 (the coef_ and intercept outputs of our regressor) are the best values for m and b in our regression line. The line they specify looks good enough in Figure 2-2, but it's not the only line we could draw through our cloud of points. A literally infinite number of conceivable lines also go through our cloud and could be said to fit our data. For example, we might hypothesize that the following line is a better approximation of the relationship between time period and sales:

$$car\ sales = 125 \cdot period + 8000$$

Let's call this new line our *hypothesized line.* If we use it as a model of our data, we have a new *m* and *b,* and so we have a new interpretation. In particular, the slope of this line is 125, which we would interpret as an expectation that monthly car sales will increase by about 125 every month—significantly higher than 81.2, the estimate from our regression line. Let's plot our regression line and this new hypothesized line together with the data as follows:

```
plt.scatter(carsales['period'],carsales['sales'])
plt.plot(carsales['period'],[81.2 * i + 10250.8 for i in \
carsales['period']],'r-',label='Regression Line')
plt.plot(carsales['period'],[125 * i + 8000 for i in \
carsales['period']],'r--',label='Hypothesized Line')
plt.legend(loc="upper left")
plt.title('Car Sales by Month')
plt.xlabel('Month')
plt.ylabel('Sales')
plt.show()
```

You can see the output of this snippet in Figure 2-3, where we've drawn the data, our regression line (the shallow solid line), and our new hypothesized line (the steeper dashed line).
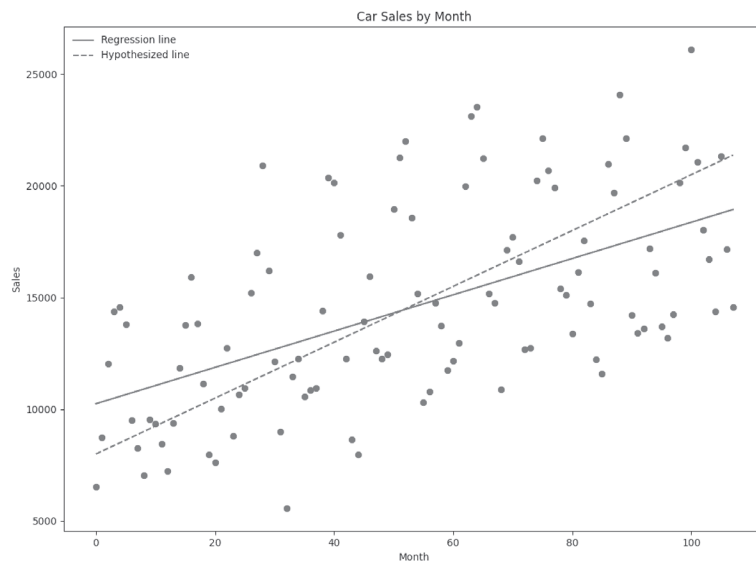

Figure 2-3: A regression line and a steeper line that also fits the data

Both lines go through our cloud of points. Both show an upward trend over time. Both are reasonable candidates to be approximations of the relationship between time and sales, and both could be said to fit the data. Why has our regressor output one line instead of the other? We said that the regression line output by the linear regression process is the *line of best fit.* What is it that enables us to say that it fits better than any other line?

Calculating Error Measurements

We can find the answer by looking at measurements related to *regression errors*. Remember that we interpret each point of a regression line as our prediction of what value we expect in the data. Figure 2-4 shows a regression line and the data used to create it.
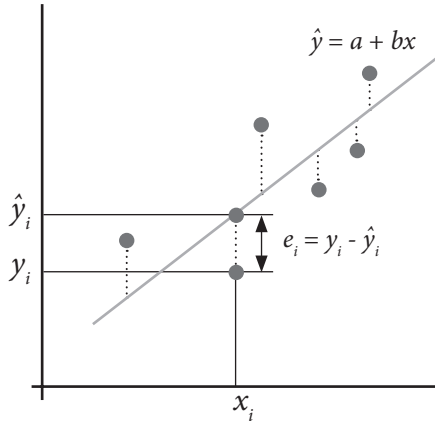

Figure 2-4: Regression errors: vertical distances between points and a regression line

You can see that this regression line is a good fit to the data, meaning it gets close to most of the illustrated points. However, it's not a perfect fit. For every data point, we can calculate the vertical distance between the data point and the regression line. The regression line predicts a certain value, and the point in the data has a particular distance from that prediction. This distance between a predicted and an actual value of a data point is called the regression's *error* relative to that point. In Figure 2-4, the variable e_i is an error measurement for one of the points in the data. You can see that e_i is the vertical distance between a particular point and the regression line. We can calculate this distance for every point in our data.

Calculating the error relative to each data point will give us a way to quantify how well any line fits our data. Lines with low errors fit the data well, and lines with high errors fit the data poorly. That's why we say that measuring regression errors is one way to measure a regression line's *goodness of fit*, the degree to which a line fits the data well.

Let's calculate those error measurements for our car sales regression. We'll calculate each point of the lines we're interested in and compare those points to each point of our dataset:

```
saleslist=carsales['sales'].tolist()
regressionline=[81.2 * i + 10250.8 for i in carsales['period']]
hypothesizedline=[125 * i + 8000 for i in carsales['period']]
error1=[(x-y) for x, y in zip(regressionline,saleslist)]
error2=[(x-y) for x, y in zip(hypothesizedline,saleslist)]
```

In this snippet, we create `saleslist`, a variable that includes the raw car sales numbers for every month. Then we create two variables, `regressionline` and `hypothesizedline`. These variables record every point on the regression and hypothesized lines, respectively. We want to measure how far each true sales number is from both of these lines, so we create two more variables: `error1` to record the distance between true sales numbers and the regression line, and `error2` to record the distance between true sales numbers and the hypothesized line.

We can print out these variables to look at what errors we find for both of our lines:

```
print(error1)
print(error2)
```

When you look at these lists of errors, you can see 108 separate measurements of how far these lines are from the raw data. These 108 measurements are an expression of how well these lines fit the raw data. However, looking at all 216 of these measurements at once is difficult. It would be easier if we could boil down all this information indicating how well a line fits to just one number. The following snippet shows one way to do this:

```
import numpy as np

error1abs=[abs(value) for value in error1]
error2abs=[abs(value) for value in error2]

print(np.mean(error1abs))
print(np.mean(error2abs))
```

In this snippet, we import Python's NumPy package. NumPy is used often in data science, especially for calculations with arrays and matrices. Here, we import it because it gives us the ability to find the mean of a list. Then we define two new variables: `error1abs` and `error2abs`, each containing a list of the absolute values of our error measurements for our two respective lines. Finally, we take the means of these lists.

The means that we find are called the *mean absolute error (MAE)* measurements of each line. Hopefully, the MAE feels like an intuitive measurement of error to you: it's just the average vertical distance between a line and the points in a dataset. A line that gets very close to the points in a dataset will have a low MAE, and a line that is very far from most points will have a higher MAE.

The MAE is a reasonable way to express the degree of goodness of fit of a regression line or any other line. The lower the MAE, the better. In this case, we can see that the MAE for our regression line is 3,154.4, while the MAE for our hypothesized line is 3,239.8. At least according to this measurement, the regression line fits the data better than our hypothesized line.

The MAE has an easy interpretation: it's the average error we expect to have if we use a particular line for prediction. When we say that the MAE for our regression line is 3,154.4, we mean that if we use this regression line to make predictions, we expect our predictions to be wrong by about 3,154.4 on average (either 3,154.4 too low or too high).

For example, suppose we predict that three months from now, we will sell exactly 20,000 cars. We wait three months, count our monthly sales, and find that we actually sold 23,154 cars instead of 20,000. Our prediction was wrong; we underestimated car sales by 3,154. So, we're not perfect at prediction, and the size of our prediction error tells us exactly how imperfect we are. Is the size of our error a surprise? The MAE we just measured (3,154.4) tells us that having an error this high isn't surprising—in fact, underestimating by 3,154 is (after rounding) exactly the size of error we expect to encounter in any month when we're using this regression. Sometimes we'll overestimate instead of underestimating, and sometimes we'll have lower or higher errors than 3,154. Regardless, the MAE is telling us that having an error of about 3,154 is what we expect when using this regression for this prediction scenario.

MAE is not the only measurement that indicates how well a line fits a dataset. Let's look at another possible measurement:

```
error1squared=[(value)**2 for value in error1]
error2squared=[(value)**2 for value in error2]

print(np.sqrt(np.mean(error1squared)))
print(np.sqrt(np.mean(error2squared)))
```

Here, we create lists of the squared values of each error. Then we take the square root of the sum of these errors. This measurement is called the *root mean squared error (RMSE).* Lower RMSE values indicate a line that is a better fit—one that's expected to make better predictions.

We can create simple Python functions that perform calculations of MAE and RMSE:

```
def get_mae(line,actual):
    error=[(x-y) for x,y in zip(line,actual)]
    errorabs=[abs(value) for value in error]
    mae=np.mean(errorabs)
    return(mae)

def get_rmse(line,actual):
    error=[(x-y) for x,y in zip(line,actual)]
    errorsquared=[(value)**2 for value in error]
    rmse=np.sqrt(np.mean(errorsquared))
    return(rmse)
```

These functions just calculate MAE and RMSE, respectively, exactly as we did previously. If you run `print(get_rmse(regressionline,saleslist))`, you can see that the RMSE of our regression line is about 3,725, and if you run `print(get_rmse(hypothesizedline,saleslist))`, you can see that the RMSE of our hypothesized line is about 3,969.

You'll notice that the RMSE of our regression line is smaller than the RMSE of our hypothesized line. This enables us to say that the regression line is a better fit to the data than the hypothesized line, according to the RMSE metric.

It's not a coincidence that our regression line has a lower RMSE than our hypothesized line. When we ran the command `regressor.fit(x,y)` in Python earlier, the `regressor.fit()` method performed linear algebraic calculations that were invented by the great mathematician Adrien-Marie Legendre and first published in 1805. Legendre's calculations take a collection of points as an input, and their output is the intercept and coefficients that minimize the value of the RMSE. In other words, the line whose coefficients are determined by Legendre's method is mathematically guaranteed to have a lower RMSE than any of the other infinite possible lines that we could draw to try to fit our data. When we call the regression line the line of best fit, we mean that it is mathematically guaranteed to have the lowest possible RMSE of all possible lines that use the variables we specified. This guarantee is a reason for the enduring popularity of linear regression, and why it's still a standard way to find a line that fits a dataset after all these years.

The line that the regressor outputs is the best-fit line, not just in the loose sense that it looks like it fits the cloud very well, but in the strict quantitative sense that out of all the infinite lines that go through the cloud of points, it is guaranteed to have the lowest RMSE. You can feel free to try other straight lines and check their RMSE values—you won't find one that performs better than our regression line.

Using Regression to Forecast Future Trends

So far, we've used linear regression to find the line that is the best fit of our historical data. But our historical data is all from the past, so we haven't done any real forecasting yet. Going from a linear regression to a forecast is simple: we just need to extrapolate.

The dashed regression line we drew in Figure 2-2 stops at the edges of our plot, at month 0 on the left and month 107 on the right, but there's no reason it needs to stop there. If we continue to draw our regression line farther to the right, we can see the values we expect for any month, however far in the future. Of course, we'll keep the same slope and intercept as we extend the line in this way. Let's write code that will do this:

```
x_extended = np.append(carsales['period'], np.arange(108, 116))
```

Here, we create the variable `x_extended`. This variable is a combination of two sets of numbers. First, it includes the values of our dataset's `period` column that records the periods from 0 to 107 in order. Second, it includes all the numbers 108 through 115 in order—these are meant to represent future months after the end of our data (month 108, month 109, . . . all the way to month 115). We combine these two things by using the `np.append()` method, and the end result is an extended version of our original *x* variable.

Next, we can use our regressor's `predict` method to calculate the values that will be on our regression line for each of the month numbers in `x_extended`:

```
x_extended=x_extended.reshape(-1,1)
extended_prediction=regressor.predict(x_extended)
```

Now we have the forecast values stored in the variable `extended_prediction`. If you look at `extended_prediction`, you can see what these predictions are. These predictions follow a simple pattern: each is about 81.2 higher than the previous one. This is because 81.2 is the slope of our regression line. Remember, 81.2 is not just the slope of the line but also the size of the increase we expect in car sales every month, ignoring randomness and seasonal variation.

The prediction method we used here is helpful, but we don't really need it. We can get any values we want on our regression line just by plugging in numbers to our regression equation:
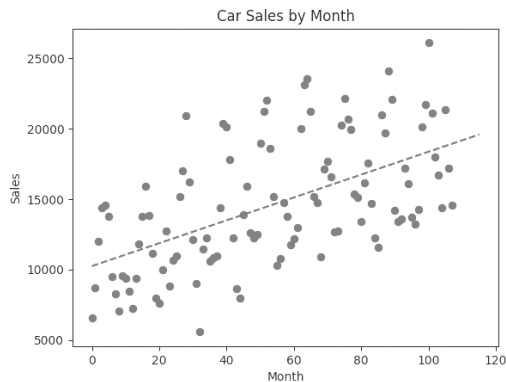
$$car\ sales = 81.2 \cdot period + 10250.8$$

However we get the next predicted values, we can plot them and see what they look like on a graph (Figure 2-5):

```
plt.scatter(carsales['period'],carsales['sales'])
plt.plot(x_extended,extended_prediction,'r--')
plt.title('Car Sales by Month')
plt.xlabel('Month')
plt.ylabel('Sales')
plt.show()
```


Figure 2-5: A regression line extrapolated several periods forward, for forecasting

This plot will probably not surprise you. It looks almost identical to Figure 2-2, and it's supposed to. The only difference is that we've extended our regression line out a few more periods to the right, to see what car sales we expect—that is, how many we forecast—in the near future. This extrapolation of a regression line is a simple but effective way to forecast.

We've accomplished forecasting with linear regression, but there's more we can do to improve our forecasts. In the next sections, we'll talk about ways to evaluate and improve the performance of our forecasts.

Trying More Regression Models

The linear regression we did in the previous sections is a simple kind called *univariate linear regression.* This type of regression uses only one variable to predict one other variable. In our case, we used the period variable alone to predict sales. Using only one variable has a couple of advantages: first, it is easy; and second, it creates a simple, straight line that expresses some order in the data without also including its random noise. But we have other options.

Multivariate Linear Regression to Predict Sales

If we use other variables to predict sales as well as just the period, we can perform a more complex kind of regression called *multivariate linear regression.* The details of multivariate linear regression are essentially the same as univariate linear regression; the only real difference is the number of variables we use for prediction. We can use any variables we like for multivariate regression: gross domestic product (GDP) growth rates, population estimates, car prices, inflation rates, or anything else we want.

For now, we're limited because our dataset doesn't contain any of those variables. It contains only the period and the sales. However, we can still perform multivariate regression, by using variables that we derive from the period variable. For example, we could use $period^2$ as a new variable in a multivariate regression, or $\log(period)$, or any other mathematical transformation of the period variable.

Remember that when we performed regression before, we found the m and the b (slope and intercept) variables in the following equation:

$$y = m \cdot x + b$$

When we use multiple variables to predict car sales, we're also finding slope and intercept variables. The only difference is that we're also finding more variables. If we're using three variables to do prediction (which we can call x_1, x_2, and x_3), then we're finding the m_1, m_2, m_3, and b variables in the following equation:

$$y = m_1 \cdot x_1 + m_2 \cdot x_2 + m_3 \cdot x_3 + b$$

The idea is the same as in univariate regression, but we end up with more slopes for more predictor variables. If we want to use $period$, $period^2$, and $period^3$ to predict car sales in our regression, we'll need to estimate the m_1, m_2, m_3, and b variables in Equation 2-1:

$$car\ sales = m_1 \cdot period + m_2 \cdot period^2 + m_3 \cdot period^3 + b$$

Equation 2-1: An equation for multivariate regression using our car sales data

Let's look at the code that will create these transformations of our period variable and do linear regression with three variables:

```python
carsales['quadratic']=carsales['period'].apply(lambda x: x**2)
carsales['cubic']=carsales['period'].apply(lambda x: x**3)

x3 = carsales.loc[:,['period','quadratic','cubic']].values.reshape(-1,3)
y = carsales['sales'].values.reshape(-1,1)

regressor_cubic = LinearRegression()
regressor_cubic.fit(x3, y)
plt.scatter(carsales['period'],carsales['sales'])
plt.plot(x,regressor.predict(x),'r-')
plt.plot(x,regressor_cubic.predict(x3),'r--')
plt.title('Car Sales by Month')
plt.xlabel('Month')
plt.ylabel('Sales')
plt.show()
```

In this snippet, we define two new variables: quadratic, whose value is equal to $period^2$, and cubic, whose value is equal to $period^3$. Then, we define a new x3 dataframe that includes all three of these new variables, and we reshape it so that they will be the right shape for our regressor. The right shape for this three-variable multivariate regression is an array of 108 rows, in which each row is a list of the values of our three variables for a particular month. As long as the data is in the right shape, we can use the fit() method for any univariate or multivariate linear regression with any number of variables. After calling fit(), we calculate the values predicted by this regression for our data and plot them. This snippet creates the plot in Figure 2-6.
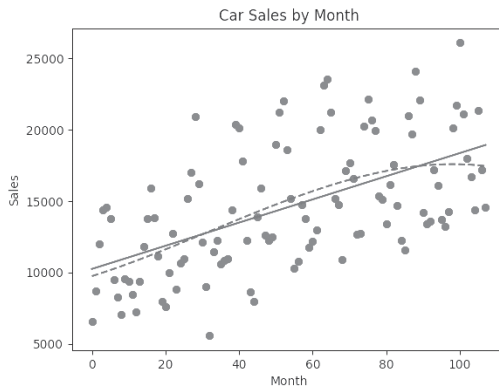

Figure 2-6: A curve that also fits the data

Here, you can see two regression lines. One is the (solid) straight line that is the result of our previous (univariate) regression. The other, newer regression line is not a straight line, but rather a (dashed) curve—a *cubic curve*, to be precise. Linear regression was originally designed to work with

straight lines (hence the name *linear*), but we can also use it to find best-fit curves and nonlinear functions like the cubic polynomial in Figure 2-6.

Whether we find a best-fit straight line or a best-fit curve, the linear regression methods that we're using are exactly the same. Similarly, using multiple variables for prediction is not really different from univariate regression with one variable: the output still fits our data, and in fact, our new curve goes very close to the straight line. Every time we select different variables for our regression, the output will look a little different: it may have a different shape or a different curve. But it will always fit the data. In this case, if you want to know the unknown variables in Equation 2-1, we can print them out as follows:

```
print(regressor_cubic.coef_)
print(regressor_cubic.intercept_)
```

The outputs from these `print()` statements are the following:

```
[[ 8.13410634e+01 7.90279561e-01 -8.19451188e-03]]
[9746.41276055]
```

These outputs enable us to fill in all the variables in Equation 2-1 to get an equation for estimating car sales using a cubic polynomial of the period:

$$car\ sales = 81.34 \cdot period + 0.79 \cdot period^2 - 0.008 \cdot period^3 + 9746.41$$

One important thing to notice about Figure 2-6 is the different behavior of our regression lines during the last few periods, on the right side of the plot. The straight line from our univariate regression increases by about 81.2 every period, and when we extrapolate it farther to the right, it will continue to predict increases of about 81.2 every period. By contrast, the curved line from our multivariate regression begins to curve downward on the right side of the plot. If we extrapolated it farther to the right, it would predict a decrease in car sales every month forever.

These two lines, though they behave similarly and are both the result of linear regression, make opposite predictions about the future: one predicts growth, and the other predicts contraction. Later in the chapter, we'll talk more about how to choose which regression line to use for forecasting.

Trigonometry to Capture Variations

There's no limit to the number of variables we can add to a multivariate regression. Each selection of variables will lead to a curve with a slightly different shape. One of the difficult choices we have to make in every regression problem is which variables to add to the regression.

In this case, the univariate regression line (the straight line in Figure 2-2) and the cubic regression line (the curved line in Figure 2-6) are both acceptable and can be used to forecast the future. However, though they both pass through what looks like the middle of our cloud of points, there is so much variation that they don't capture—sales for many individual months are much

higher or much lower than these lines. Ideally, we could find a collection of variables that, when fit using a linear regression, lead to a curve that better fits some of this variation. In this case, making one small change to the way we plot our data can make what we should do next clearer.
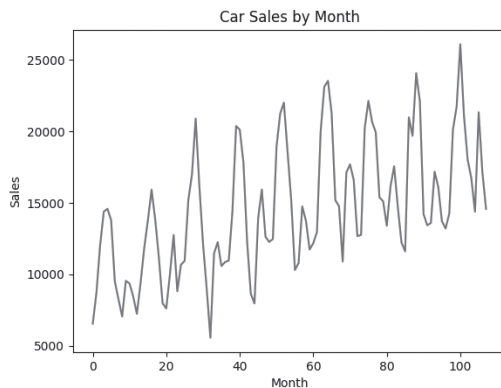
Let's change Figure 2-1 from a scatterplot to a line plot, by making just one small change in our code (shown in bold):

```
from matplotlib import pyplot as plt
plt.plot(carsales['period'],carsales['sales'])
plt.title('Car Sales by Month')
plt.xlabel('Month')
plt.ylabel('Sales')
plt.show()
```

Figure 2-7 shows the new plot.

Figure 2-7: A line plot makes the patterns within years
(high summers and low winters) more apparent.

This new plot shows the same data, but plotted as a line rather than a collection of points. With a line plot, another pattern becomes much clearer. We see that the noisy ups and downs of monthly sales within individual years are more ordered than they looked in the scatterplot.

In particular, our data includes nine years of sales figures, and exactly nine major peaks are apparent in the contour of the line plot. What looked like totally random noise actually has some structure: a predictable peak in sales occurs every summer, with a corresponding trough every winter. If you think about it a little more, you might realize why variation could exist within a year: it's because this data comes from Quebec, where very cold winters are associated with lower activity levels, and beautiful warm summers are associated with going outside and shopping and taking long road trips that require cars.

Now that you can see the way the number of car sales goes up and down during a year, maybe it reminds you of a mathematical function. In fact, the pattern of periodic increases and decreases looks like a trigonometric curve, like a sine or cosine curve. Figure 2-8 shows an example of sine and cosine curves.
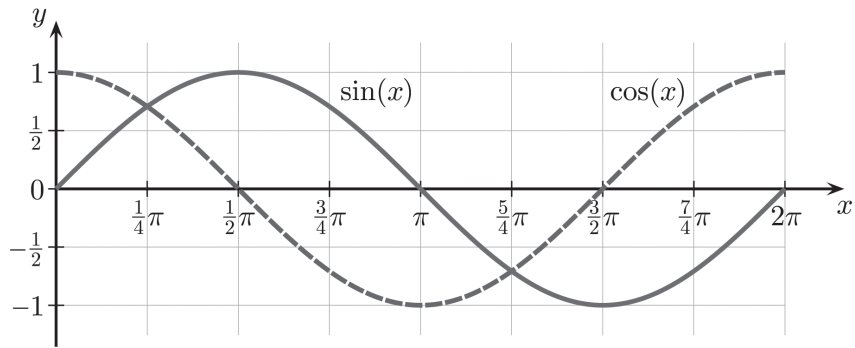
Figure 2-8: A plot of a sine curve and a cosine curve

Let's try a regression that uses the sine and cosine of the period in a multivariate regression:

```
import math
carsales['sin_period']=carsales['period'].apply(lambda x: math.sin(x*2*math.pi/12))
carsales['cos_period']=carsales['period'].apply(lambda x: math.cos(x*2*math.pi/12))

x_trig = carsales.loc[:,['period','sin_period','cos_period']].values.reshape(-1,3)
y = carsales['sales'].values.reshape(-1,1)

regressor_trig = LinearRegression()
regressor_trig.fit(x_trig, y)

plt.plot(carsales['period'],carsales['sales'])
plt.plot(x,regressor_trig.predict(x_trig),'r--')
plt.title('Car Sales by Month')
plt.xlabel('Month')
plt.ylabel('Sales')
plt.show()
```

In this snippet, we define sine and cosine transformations of the `period` variable, and then we fit a regression that uses these new variables as predictors. Finally, we plot the result, which is shown in Figure 2-9.
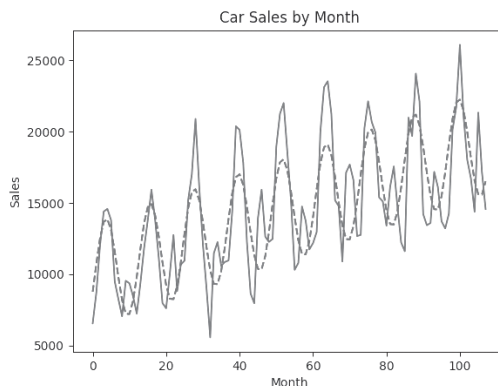

Figure 2-9: A trigonometric curve fit to our data

In Figure 2-9, you can see the raw sales data plotted as a solid line, and the trigonometric regression curve plotted as a dashed line. You can see that we're really getting somewhere now. The regression that relies on trigonometric functions seems to fit the data especially well. In particular, it seems to go up during the yearly peaks and down during the yearly troughs, thereby getting much closer to the true sales numbers. We can verify that this trigonometric curve has a lower RMSE than the straight line as follows:

```
trig_line=regressor_trig.predict(x_trig)[:, 0]
print(get_rmse(trig_line,saleslist))
```

The RMSE we get as output is the lowest one we've seen yet: about 2,681. It is not entirely a coincidence that trigonometric functions enable us to fit the data well. In fact, the increases and decreases in temperature during seasons on our planet are due to a change in the angle of the Earth during its revolution around the sun. The change in the angle of the Earth with respect to the sun follows a curve that's like a sine curve, and therefore temperature changes throughout each year also follow sine-like curves. If car sales are reacting to winter and summer weather changes due to temperature, it makes sense that they would also follow sine-like curves. Regardless of whether we found the trigonometric model by blind chance, by looking at our scatterplot in Figure 2-1, or because we know about the astronomy of the Earth's rotation around the sun, it seems like we've found a good regression curve that fits the data well.

Choosing the Best Regression to Use for Forecasting

We've observed that the regression line that includes terms for the sine and cosine of the period seem to fit the data well. When we say that this line fits the data well, we mean that, qualitatively, the dashed line in Figure 2-9 gets quite close to the solid line. More precisely, we mean that quantitatively, the RMSE for the trigonometric line is lower than the RMSE for the other lines we've looked at. Whenever we find a model with a lower RMSE, we are getting a model that fits our data better.

The natural temptation is to keep looking for new regression specifications that have lower and lower RMSEs. For example, let's try a new regression specification that includes seven prediction terms to forecast sales, and find the RMSE for that model:

```
carsales['squareroot']=carsales['period'].apply(lambda x: x**0.5)
carsales['exponent15']=carsales['period'].apply(lambda x: x**1.5)
carsales['log']=carsales['period'].apply(lambda x: math.log(x+1))

x_complex = carsales.loc[:,['period','log','sin_period','cos_period', \
'squareroot','exponent15','log','quadratic', 'cubic']].values.reshape(-1,9)
y = carsales['sales'].values.reshape(-1,1)

regressor_complex = LinearRegression()
regressor_complex.fit(x_complex,y)
```

```
complex_line=[prediction for sublist in regressor_complex.predict(x_complex) \
for prediction in sublist]
print(get_rmse(complex_line,saleslist))
```

In this snippet, we repeated steps that we've done before: define some variables, use the variables in a linear regression, and check the RMSE of the regression. Notice the backslash (\) at the end of a few of the lines. These are *line continuation characters*: they're telling Python that the line they're on and the next line should be treated as one single line of code. We are using them here because the full line doesn't fit on the book's page. At home, you can use line continuation characters, or you can ignore them if you're able to type the full lines without breaks.

At the end of the preceding snippet, we check the RMSE for this new regression model, and we find that it's about 2,610: even lower than the RMSE for the trigonometric model shown in Figure 2-9. If RMSE is our metric for judging how well a model fits, and we have gotten the lowest RMSE yet, it probably seems natural to conclude that this is our best model yet, and we should use this model for forecasting.

But be careful; this apparently reasonable conclusion is not correct. The approach we've been taking to model selection has a problem: it doesn't fully resemble the reality of forecasting as we encounter it in real life. Think about what we've done. We've used past data to fit a regression line and then judged how good that regression line is, based on how close it gets to past data points (its RMSE). We're using the *past* both for fitting our regression line and for judging its performance. In a real-world forecasting scenario, we'll use the past to fit our regression line, but we should use the *future* for judging its performance. A forecasting method is worthwhile only if it can predict the unknown future.

When we're choosing the best regression line to use for forecasting, we want to find a way to evaluate various regression lines based on their performance on future data. This is not possible because the future hasn't happened yet, so we can't ever have future data. But we can make a small change in the way we perform and evaluate regressions so that our measurements of performance on past data give a good estimate of how they'll perform when predicting the future.

What we need to do is split our full dataset into two separate, mutually exclusive subsets: a *training set,* consisting of the majority of our data, and a *test set*, consisting of the rest. We'll use only the training set to fit our regressions, or in other words, to *train* them. After fitting/training our regressions, we'll use the test set to evaluate how good the regressions are, using metrics like RMSE or MAE.

This simple change makes an important difference. Instead of evaluating performance based on the same data used to fit our regressions, we are evaluating based on separate data that was not used during the fitting process. Our test set is from the past, but it's *as if* it's from the future, since it's not used to determine the coefficients and intercept in our regression, and it's used only to test how good the regression's predictions are. Since the test set hasn't been used to fit our regressions, we sometimes say that the

regressions haven't *learned from* the test data, or that it's *as if* the test data is from the future. By having some data that's as if it's from the future, our regression evaluation more closely resembles a true forecasting process, where prediction of the future is the most important goal.

Let's look at the code to accomplish this training/test split, and then we'll see what makes it work so well:

```
x_complex_train = carsales.loc[0:80,['period','log','sin_period','cos_period','squareroot', \
'exponent15','log','quadratic','cubic']].values.reshape(-1,9)
y_train = carsales.loc[0:80,'sales'].values.reshape(-1,1)

x_complex_test = carsales.loc[81:107,['period','log','sin_period','cos_period','squareroot', \
'exponent15','log','quadratic','cubic']].values.reshape(-1,9)
y_test = carsales.loc[81:107,'sales'].values.reshape(-1,1)

regressor_complex.fit(x_complex_train, y_train)
```

Here, we split the data into two sets: a training set and a test set. We use the training set to train the data (to fit a regression line). We can then use the test set to test how well our regression performs. If you think about this approach, it resembles an actual forecasting situation: we train a model knowing only the past, but the model has to do well on data that wasn't used to train it (the future, or data that's as if it's from the future). Creating a test set like this is essentially creating a simulated future.

In the preceding snippet, we use the first 81 time periods as our training data and the rest (27 time periods) as our test data. In percentage terms, we use 75 percent of our data for training and reserve about 25 percent for testing. Splitting training and test data in proportions close to this is common: 70 percent training data and 30 percent testing data is also common, as are 80/20 and 90/10 splits. We usually keep the large majority of our data in the training set since finding the right regression line is crucial, and using more data for training can help us find the best regression line (the one with the most predictive accuracy). At the same time, we need more than a negligible amount of data in the test set, since we also need to get an accurate estimate of how our regression is expected to perform with new data.

After we've created training and test sets, we can test our different regression models on the test set and check the RMSE or the MAE for each model. The model that has the lowest RMSE or MAE on the test set is a reasonable choice for the model we can use for forecasts of the actual future. Let's check the RMSE for several of the regressions we've run so far:

```
x_train = carsales.loc[0:80,['period']].values.reshape(-1,1)
x_test = carsales.loc[81:107,['period']].values.reshape(-1,1)
x_trig_train = carsales.loc[0:80,['period','sin_period','cos_period']].values.reshape(-1,3)
x_trig_test = carsales.loc[81:107,['period','sin_period','cos_period']].values.reshape(-1,3)

regressor.fit(x_train, y_train)
regressor_trig.fit(x_trig_train, y_train)
```

```
complex_test_predictions=[prediction for sublist in \
      regressor_complex.predict(x_complex_test) for prediction in sublist]
test_predictions=[prediction for sublist in regressor.predict(x_test) for \
      prediction in sublist]
trig_test_predictions=[prediction for sublist in \
      regressor_trig.predict(x_trig_test) for prediction in sublist]

print(get_rmse(test_predictions,saleslist[81:107]))
print(get_rmse(trig_test_predictions,saleslist[81:107]))
print(get_rmse(complex_test_predictions,saleslist[81:107]))
```

After you run the preceding snippet, you can see that our univariate regression has an RMSE of about 4,116 on the test set. The trigonometric multivariate regression has an RMSE of about 3,461—much better than the univariate regression. By contrast, the complex regression model that includes nine prediction terms has an RMSE of about 6,006 on the test set—an awful performance. Though it had excellent performance on the training set, we find that it has awful performance on the test set.

This complex model shows a particularly bad example of *overfitting*. In this common machine learning problem, a model is too complex and fits the data's noise and coincidences instead of the data's true patterns. Overfitting often happens when our attempts to get low errors on a training set lead to us getting much higher errors on a test set.

For example, suppose that by some coincidence, car sales in Quebec spiked every time the star Betelgeuse had a V-band magnitude greater than 0.6 between 1960 and 1968. If we included Betelguese's V-band magnitude as a parameter in our regressions, we would find that our RMSE was quite low when predicting 1960 to 1968 because of this coincidence. Finding a low RMSE might make us quite confident that we had a great model that would perform well. We might extrapolate this pattern into the future and forecast future sales spikes at future high points of Betelgeuse's brightness cycle. However, since the past relationship between Betelgeuse and car sales was only coincidental, extrapolating this pattern into the future would give us huge errors; it would cause RMSE on future predictions to be quite high. The Betelgeuse/car sales relationship was only noise, and our regressions are supposed to capture only true signals, not noise. Including Betelguese's brightness measurements in our regression would be an example of overfitting, since our zeal to decrease RMSE for the past would lead us to increase RMSE in the future.

This example should make it clear that using error measurements on a training set to choose the best model could lead us to choose a model that has high error measurements on the test set. For that reason, error measurement on the test set is the right metric to use to compare models in all forecasting tasks. As a general rule, you can expect overfitting to happen when you've included too many irrelevant variables in your regression. So, you can avoid overfitting by removing the irrelevant variables (like the brightness of Betelgeuse) from your regressions.

The problem is that we're not always entirely sure which variables are irrelevant and which ones are actually useful. That's why we have to try several models and check performance. Find the model that has the lowest RMSE on the test set, and that will be the one that has the right mix of variables and doesn't lead you to get distracted by coincidences and overfit.

Now that we've compared models based on their RMSE on the test set, we can choose the trigonometric model as our best model so far. We can extrapolate one period forward in this model and determine a forecast for consumer demand next month, just as we extrapolated for our univariate model before. We can report this number back to the business as an estimate based on rigorous linear regression analysis. Not only that, we can explain why we made this prediction and why we used our model, including the idea of the best-fit line, the trigonometric modeling of the seasons, and the favorable (low) errors on the test set. If no objections or countervailing business considerations arise, we can order this number of cars next month, and we can expect that customers will want to purchase close to this number of cars.

Exploring Further

Linear regression and forecasting are both topics that can fill many textbooks. If you continue in your data science education, you'll have a chance to learn many of the subtleties and nuances related to these subjects.

One thing you should consider studying if you want to get to an advanced level in data science is the linear algebra behind linear regression. You can think of each observation in your data as a row of a matrix, and then you can use matrix multiplication and matrix inversion to calculate the line of best fit, instead of relying on a Python library to do the calculations for you. If you deeply explore these linear algebra concepts, you'll learn about the mathematical assumptions underlying linear regression. Understanding these mathematical assumptions will enable you to more accurately judge whether linear regression is the best method to use with your data, or whether you should use some of the methods described later in the book instead (especially the supervised learning topics discussed in Chapter 6).

Another issue you should become familiar with is the limitation of linear regression as a forecasting method. As the name indicates, linear regression is a linear method, and it's meant to be used with variables that have linear relationships. For example, if customers order about 10 more units of your product each week than they did the previous week, a linear relationship exists between time and customer demand, and linear regression would be a perfect method to measure that increase and forecast future customer demand. On the other hand, if your sales double every week for a year and then suddenly crash and then slowly rise again for a while, the relationship between time and sales would be highly nonlinear, and linear regression may not yield accurate predictions.

Similarly, remember that when we use linear regression for forecasting, we are extrapolating past growth to predict future growth. If certain

circumstances aren't present or accounted for in your historical data, your linear regression won't be able to accurately predict their occurrence in the future. For example, if you use data from steady, prosperous years as your training data, you'll probably predict steady, prosperous growth in the future. Instead, you may find that a global financial crisis or pandemic changes everything, and since the regression's training data didn't include a pandemic, no prediction of any pandemic will be given for the future. Regression works only when the future resembles the past. Some events like wars and pandemics are so inherently unpredictable that regression can never give us completely accurate predictions about them. In those cases, preparation is more important than forecasting; make sure your business is ready for hard times and surprises instead of assuming that linear regression will always give you completely correct answers. Though forecasting is important and linear regression is powerful, it's important to remember that these limitations exist.

Summary

We began this chapter with a common business scenario: a company needs to decide how much new inventory it should order. We used linear regression as our main forecasting tool, and we looked a little at the programming side of it (how to write code for regression), the statistical side of it (which error metrics we can use to determine a model's goodness of fit), and the math side of it (why our particular line is the best-fit line). After we went through all these aspects of the problem, we arrived at a model that we thought was best, which we could use to obtain a forecast of next month's consumer demand.

This scenario—considering a business problem and using programming, mathematical theory, and common sense to find a data-driven solution—is typical of data science. In the remaining chapters, we'll examine other business scenarios and talk about how to use data science to find ideal solutions to them. In the next chapter, we'll go over data distributions and show how to test two groups to see whether they're significantly different from each other.