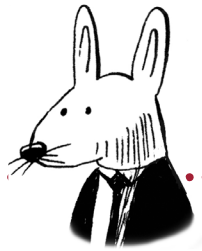
Drawing a line close to our points: Linear regression

3



In this chapter

- what is linear regression
- fitting a line through a set of data points
- coding the linear regression algorithm in Python
- using Turi Create to build a linear regression model to predict housing prices in a real dataset
- what is polynomial regression
- fitting a more complex curve to nonlinear data
- discussing examples of linear regression in the real world, such as medical applications and recommender systems

In this chapter, we will learn about linear regression. Linear regression is a powerful and widely used method to estimate values, such as the price of a house, the value of a certain stock, the life expectancy of an individual, or the amount of time a user will watch a video or spend on a website. You may have seen linear regression before as a plethora of complicated formulas including derivatives, systems of equations, and determinants. However, we can also see linear regression in a more graphical and less formulaic way. In this chapter, to understand linear regression, all you need is the ability to visualize points and lines moving around.

Let's say that we have some points that roughly look like they are forming a line, as shown in figure 3.1.

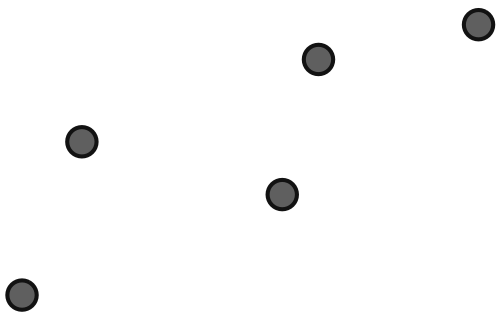


Figure 3.1 Some points that roughly look like they are forming a line

The goal of linear regression is to draw the line that passes as close to these points as possible. What line would you draw that passes close to those points? How about the one shown in figure 3.2?

Think of the points as houses in a town, and our goal is to build a road that goes through the town. We want the line to pass as close as possible to the points because the town's inhabitants all want to live close to the road, and our goal is to please them as much as we can.

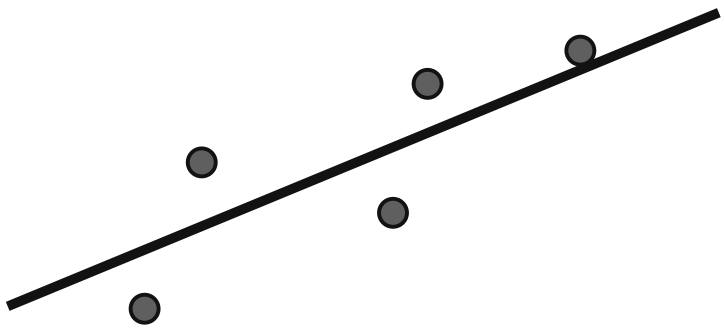


Figure 3.2 A line that passes close to the points

We can also imagine the points as magnets lying bolted to the floor (so they can't move). Now imagine throwing a straight metal rod on top of them. The rod will move around, but because the magnets pull it, it will eventually end up in a position of equilibrium, as close as it can to all the points.

Of course, this can lead to a lot of ambiguity. Do we want a road that goes somewhat close to all the houses, or maybe really close to a few of them and a bit farther from others? Some questions that arise follow:

- What do we mean by "points that roughly look like they are forming a line"?
- What do we mean by "a line that passes really close to the points"?
- How do we find such a line?
- Why is this useful in the real world?
- Why is this machine learning?

In this chapter we answer all these questions, and we build a linear regression model to predict housing prices in a real dataset.

You can find all the code for this chapter in the following GitHub repository: https://github.com/luisguiserrano/manning/tree/master/Chapter_3_Linear_Regression.

The problem: We need to predict the price of a house

Let's say that we are real estate agents in charge of selling a new house. We don't know the price, and we want to infer it by comparing it with other houses. We look at features of the house that could influence the price, such as size, number of rooms, location, crime rate, school quality, and distance to commerce. At the end of the day, we want a formula for all these features that gives us the price of the house, or at least a good estimate for it.

The solution: Building a regression model for housing prices

Let's go with as simple an example as possible. We look at only one of the features—the number of rooms. Our house has four rooms, and there are six houses nearby, with one, two, three, five, six, and seven rooms, respectively. Their prices are shown in table 3.1.

Table 3.1 A table of houses with the number of rooms and prices. House 4 is the one whose price we are trying to infer.

Number of rooms	Price
1	150
2	200
3	250
4	?
5	350
6	400
7	450

What price would you give to house 4, just based on the information on this table? If you said \$300, then we made the same guess. You probably saw a pattern and used it to infer the price of the house. What you did in your head was linear regression. Let's study this pattern more. You may have noticed that each time you add a room, \$50 is added to the price of the house. More specifically, we can think of the price of a house as a combination of two things: a base price of \$100, and an extra charge of \$50 for each of the rooms. This can be summarized in a simple formula:

$$\text{Price} = 100 + 50(\text{Number of rooms})$$

What we did here is come up with a model represented by a formula that gives us a *prediction* of the price of the house, based on the *feature*, which is the number of rooms. The price per room is called the *weight* of that corresponding feature, and the base price is called the *bias* of the model. These are all important concepts in machine learning. We learned some of them in chapter 1 and 2, but let's refresh our memory by defining them from the perspective of this problem.

features The features of a data point are those properties that we use to make our prediction. In this case, the features are the number of rooms in the house, the crime rate, the age of the house, the size, and so on. For our case, we've decided on one feature: the number of rooms in the house.

labels This is the target that we try to predict from the features. In this case, the label is the price of the house.

model A machine learning model is a rule, or a formula, which predicts a label from the features. In this case, the model is the equation we found for the price.

prediction The prediction is the output of the model. If the model says, "I think the house with four rooms is going to cost $300," then the prediction is 300.

weights In the formula corresponding to the model, each feature is multiplied by a corresponding factor. These factors are the weights. In the previous formula, the only feature is the number of rooms, and its corresponding weight is 50.

bias As you can see, the formula corresponding to the model has a constant that is not attached to any of the features. This constant is called the bias. In this model, the bias is 100, and it corresponds to the base price of a house.

Now the question is, how did we come up with this formula? Or more specifically, how do we get the computer to come up with this weight and bias? To illustrate this, let's look at a slightly more complicated example. And because this is a machine learning problem, we will approach it using the remember-formulate-predict framework that we learned in chapter 2. More specifically, we'll *remember* the prices of other houses, *formulate* a model for the price, and use this model to *predict* the price of a new house.

The remember step: Looking at the prices of existing houses

To see the process more clearly, let's look at a slightly more complicated dataset, such as the one in table 3.2.

Table 3.2 A slightly more complicated dataset of houses with their number of rooms and their price

Number of rooms	Price
1	155
2	197
3	244
4	?
5	356
6	407
7	448

This dataset is similar to the previous one, except now the prices don't follow a nice pattern, where each price is $50 more than the previous one. However, it's not that far from the original dataset, so we can expect that a similar pattern should approximate these values well.

Normally, the first thing we do when we get a new dataset is to plot it. In figure 3.3, we can see a plot of the points in a coordinate system in which the horizontal axis represents the number of rooms, and the vertical axis represents the price of the house.

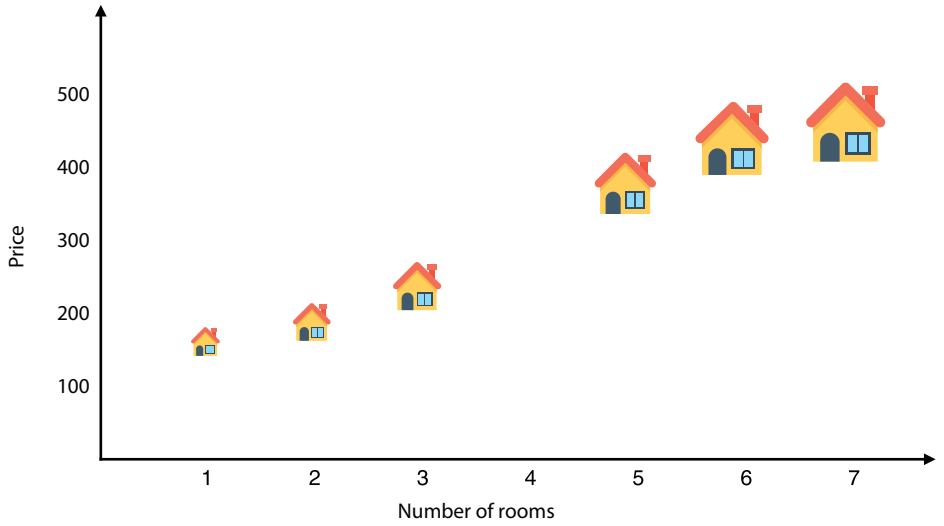


Figure 3.3 Plot of the dataset in table 3.2. The horizontal axis represents the number of rooms, and the vertical axis represents the price of the house.

The formulate step: Formulating a rule that estimates the price of the house

The dataset in table 3.2 is close enough to the one in table 3.1, so for now, we can feel safe using the same formula for the price. The only difference is that now the prices are not exactly what the formula says, and we have a small error. We can write the equation as follows:

$$\text{Price} = 100 + 50(\text{Number of rooms}) + (\text{Small error})$$

If we want to predict prices, we can use this equation. Even though we are not sure we'll get the actual value, we know that we are likely to get close. Now the question is, how did we find this equation? And most important, how does a computer find this equation?

Let's go back to the plot and see what the equation means there. What happens if we look at all the points in which the vertical (y) coordinate is 100 plus 50 times the horizontal (x) coordinate? This set of points forms a line with slope 50 and y-intercept 100. Before we unpack the previous statement, here are the definitions of slope, y-intercept, and the equation of a line. We delve into these in more detail in the "Crash course on slope and y-intercept" section.

slope The slope of a line is a measure of how steep it is. It is calculated by dividing the rise over the run (i.e., how many units it goes up, divided by how many units it goes to the right). This ratio is constant over the whole line. In a machine learning model, this is the weight of the corresponding feature, and it tells us how much we expect the value of the

label to go up, when we increase the value of the feature by one unit. If the line is horizontal, then the slope is zero, and if the line goes down, the slope is negative.

y-intercept The y-intercept of a line is the height at which the line crosses the vertical (y-) axis. In a machine learning model, it is the bias and tells us what the label would be in a data point where all the features are precisely zero.

linear equation This is the equation of a line. It is given by two parameters: the slope and the y-intercept. If the slope is m and the y-intercept is b, then the equation of the line is $y = mx + b$, and the line is formed by all the points (x,y) that satisfy the equation. In a machine learning model, x is the value of the feature and y is the prediction for the label. The weight and bias of the model are m and b, respectively.

We can now analyze the equation. When we say that the slope of the line is 50—this means that each time we add one room to the house, we estimate that the price of the house will go up by \$50. When we say that the y-intercept of the line is 100, this means that the estimate for the price of a (hypothetical) house with zero rooms would be the base price of \$100. This line is drawn in figure 3.4.

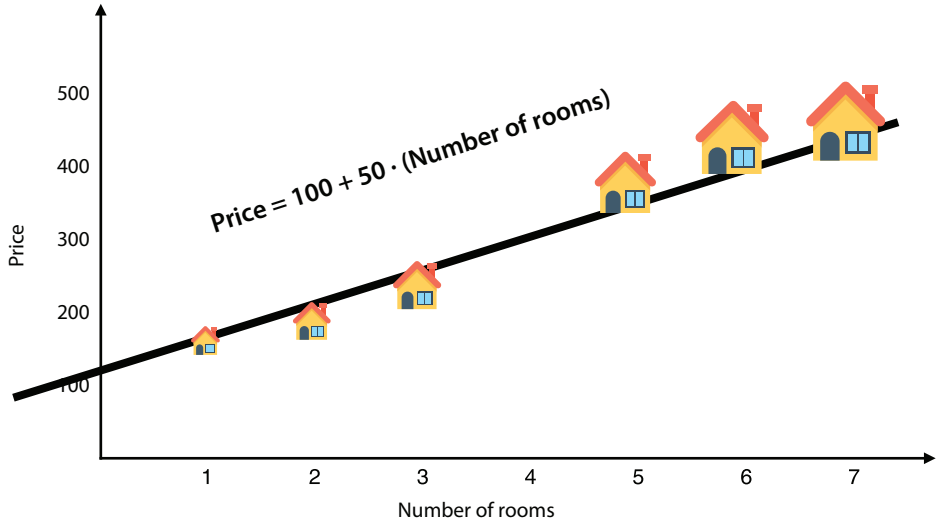


Figure 3.4 The model we formulate is the line that goes as close as possible to all the houses.

Now, of all the possible lines (each with its own equation), why did we pick this one in particular? Because that one passes close to the points. There may be a better one, but at least we know this one is good, as opposed to one that goes nowhere near the points. Now we are back to the original problem, where we have a set of houses, and we want to build a road as close as possible to them.

How do we find this line? We'll look at this later in the chapter. But for now, let's say that we have a crystal ball that, given a bunch of points, finds the line that passes the closest to them.

The predict step: What do we do when a new house comes on the market?

Now, on to using our model to predict the price of the house with four rooms. For this, we plug the number four as the feature in our formula to get the following:

$$\text{Price} = 100 + 50 \cdot 4 = 300$$

Therefore, our model predicts that the house costs $300. This can also be seen graphically by using the line, as illustrated in figure 3.5.

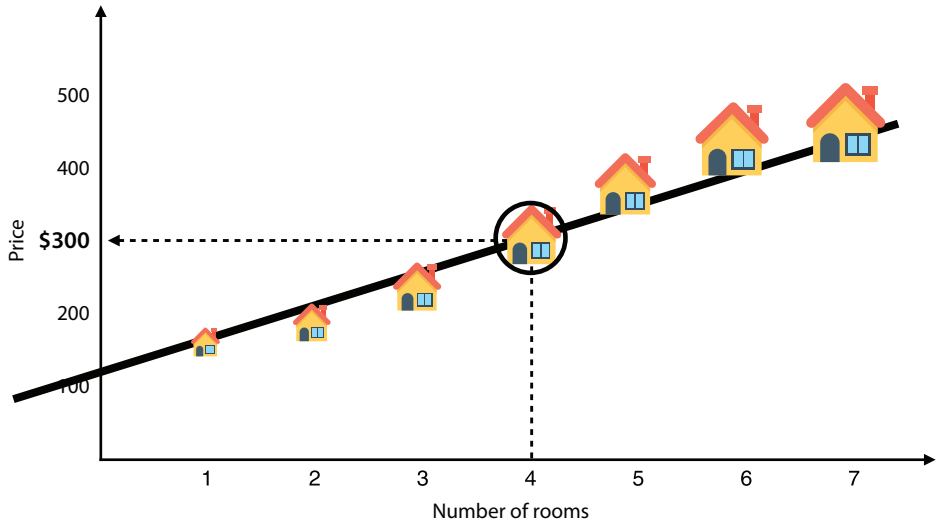


Figure 3.5 Our task is now to predict the price of the house with four rooms. Using the model (line), we deduce that the predicted price of this house is $300.

What if we have more variables? Multivariate linear regression

In the previous sections we learned about a model that predicts the price of a house based on one feature—the number of rooms. We may imagine many other features that could help us predict the price of a house, such as the size, the quality of the schools in the neighborhood, and the age of the house. Can our linear regression model accommodate these other variables? Absolutely. When the only feature is the number of rooms, our model predicts the price as the sum of the feature times their corresponding weight, plus a bias. If we have more features, all we need to do is multiply them by their corresponding weights and add them to the predicted price. Therefore, a model for the price of a house could look like this:

$$\text{Price} = 30(\text{number of rooms}) + 1.5(\text{size}) + 10(\text{quality of the schools}) - 2(\text{age of the house}) + 50$$

In this equation, why are all of the weights positive, except for the one corresponding to the age of the house? The reason is the other three features (number of rooms, size, and quality of the

schools) are *positively correlated* to the price of the house. In other words, because houses that are bigger and well located cost more, the higher this feature is, the higher we expect the price of the house to be. However, because we would imagine that older houses tend to be less expensive, the age feature is *negatively correlated* to the price of the house.

What if the weight of a feature is zero? This happens when a feature is irrelevant to the price. For example, imagine a feature that measured the number of neighbors whose last name starts with the letter *A*. This feature is mostly irrelevant to the price of the house, so we would expect that in a reasonable model, the weight corresponding to this feature is either zero or something very close to it.

In a similar way, if a feature has a very high weight (whether negative or positive), we interpret this as the model telling us that that feature is important in determining the price of the house. In the previous model, it seems that the number of rooms is an important feature, because its weight is the largest (in absolute value).

In the section called "Dimensionality reduction simplifies data without losing too much information" in chapter 2, we related the number of columns in a dataset to the dimension in which the dataset lives. Thus, a dataset with two columns can be represented as a set of points in the plane, and a dataset with three columns can be represented as a set of points in three-dimensional space. In such a dataset, a linear regression model corresponds not to a line but to a plane that passes as close as possible to the points. Imagine having many flies flying around in the room in a stationary position, and our task is to try to pass a gigantic cardboard sheet as close as we can to all the flies. This is multivariate linear regression with three variables. The problem becomes hard to visualize for datasets with more columns, but we can always imagine a linear equation with many variables.

In this chapter, we mostly deal with training linear regression models with only one feature, but the procedure is similar with more features. I encourage you to read about it while keeping this fact in the back of your mind, and imagine how you would generalize each of our next statements to a case with several features.

Some questions that arise and some quick answers

OK, your head may be ringing with lots of questions. Let's address some (hopefully all) of them!

1. What happens if the model makes a mistake?
2. How did you come up with the formula that predicts the price? And what would we do if instead of six houses, we had thousands of them?
3. Say we've built this prediction model, and then new houses start appearing in the market. Is there a way to update the model with new information?

This chapter answers all these questions, but here are some quick answers:

1. **What happens if the model makes a mistake?**
   The model is estimating the price of a house, so we expect it to make a small mistake pretty much all the time, because it is very hard to hit the exact price. The training process consists of finding the model that makes the smallest errors at our points.

2. **How did you come up with the formula that predicts the price? And what would we do if instead of six houses, we had thousands of them?**
   Yes, this is the main question we address in this chapter! When we have six houses, the problem of drawing a line that goes close to them is simple, but if we have thousands of houses, this task gets hard. What we do in this chapter is devise an algorithm, or a procedure, for the computer to find a good line.
3. **Say we've built this prediction model, and then new houses start appearing in the market. Is there a way to update the model with new information?**
   Absolutely! We will build the model in a way that it can be easily updated if new data appears. This is always something to look for in machine learning. If we've built our model in such a way that we need to recalculate the entire model every time new data comes in, it won't be very useful.

How to get the computer to draw this line: The linear regression algorithm

Now we get to the main question of this chapter: how do we get a computer to draw a line that passes really close to the points? The way we do this is the same way we do many things in machine learning: step by step. Start with a random line, and figure out a way to improve this line a *little bit* by moving it closer to the points. Repeat this process many times, and voilà, we have the desired line. This process is called the linear regression algorithm.

The procedure may sound silly, but it works really well. Start with a random line. Pick a random point in the dataset, and move the line slightly closer to that one point. Repeat this process many times, always picking a random point in the dataset. The pseudocode for the linear regression algorithm, viewed in this geometric fashion, follows. The illustration is shown in figure 3.6.

Pseudocode for the linear regression algorithm (geometric)

Inputs: A dataset of points in the plane

Outputs: A line that passes close to the points

Procedure:

- Pick a random line.
- Repeat many times:
  - Pick a random data point.
  - Move the line a little closer to that point.
- **Return** the line you've obtained.

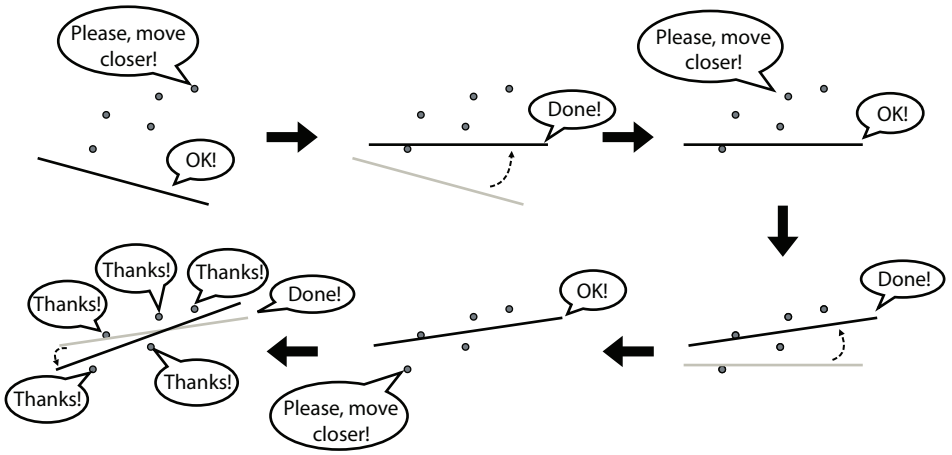


Figure 3.6 An illustration of the linear regression algorithm. We start at the top left with a random line and end in the bottom left with a line that fits the dataset well. At each stage, two things happen: (1) we pick a random point, and (2) the point asks the line to move closer to it. After many iterations, the line will be in a good position. This figure has only three iterations for illustrative purposes, but in real life, many more iterations are needed.

That was the high-level view. To study the process more in detail, we need to delve into the mathematical details. Let's begin by defining some variables.

- p: The price of a house in the dataset
- $\hat{p}$: The predicted price of a house
- r: The number of rooms
- m: The price per room
- b: The base price for a house

Why the hat over the predicted price, $\hat{p}$? Throughout this book, the hat indicates that this is the variable that our model is predicting. In that way, we can tell the actual price of a house in the dataset from its predicted price.

Thus, the equation of a linear regression model that predicts the price as the base price plus the price per room times the number of rooms is

$$\hat{p} = mr + b.$$

This is a formulaic way of saying

$$\text{Predicted price} = (\text{Price per room})(\text{Number of rooms}) + \text{Base price of the house}.$$

To get an idea of the linear regression algorithm, imagine that we have a model in which the price per room is \$40 and the base price of the house is \$50. This model predicts the price of a house using the following formula:

$$\hat{p} = 40 \cdot r + 50$$

To illustrate the linear regression algorithm, imagine that in our dataset we have a house with two rooms that costs \$150. This model predicts that the price of the house is $50 + 40 \cdot 2 = 130$. That is not a bad prediction, but it is less than the price of the house. How can we improve the model? It seems like the model's mistake is thinking that the house is too cheap. Maybe the model has a low base price, or maybe it has a low price per room, or maybe both. If we increase both by a small amount, we may get a better estimate. Let's increase the price per room by \$0.50 and the base price by \$1. (I picked these numbers randomly.) The new equation follows:

$$\hat{p} = 40.5 \cdot r + 51$$

The new predicted price for the house is $40.5 \cdot r + 51 = 132$. Because \$132 is closer to \$150, our new model makes a better prediction for this house. Therefore, it is a better model for that data point. We don't know if it is a better model for the other data points, but let's not worry about that for now. The idea of the linear regression algorithm is to repeat the previous process many times. The pseudocode of the linear regression algorithm follows:

Pseudocode for the linear regression algorithm

Inputs: A dataset of points

Outputs: A linear regression model that fits that dataset

Procedure:

- Pick a model with random weights and a random bias.
- Repeat many times:
  - Pick a random data point.
  - Slightly adjust the weights and bias to improve the prediction for that particular data point.
- **Return** the model you've obtained.

You may have a few questions, such as the following:

- By how much should I adjust the weights?
- How many times should I repeat the algorithm? In other words, how do I know when I'm done?
- How do I know that this algorithm works?

We answer all of these questions in this chapter. In the sections "The square trick" and "The absolute trick," we learn some interesting tricks to find good values to adjust the weights. In the

sections "The absolute error" and "The square error," we see the error function, which will help us decide when to stop the algorithm. And finally, in the section "Gradient descent," we cover a powerful method called gradient descent, which justifies why this algorithm works. But first, let's start by moving lines in the plane.

Crash course on slope and *y*-intercept

In the section "The formulate step," we talked about the equation of a line. In this section, we learn how to manipulate this equation to move our line. Recall that the equation of a line has the following two components:

- The slope
- The *y*-intercept

The slope tells us how steep the line is, and the *y*-intercept tells us where the line is located. The slope is defined as the rise divided by the run, and the *y*-intercept tells us where the line crosses the *y*-axis (the vertical axis). In figure 3.7, we can see both in an example. This line has the following equation:

$$y = 0.5x + 2$$

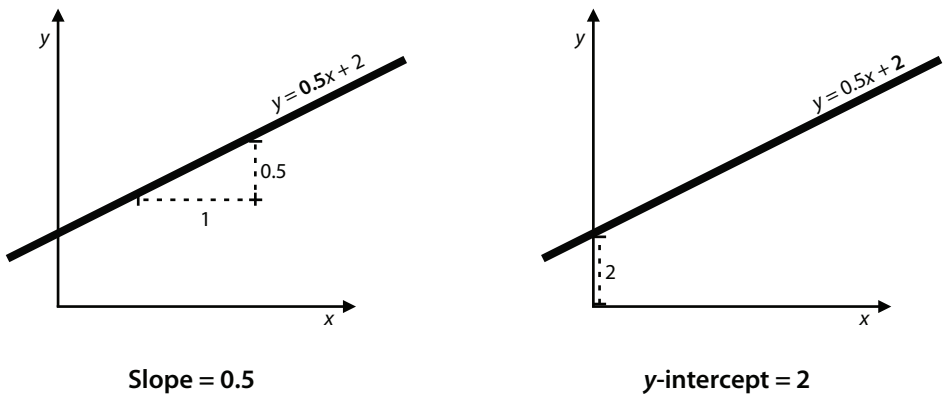


Figure 3.7 The line with equation $y = 0.5x + 2$ has slope 0.5 (left) and *y*-intercept 2 (right).

What does this equation mean? It means that the slope is 0.5, and the *y*-intercept is 2.

When we say that the slope is 0.5, it means that when we walk along this line, for every unit that we move to the right, we are moving 0.5 units up. The slope can be zero if we don't move up at all or negative if we move down. A vertical line has an undefined slope, but luckily, these don't tend to show up in linear regression. Many lines can have the same slope. If I draw any line parallel to the line in figure 3.7, this line will also rise 0.5 units for every unit it moves to the right. This is where the *y*-intercept comes in. The *y*-intercept tells us where the line cuts the *y*-axis. This line cuts the *x*-axis at height 2, and that is the *y*-intercept.

In other words, the slope of the line tells us the *direction* in which the line is pointing, and the *y*-intercept tells us the *location* of the line. Notice that by specifying the slope and the *y*-intercept, the line is completely specified. In figure 3.8, we can see different lines with the same *y*-intercept, and different lines with the same slope.

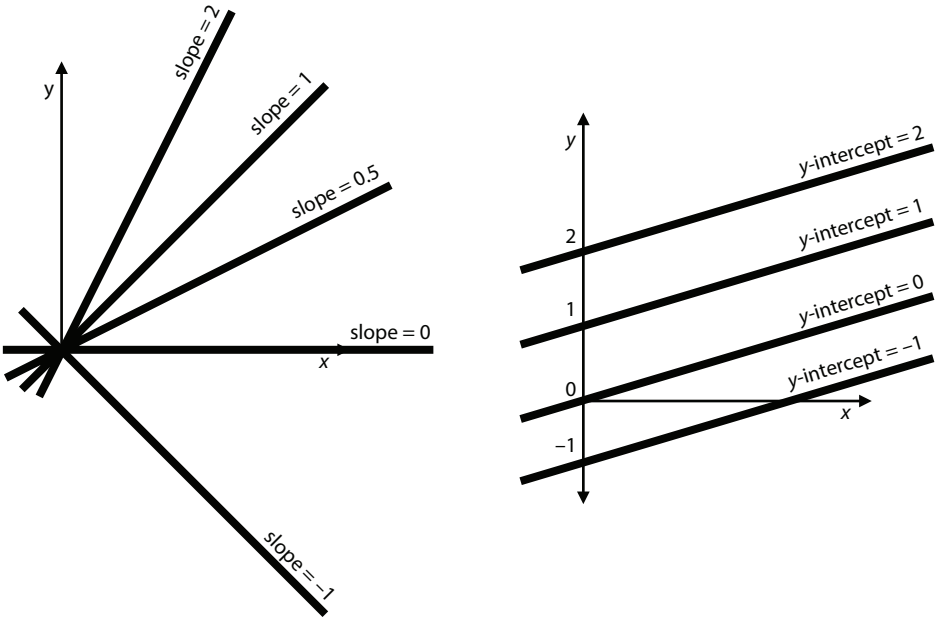


Figure 3.8 Some examples of slope and ***y***-intercept. On the left, we see several lines with the same intercept and different slopes. Notice that the higher the slope, the steeper the line. On the right, we see several lines with the same slope and different ***y***-intercepts. Notice that the higher the ***y***-intercept, the higher the line is located.

In our current housing example, the slope represents the price per room, and the *y*-intercept represents the base price of a house. Let's keep this in mind, and, as we manipulate the lines, think of what this is doing to our housing price model.

From the definitions of slope and *y*-intercept, we can deduce the following:

Changing the slope:

- If we increase the slope of a line, the line will rotate counterclockwise.
- If we decrease the slope of a line, the line will rotate clockwise.

These rotations are on the pivot shown in figure 3.9, namely, the point of intersection of the line and the *y*-axis.

Changing the *y*-intercept:

- If we increase the *y*-intercept of a line, the line is translated upward.
- If we decrease the *y*-intercept of a line, the line is translated downward.

Figure 3.9 illustrates these rotations and translations, which will come in handy when we want to adjust our linear regression models.

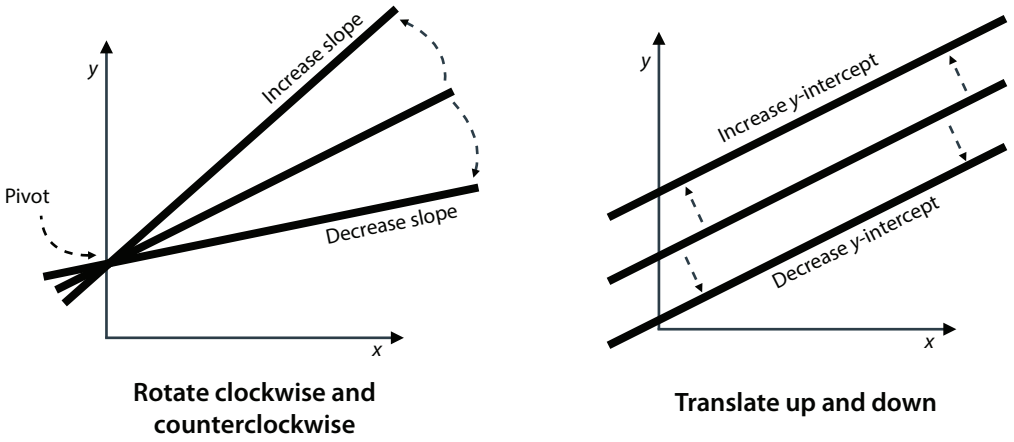


Figure 3.9 Left: Increasing the slope rotates the line counterclockwise, whereas decreasing the slope rotates it clockwise. Right: Increasing the ***y***-intercept translates the line upward, whereas decreasing the ***y***-intercept translates it downward.

As explained earlier, in general, the equation of a line is written as $y = mx + b$, where x and y correspond to the horizontal and vertical coordinates, m corresponds to the slope, and b to the y-intercept. Throughout this chapter, to match the notation, we'll write the equation as $\hat{p} = mr + b$, where $\hat{p}$ corresponds to the predicted price, r to the number of rooms, m (the slope) to the price per room, and b (the y-intercept) to the base price of the house.

A simple trick to move a line closer to a set of points, one point at a time

Recall that the linear regression algorithm consists of repeating a step in which we move a line closer to a point. We can do this using rotations and translations. In this section, we learn a trick called the *simple trick*, which consists of slightly rotating and translating the line in the direction of the point to move it closer (figure 3.10).

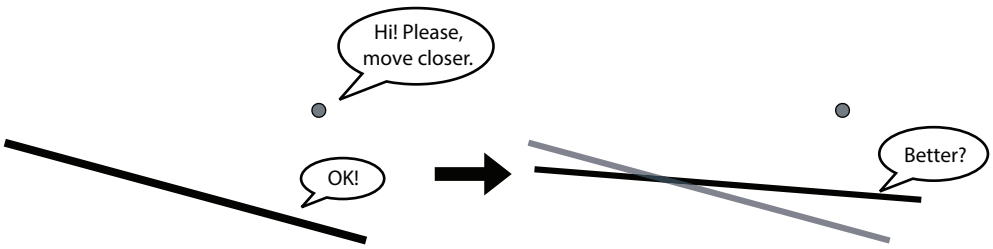


Figure 3.10 Our goal is to rotate and translate the line by a small amount to get closer to the point.

The trick to move the line correctly toward a point is to identify where the point is with respect to the line. If the point is above the line, we need to translate the line up, and if it is below, we need to translate it down. Rotation is a bit harder, but because the pivot is the point of intersection of the line and the *y*-axis, we can see that if the point is above the line and to the right of the *y*-axis, or below the line and to the left of the *y*-axis, we need to rotate the line counterclockwise. In the other two scenarios, we need to rotate the line clockwise. These are summarized in the following four cases, which are illustrated in figure 3.11:

Case 1: If the point is above the line and to the right of the *y*-axis, we rotate the line counterclockwise and translate it upward.

Case 2: If the point is above the line and to the left of the *y*-axis, we rotate the line clockwise and translate it upward.

Case 3: If the point is below the line and to the right of the *y*-axis, we rotate the line clockwise and translate it downward.

Case 4: If the point is below the line and to the left of the *y*-axis, we rotate the line counterclockwise and translate it downward.

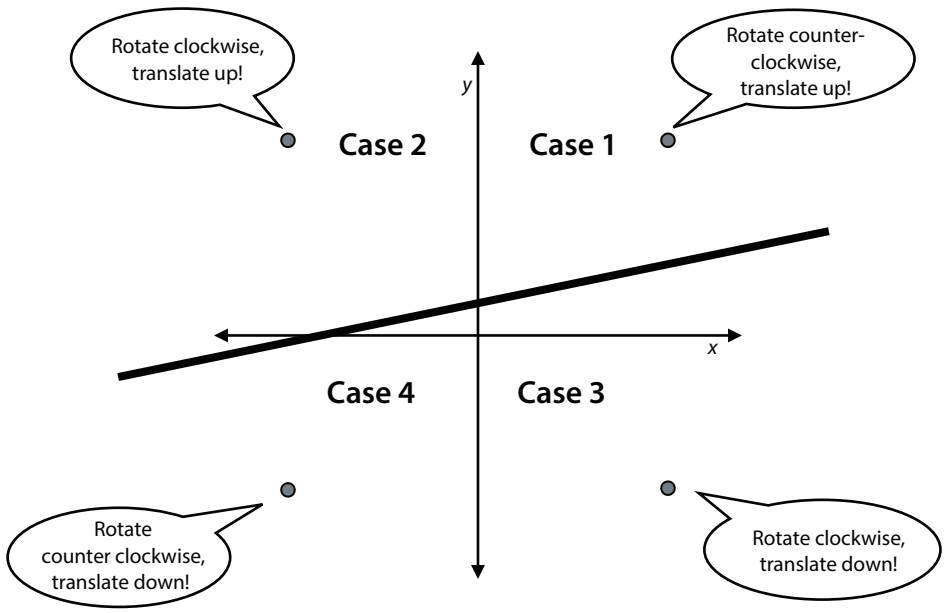


Figure 3.11 The four cases. In each of these we must rotate the line and translate it in a different way to move the line closer to the corresponding point.

Now that we have these four cases, we can write the pseudocode for the simple trick. But first, let's clarify some notation. In this section we've been talking about lines with equation $y = mx + b$,

where m is the slope and b is the y-intercept. In the housing example, we used the following similar notation:

- The point with coordinates (r, p) corresponds to a house with r rooms and price p.
- The slope m corresponds to the price per room.
- The y-intercept b corresponds to the base price of the house.
- The prediction $\hat{p} = mr + b$ corresponds to the predicted price of the house.

Pseudocode for the simple trick

Inputs:

- A line with slope m, y-intercept b, and equation $\hat{p} = mr + b$
- A point with coordinates (r, p)

Output:

- A line with equation $\hat{p} = m'r + b$ that is closer to the point

Procedure:

Pick two very small random numbers, and call them η_1 and η_2 (the Greek letter *eta*).

Case 1: If the point is above the line and to the right of the y-axis, we rotate the line counterclockwise and translate it upward:

- Add η_1 to the slope m. Obtain $m' = m + \eta_1$.
- Add η_2 to the y-intercept b. Obtain $b' = b + \eta_2$.

Case 2: If the point is above the line and to the left of the y-axis, we rotate the line clockwise and translate it upward:

- Subtract η_1 from the slope m. Obtain $m' = m - \eta_1$.
- Add η_2 to the y-intercept b. Obtain $b' = b + \eta_2$.

Case 3: If the point is below the line and to the right of the y-axis, we rotate the line clockwise and translate it downward:

- Subtract η_1 from the slope m. Obtain $m' = m - \eta_1$.
- Subtract η_2 from the y-intercept b. Obtain $b' = b - \eta_2$.

Case 4: If the point is below the line and to the left of the y-axis, we rotate the line counterclockwise and translate it downward:

- Add η_1 to the slope m. Obtain $m' = m + \eta_1$.
- Subtract η_2 from the y-intercept b. Obtain $b' = b - \eta_2$.

Return: The line with equation $\hat{p} = m'r + b'$.

Note that for our example, adding or subtracting a small number to the slope means increasing or decreasing the price per room. Similarly, adding or subtracting a small number to the y-intercept means increasing or decreasing the base price of the house. Furthermore, because the x-coordinate is the number of rooms, this number is never negative. Thus, only cases 1 and 3 matter in our example, which means we can summarize the simple trick in colloquial language as follows:

Simple trick

- If the model gave us a price for the house that is lower than the actual price, add a small random amount to the price per room and to the base price of the house.
- If the model gave us a price for the house that is higher than the actual price, subtract a small random amount from the price per room and the base price of the house.

This trick achieves some success in practice, but it's far from being the best way to move lines. Some questions may arise, such as the following:

- Can we pick better values for η_1 and η_2?
- Can we crunch the four cases into two, or perhaps one?

The answer to both questions is yes, and we'll see how in the following two sections.

The square trick: A much more clever way of moving our line closer to one of the points

In this section, I show you an effective way to move a line closer to a point. I call this the *square trick*. Recall that the simple trick consisted of four cases that were based on the position of the point with respect to the line. The square trick will bring these four cases down to one by finding values with the correct signs (+ or –) to add to the slope and the y-intercept for the line to always move closer to the point.

We start with the y-intercept. Notice the following two observations:

- **Observation 1:** In the simple trick, when the point is above the line, we add a small amount to the y-intercept. When it is below the line, we subtract a small amount.
- **Observation 2:** If a point is above the line, the value $p - \hat{p}$ (the difference between the price and the predicted price) is positive. If it is below the line, this value is negative. This observation is illustrated in figure 3.12.

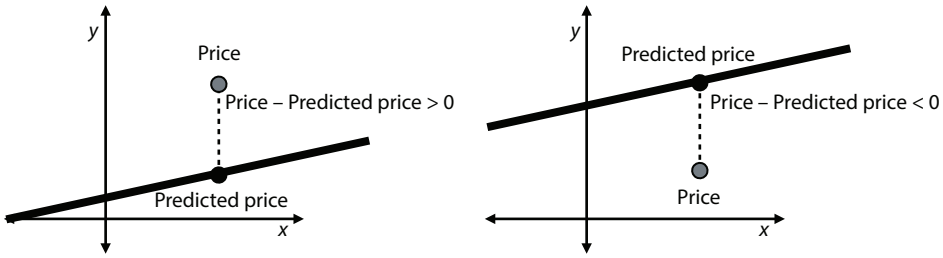


Figure 3.12 Left: When the point is above the line, the price is larger than the predicted price, so the difference is positive. Right: When the point is below the line, the price is smaller than the predicted price, so the difference is negative.

Putting together observation 1 and observation 2, we conclude that if we add the difference $p - \hat{p}$ to the y-intercept, the line will always move toward the point, because this value is positive when the point is above the line and negative when the point is below the line. However, in machine learning, we always want to take small steps. To help us with this, we introduce an important concept in machine learning: the learning rate.

learning rate A very small number that we pick before training our model. This number helps us make sure our model changes in very small amounts by training. In this book, the learning rate will be denoted by η, the Greek letter *eta*.

Because the learning rate is small, so is the value $\eta(p - \hat{p})$. This is the value we add to the y-intercept to move the line in the direction of the point.

The value we need to add to the slope is similar, yet a bit more complicated. Notice the following two observations:

- **Observation 3:** In the simple trick, when the point is in scenario 1 or 4 (above the line and to the right of the vertical axis, or below the line and to the left of the vertical axis), we rotate the line counterclockwise. Otherwise (scenario 2 or 3), we rotate it clockwise.
- **Observation 4:** If a point (r, p) is to the right of the vertical axis, then r is positive. If the point is to the left of the vertical axis, then r is negative. This observation is illustrated in figure 3.13. Notice that in this example, r will never be negative, because it is the number of rooms. However, in a general example, a feature could be negative.

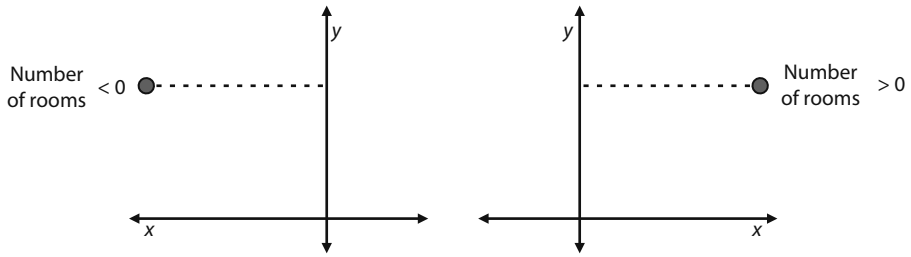


Figure 3.13 Left: When the point is to the left of the ***y***-axis, the number of rooms is negative. Right: When the point is to the right of the ***y***-axis, the number of rooms is positive.

Consider the value $r(p - \hat{p})$. This value is positive when both r and $p - \hat{p}$ are both positive or both negative. This is precisely what occurs in scenarios 1 and 4. Similarly, $r(p - \hat{p})$ is negative in scenarios 2 and 3. Therefore, due to observation 4, this is the quantity that we need to add to the slope. We want this value to be small, so again, we multiply it by the learning rate and conclude that adding $\eta r(p - \hat{p})$ to the slope will always move the line in the direction of the point.

We can now write the pseudocode for the square trick as follows:

Pseudocode for the square trick

Inputs:

- A line with slope m, y-intercept b, and equation $\hat{p} = mr + b$
- A point with coordinates (r, p)
- A small positive value η (the learning rate)

Output:

- A line with equation $\hat{p} = m'r + b'$ that is closer to the point

Procedure:

- Add $\eta r(p - \hat{p})$ to the slope m. Obtain $m' = m + \eta r(p - \hat{p})$ (this rotates the line).
- Add $\eta(p - \hat{p})$ to the y-intercept b. Obtain $b' = b + \eta(p - \hat{p})$ (this translates the line).

Return: The line with equation $\hat{p} = m'r + b'$

We are now ready to code this algorithm in Python! The code for this section follows:

- **Notebook:** Coding_linear_regression.ipynb
  - https://github.com/luisguiserrano/manning/blob/master/Chapter_3_Linear_Regression/Coding_linear_regression.ipynb

And here is code for the square trick:

```
def square_trick(base_price, price_per_room, num_rooms, price, learning_rate):
    predicted_price = base_price + price_per_room*num_rooms
    base_price += learning_rate*(price-predicted_price)
    price_per_room += learning_rate*num_rooms*(price-predicted_price)
    return price_per_room, base_price
```

Calculates the prediction

Translates the line

Rotates the line

The absolute trick: Another useful trick to move the line closer to the points

The square trick is effective, but another useful trick, which we call the *absolute trick*, is an intermediate between the simple and the square tricks. In the square trick, we used the two quantities, $p - \hat{p}$ (price – predicted price) and r (number of rooms), to help us bring the four cases down to one. In the absolute trick, we use only r to help us bring the four cases down to two. In other words, here is the absolute trick:

Pseudocode for the absolute trick

Inputs:

- A line with slope m, y-intercept b, and equation $\hat{p} = mr + b$
- A point with coordinates (r, p)
- A small positive value η (the learning rate)

Output:

- A line with equation $\hat{p} = m'r + b'$ that is closer to the point

Procedure:

Case 1: If the point is above the line (i.e., if $p > \hat{p}$):

- Add ηr to the slope m. Obtain $m' = m + \eta r$ (this rotates the line counterclockwise if the point is to the right of the y-axis, and clockwise if it is to the left of the y-axis).
- Add η to the y-intercept b. Obtain $b' = b + \eta$ (this translates the line up).

Case 2: If the point is below the line (i.e., if $p < \hat{p}$):

- Subtract ηr from the slope m. Obtain $m' = m - \eta r$ (this rotates the line clockwise if the point is to the right of the y-axis, and counterclockwise if it is to the left of the y-axis).
- Subtract η from the y-intercept b. Obtain $b' = b - \eta$ (this translates the line down).

Return: The line with equation $\hat{p} = m'r + b'$

Here is the code for the absolute trick:

```
def absolute_trick(base_price, price_per_room, num_rooms, price,
     learning_rate):
    predicted_price = base_price + price_per_room*num_rooms
    if price > predicted_price:
        price_per_room += learning_rate*num_rooms
        base_price += learning_rate
    else:
        price_per_room -= learning_rate*num_rooms
        base_price -= learning_rate
    return price_per_room, base_price
```

I encourage you to verify that the amount added to each of the weights indeed has the correct sign, as we did with the square trick.

The linear regression algorithm: Repeating the absolute or square trick many times to move the line closer to the points

Now that we've done all the hard work, we are ready to develop the linear regression algorithm! This algorithm takes as input a bunch of points and returns a line that fits them well. This algorithm consists of starting with random values for our slope and our y-intercept and then repeating the procedure of updating them many times using the absolute or the square trick. Here is the pseudocode:

Pseudocode for the linear regression algorithm

Inputs:

- A dataset of houses with number of rooms and prices

Outputs:

- Model weights: price per room and base price

Procedure:

- Start with random values for the slope and y-intercept.
- Repeat many times:
  - Pick a random data point.
  - Update the slope and the y-intercept using the absolute or the square trick.

Each iteration of the loop is called an *epoch*, and we set this number at the beginning of our algorithm. The simple trick was mostly used for illustration, but as was mentioned before, it

doesn't work very well. In real life, we use the absolute or square trick, which works a lot better. In fact, although both are commonly used, the square trick is more popular. Therefore, we'll use that one for our algorithm, but feel free to use the absolute trick if you prefer.

Here is the code for the linear regression algorithm. Note that we have used the Python `random` package to generate random numbers for our initial values (slope and *y*-intercept) and for selecting our points inside the loop:

Imports the random package to generate (pseudo) random numbers

Generates random values for the slope and the y-intercept

Picks a random point on our dataset

Repeats the update step many times

Applies the square trick to move the line closer to our point

```
import random
def linear_regression(features, labels, learning_rate=0.01, epochs = 1000):
    price_per_room = random.random()
    base_price = random.random()
    for epoch in range(epochs):
        i = random.randint(0, len(features)-1)
        num_rooms = features[i]
        price = labels[i]
        price_per_room, base_price = square_trick(base_price,
                                                  price_per_room,
                                                  num_rooms,
                                                  price,
                                                  learning_rate=learning_rate)
    return price_per_room, base_price
```

The next step is to run this algorithm to build a model that fits our dataset.

Loading our data and plotting it

Throughout this chapter, we load and make plots of our data and models using Matplotlib and NumPy, two very useful Python packages. We use NumPy for storing arrays and carrying out mathematical operations, whereas we use Matplotlib for the plots.

The first thing we do is encode the features and labels of the dataset in table 3.2 as NumPy arrays as follows:

```
import numpy as np
features = np.array([1,2,3,5,6,7])
labels = np.array([155, 197, 244, 356, 407, 448])
```

Next we plot the dataset. In the repository, we have some functions for plotting the code in the file utils.py, which you are invited to take a look at. The plot of the dataset is shown in figure 3.14. Notice that the points do appear close to forming a line.

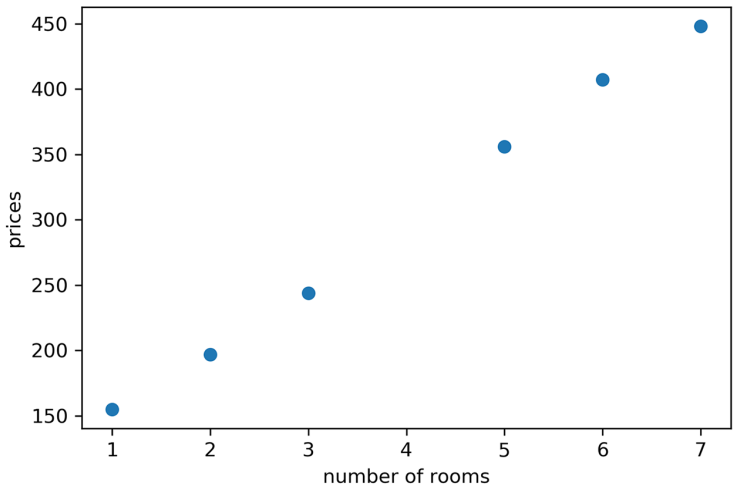


Figure 3.14 The plot of the points in table 3.2

Using the linear regression algorithm in our dataset

Now, let's apply the algorithm to fit a line to these points. The following line of code runs the algorithm with the features, the labels, the learning rate equal to 0.01, and the number of epochs equal to 10,000. The result is the plot shown in figure 3.15.

```
linear_regression(features, labels, learning_rate = 0.01, epochs = 10000)
```

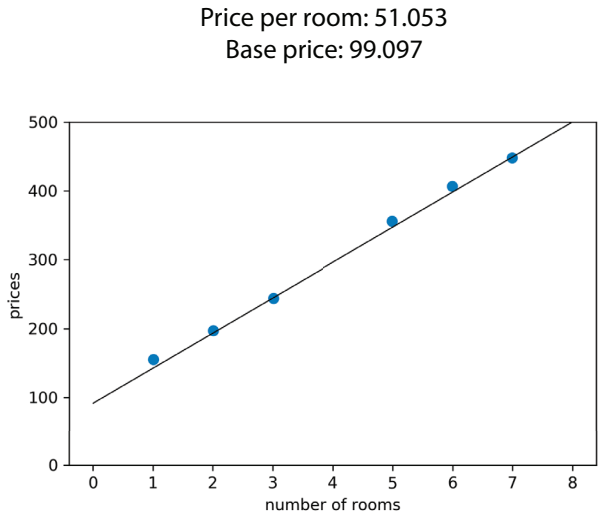


Figure 3.15 The plot of the points in table 3.2 and the line that we obtained with the linear regression algorithm

Figure 3.15 shows the line where the (rounded) price per room is $51.05, and the base price is $99.10. This is not far from the $50 and $100 we eyeballed earlier in the chapter.

To visualize the process, let's look at the progression a bit more. In figure 3.16, you can see a few of the intermediate lines. Notice that the line starts far away from the points. As the algorithm progresses, it moves slowly to fit better and better every time. Notice that at first (in the first 10 epochs), the line moves quickly toward a good solution. After epoch 50, the line is good, but it still doesn't fit the points perfectly. If we let it run for the whole 10,000 epochs, we get a great fit.

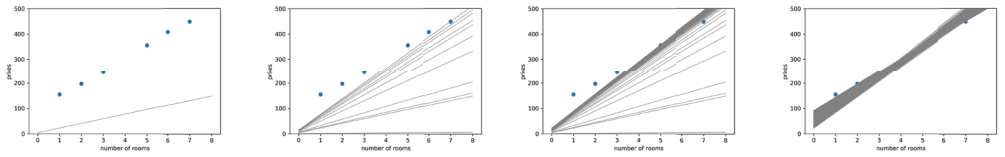


Figure 3.16 Drawing some of the lines in our algorithm, as we approach a better solution. The first graphic shows the starting point. The second graphic shows the first 10 epochs of the linear regression algorithm. Notice how the line is moving closer to fitting the points. The third graphic shows the first 50 epochs. The fourth graphic shows epochs 51 to 10,000 (the last epoch).

Using the model to make predictions

Now that we have a shiny linear regression model, we can use it to make predictions! Recall from the beginning of the chapter that our goal was to predict the price of a house with four rooms. In the previous section, we ran the algorithm and obtained a slope (price per room) of 51.05 and a y-intercept (base price of the house) of 99.10. Thus, the equation follows:

$$\hat{p} = 51.05r + 99.10$$

The prediction the model makes for a house with $r = 4$ rooms is

$$\hat{p} = 51.05 \cdot 4 + 99.10 = 303.30.$$

Note that $303.30 is not far from the $300 we eyeballed at the beginning of the chapter!

The general linear regression algorithm (optional)

This section is optional, because it focuses mostly on the mathematical details of the more abstract algorithm used for a general dataset. However, I encourage you to read it to get used to the notation that is used in most of the machine learning literature.

In the previous sections, we outlined the linear regression algorithm for our dataset with only one feature. But as you can imagine, in real life, we will be working with datasets with many features. For this, we need a general algorithm. The good news is that the general algorithm is not very different

from the specific one that we learned in this chapter. The only difference is that each of the features is updated in the same way that the slope was updated. In the housing example, we had one slope and one y-intercept. In the general case, think of many slopes and still one y-intercept.

The general case will consist of a dataset of m points and n features. Thus, the model has m weights (think of them as the generalization of the slope) and one bias. The notation follows:

- The data points are $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$. Each point is of the form $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, \ldots, x_n^{(i)})$.
- The corresponding labels are $y_1, y_2, \ldots, y_m$.
- The weights of the model are $w_1, w_2, \ldots, w_n$.
- The bias of the model is b.

Pseudocode for the general square trick

Inputs:

- A model with equation $\hat{y} = w_1x_1 + w_2x_{2+} + \cdots + w_nx_n + b$
- A point with coordinates (x, y)
- A small positive value η (the learning rate)

Output:

- A model with equation $\hat{y} = w_1'x_1 + w_2'x_2 + \cdots + w_n'x_n + b'$ that is closer to the point

Procedure:

- Add $\eta(y - \hat{y})$ to the y-intercept b. Obtain $b' = b + \eta(y - \hat{y})$.
- For $i = 1, 2, \ldots, n$:
  - Add $\eta x(y - \hat{y})$ to the weight w_i. Obtain $w_i' = w_i + \eta r(y - \hat{y})$.

Return: The model with equation $\hat{y} = w_1'x_1 + w_2'x_2 + \cdots + w_n'x_n + b'$

The pseudocode of the general linear regression algorithm is the same as the one in the section "The linear regression algorithm," because it consists of iterating over the general square trick, so we'll omit it.

How do we measure our results? The error function

In the previous sections, we developed a direct approach to finding the best line fit. However, many times using a direct approach is difficult to solve problems in machine learning. A more indirect, yet more mechanical, way to do this is using *error functions*. An error function is a metric that tells us how our model is doing. For example, take a look at the two models in figure 3.17. The one on the left is a bad model, whereas the one on the right is a good one. The error

function measures this by assigning a large value to the bad model on the left and a small value to the good model on the right. Error functions are also sometimes called *loss functions* or *cost functions* on the literature. In this book, we call them error functions except in some special cases in which the more commonly used name requires otherwise.

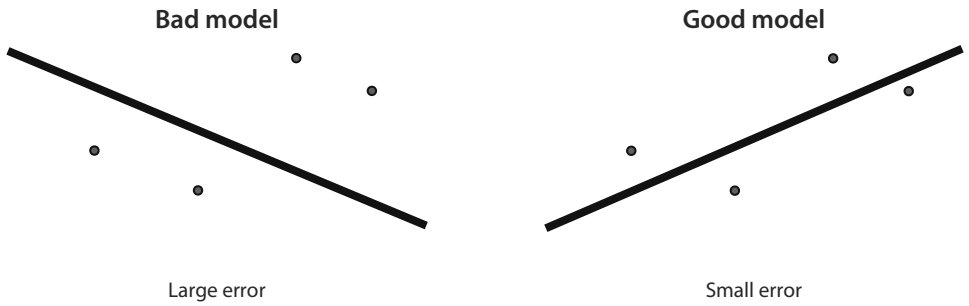


Figure 3.17 Two models, a bad one (on the left) and a good one (on the right). The bad one is assigned a large error, and the good one is assigned a small error.

Now the question is, how do we define a good error function for linear regression models? We have two common ways to do this called the *absolute error* and the *square error*. In short, the absolute error is the sum of vertical distances from the line to the points in the dataset, and the square error is the sum of the squares of these distances.

In the next few sections, we learn about these two error functions in more detail. Then we see how to reduce them using a method called gradient descent. Finally, we plot one of these error functions in our existing example and see how quickly the gradient descent method helps us decrease it.

The absolute error: A metric that tells us how good our model is by adding distances

In this section we look at the absolute error, which is a metric that tells us how good our model is. The absolute error is the sum of the distances between the data points and the line. Why is it called the absolute error? To calculate each of the distances, we take the difference between the label and the predicted label. This difference can be positive or negative depending on whether the point is above or below the line. To turn this difference into a number that is always positive, we take its absolute value.

By definition, a good linear regression model is one where the line is close to the points. What does *close* mean in this case? This is a subjective question, because a line that is close to some of the points may be far from others. In that case, would we rather pick a line that is very close to some of the points and far from some of the others? Or do we try to pick one that is somewhat close to all the points? The absolute error helps us make this decision. The line we pick is the one that minimizes the absolute error, that is, the one for which the sum of vertical distances from each of the points to the line is minimal. In figure 3.18, you can see two lines, and their absolute error is illustrated as the sum of the vertical segments. The line on the left has a large absolute

error, whereas the one on the right has a small absolute error. Thus, between these two, we would pick the one on the right.

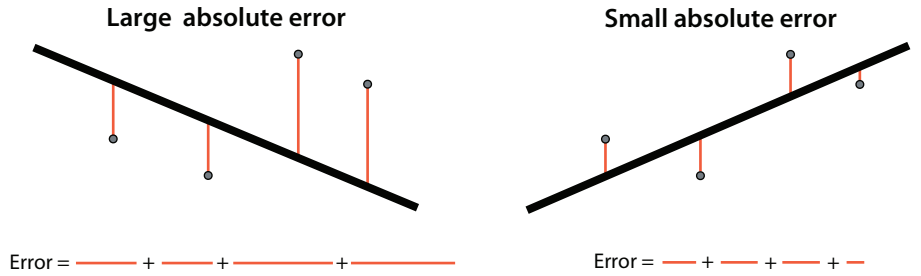


Figure 3.18 The absolute error is the sum of the vertical distances from the points to the line. Note that the absolute error is large for the bad model on the left and small for the good model on the right.

The square error: A metric that tells us how good our model is by adding squares of distances

The square error is very similar to the absolute error, except instead of taking the absolute value of the difference between the label and the predicted label, we take the square. This always turns the number into a positive number, because squaring a number always makes it positive. The process is illustrated in figure 3.19, where the square error is illustrated as the sum of the areas of the squares of the lengths from the points to the line. You can see how the bad model on the left has a large square error, whereas the good model on the right has a small square error.

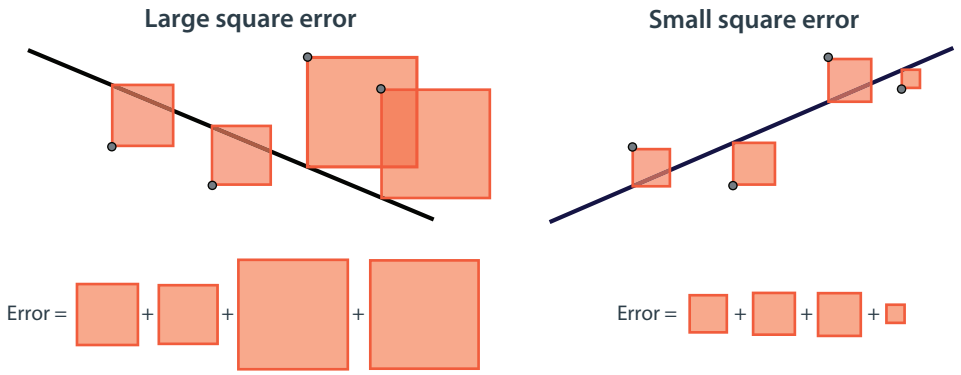


Figure 3.19 The square error is the sum of the squares of the vertical distances from the points to the line. Note that the square error is large for the bad model on the left and small for the good model on the right.

As was mentioned earlier, the square error is used more commonly in practice than the absolute error. Why? A square has a much nicer derivative than an absolute value, which comes in handy during the training process.

Mean absolute and (root) mean square errors are more common in real life

Throughout this chapter we use absolute and square errors for illustration purposes. However, in practice, the *mean absolute error* and the *mean square error* are used much more commonly. These are defined in a similar way, except instead of calculating sums, we calculate averages. Thus, the mean absolute error is the average of the vertical distances from the points to the line, and the mean square error is the average of the squares of these same distances. Why are they more common? Imagine if we'd like to compare the error or a model using two datasets, one with 10 points and one with 1 million points. If the error is a sum of quantities, one for every point, then the error is probably much higher on the dataset of 1 million points, because we are adding many more numbers. If we want to compare them properly, we instead use averages in the calculation of our error to obtain a measure of how far the line is from each point *on average*.

For illustration purposes, another error commonly used is the *root mean square error*, or *RMSE* for short. As the name implies, this is defined as the root of the mean square error. It is used to match the units in the problem and also to give us a better idea of how much error the model makes in a prediction. How so? Imagine the following scenario: if we are trying to predict house prices, then the units of the price and the predicted price are, for example, dollars ($). The units of the square error and the mean square error are dollars squared, which is not a common unit. If we take the square root, then not only do we get the correct unit, but we also get a more accurate idea of roughly by how many dollars the model is off per house. Say, if the root mean square error is $10,000, then we can expect the model to make an error of around $10,000 for any prediction we make.

Gradient descent: How to decrease an error function by slowly descending from a mountain

In this section, I show you how to decrease any of the previous errors using a similar method to the one we would use to slowly descend from a mountain. This process uses derivatives, but here is the great news: you don't need derivatives to understand it. We already used them in the training process in the sections "The square trick" and "The absolute trick" earlier. Every time we "move a small amount in this direction," we are calculating in the background a derivative of the error function and using it to give us a direction in which to move our line. If you love calculus and you want to see the entire derivation of this algorithm using derivatives and gradients, see appendix B.

Let's take a step back and look at linear regression from far away. What is it that we want to do? We want to find the line that best fits our data. We have a metric called the error function, which tells us how far a line is from the data. Thus, if we could just reduce this number as much as possible, we would find the best line fit. This process, common in many areas in mathematics, is called *minimizing functions*, that is, finding the smallest possible value that a function can return. This is where gradient descent comes in: it is a great way to minimize a function.

In this case, the function we are trying to minimize is the error (absolute or square) of our model. A small caveat is that gradient descent doesn't always find the exact minimum value of the function, but it may find something very close to it. The good news is that, in practice, gradient descent is fast and effective at finding points where the function is low.

How does gradient descent work? Gradient descent is the equivalent of descending from a mountain. Let's say we find ourselves on top of a tall mountain called Mount Errorest. We wish

to descend, but it is very foggy, and we can see only about one meter away. What do we do? A good method is to look around ourselves and figure out in what direction we can take one single step, in a way that we descend the most. This process is illustrated in figure 3.20.

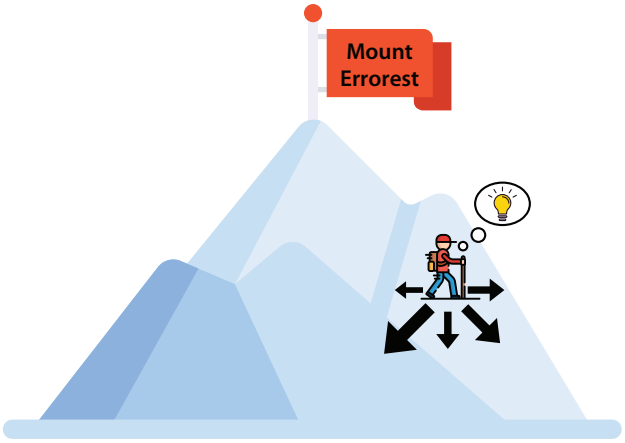


Figure 3.20 We are on top of Mount Errorest and wish to get to the bottom, but we can't see very far. A way to go down is to look at all the directions in which we can take one step and figure out which one helps us descend the most. Then we are one step closer to the bottom.

When we find this direction, we take one small step, and because that step was taken in the direction of greatest descent, then most likely, we have descended a small amount. All we have to do is repeat this process many times until we (hopefully) reach the bottom. This process is illustrated in figure 3.21.

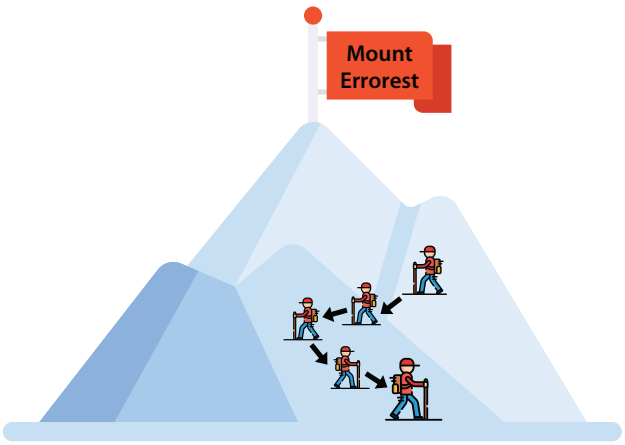


Figure 3.21 The way to descend from the mountain is to take that one small step in the direction that makes us descend the most and to continue doing this for a long time.

Why did I say *hopefully*? Well, this process has many caveats. We could reach the bottom, or we could also reach a valley and then we have nowhere else to move. We won't deal with that now, but we have several techniques to reduce the probability of this happening. In appendix B, "Using gradient descent to train neural networks," some of these techniques are outlined.

A lot of math here that we are sweeping under the rug is explained in more detail in appendix B. But what we did in this chapter was exactly gradient descent. How so? Gradient descent works as follows:

1. Start somewhere on the mountain.
2. Find the best direction to take one small step.
3. Take this small step.
4. Repeat steps 2 and 3 many times.

This may look familiar, because in the section "The linear regression algorithm," after defining the absolute and square tricks, we defined the linear regression algorithm in the following way:

1. Start with any line.
2. Find the best direction to move our line a little bit, using either the absolute or the square trick.
3. Move the line a little bit in this direction.
4. Repeat steps 2 and 3 many times.

The mental picture of this is illustrated in figure 3.22. The only difference is that this error function looks less like a mountain and more like a valley, and our goal is to descend to the lowest point. Each point in this valley corresponds to some model (line) that tries to fit our data. The

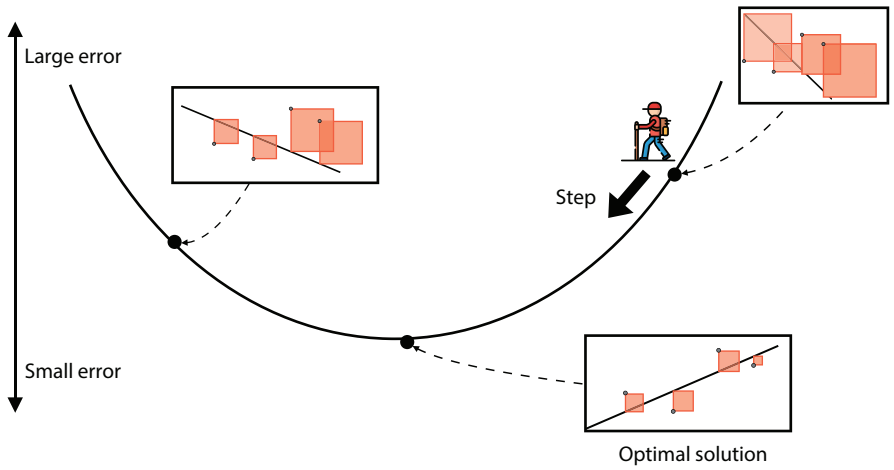


Figure 3.22 Each point on this mountain corresponds to a different model. The points below are good models with a small error, and the points above are bad models with a large error. The goal is to descend from this mountain. The way to descend is by starting somewhere and continuously taking a step that makes us descend. The gradient will help us decide in what direction to take a step that helps us descend the most.

height of the point is the error given by that model. Thus, the bad models are on top, and the good models are on the bottom. We are trying to go as low as possible. Each step takes us from one model to a slightly better model. If we take a step like this many times, we'll eventually get to the best model (or at least, a pretty good one!).

Plotting the error function and knowing when to stop running the algorithm

In this section, we see a plot of the error function for the training that we performed earlier in the section "Using the linear regression algorithm in our dataset." This plot gives us useful information about training this model. In the repository, we have also plotted the root mean square error function (RMSE) defined in the section "Mean absolute and (root) mean square errors ...". The code for calculating the RMSE follows:

```
def rmse(labels, predictions):
    n = len(labels)
    differences = np.subtract(labels, predictions)
    return np.sqrt(1.0/n * (np.dot(differences, differences)))
```

dot product To code the RMSE function, we used the dot product, which is an easy way to write a sum of products of corresponding terms in two vectors. For example, the dot product of the vectors (1,2,3) and (4,5,6) is $1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 32$. If we calculate the dot product of a vector and itself, we obtain the sum of squares of the entries.

The plot of our error is shown in figure 3.23. Note that it quickly dropped after about 1,000 iterations, and it didn't change much after that. This plot gives us useful information: it tells us that for this model, we can run the training algorithm for only 1,000 or 2,000 iterations instead of 10,000 and still get similar results.

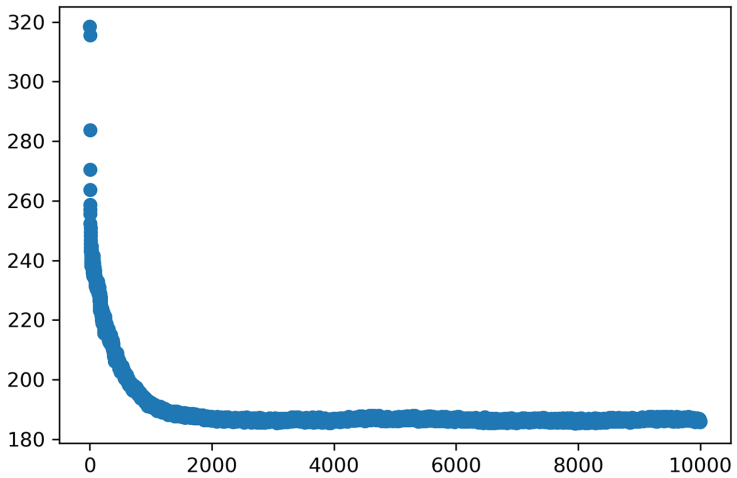


Figure 3.23 The plot of the root mean square error for our running example. Notice how the algorithm succeeded in reducing this error after a little over 1,000 iterations. This means that we don't need to keep running this algorithm for 10,000 iterations, because around 2,000 of them do the job.

In general, the error function gives us good information to decide when to stop running the algorithm. Often, this decision is based on the time and the computational power available to us. However, other useful benchmarks are commonly used in the practice, such as the following:

- When the loss function reaches a certain value that we have predetermined
- When the loss function doesn't decrease by a significant amount during several epochs

Do we train using one point at a time or many? Stochastic and batch gradient descent

In the section "How to get the computer to draw this line," we trained a linear regression model by repeating a step many times. This step consisted of picking one point and moving the line toward that point. In the section "How do we measure our results," we trained a linear regression model by calculating the error (absolute or squared) and decreasing it using gradient descent. However, this error was calculated on the entire dataset, not on one point at a time. Why is this?

The reality is that we can train models by iterating on one point at a time or on the entire dataset. However, when the datasets are very big, both options may be expensive. We can practice a useful method called *mini-batch learning,* which consists of dividing our data into many mini-batches. In each iteration of the linear regression algorithm, we pick one of the mini-batches and proceed to adjust the weights of the model to reduce the error in that mini-batch. The decision of using one point, a mini-batch of points, or the entire dataset on each iteration gives rise to three general types of gradient descent algorithms. When we use one point at a time, it is called *stochastic gradient descent*. When we use a mini-batch, it is called *mini-batch gradient descent*. When we use the entire dataset, it is called *batch gradient descent*. This process is illustrated in more detail in appendix B, "Using gradient descent to train models."

Real-life application: Using Turi Create to predict housing prices in India

In this section, I show you a real-life application. We'll use linear regression to predict housing prices in Hyderabad, India. The dataset we use comes from Kaggle, a popular site for machine learning competitions. The code for this section follows:

- **Notebook:** House_price_predictions.ipynb
  - https://github.com/luisguiserrano/manning/blob/master/Chapter_3_Linear_Regression/House_price_predictions.ipynb
- **Dataset:** Hyderabad.csv

This dataset has 6,207 rows (one per house) and 39 columns (features). As you can imagine, we won't code the algorithm by hand. Instead, we use Turi Create, a popular and useful package in which many machine learning algorithms are implemented. The main object to store data in

Turi Create is the SFrame. We start by downloading the data into an SFrame, using the following command:

```
data = tc.SFrame('Hyderabad.csv')
```

The table is too big, but you can see the first few rows and columns in table 3.3.

Table 3.3 The first five rows and seven columns of the Hyderabad housing prices dataset

Price	Area	No. of Bedrooms	Resale	MaintenanceStaff	Gymnasium	SwimmingPool
30000000	3340	4	0	1	1	1
7888000	1045	2	0	0	1	1
4866000	1179	2	0	0	1	1
8358000	1675	3	0	0	0	0
6845000	1670	3	0	1	1	1

Training a linear regression model in Turi Create takes only one line of code. We use the function `create` from the package `linear_regression`. In this function, we only need to specify the target (label), which is `Price`, as follows:

```
model = tc.linear_regression.create(data, target='Price')
```

It may take a few moments to train, but after it trains, it outputs some information. One of the fields it outputs is the root mean square error. For this model, the RMSE is in the order of 3,000,000. This is a large RMSE, but it doesn't mean the model makes bad predictions. It may mean that the dataset has many outliers. As you can imagine, the price of a house may depend on many other features that are not in the dataset.

We can use the model to predict the price of a house with an area of 1,000, and three bedrooms as follows:

```
house = tc.SFrame({'Area': [1000], 'No. of Bedrooms':[3]})
model.predict(house)
Output: 2594841
```

The model outputs that the price for a house of size 1,000 and three bedrooms is 2,594,841.

We can also train a model using fewer features. The `create` function allows us to input the features we want to use as an array. The following line of code trains a model called `simple_model` that uses the area to predict the price:

```
simple_model = tc.linear_regression.create(data, features=['Area'],
    target='Price')
```

We can explore the weights of this model with the following line of code:

```
simple_model.coefficients
```

The output gives us the following weights:

- Slope: 9664.97
- y-intercept: –6,105,981.01

The intercept is the bias, and the coefficient for area is the slope of the line, when we plot area and price. The plot of the points with the corresponding model is shown in figure 3.24.

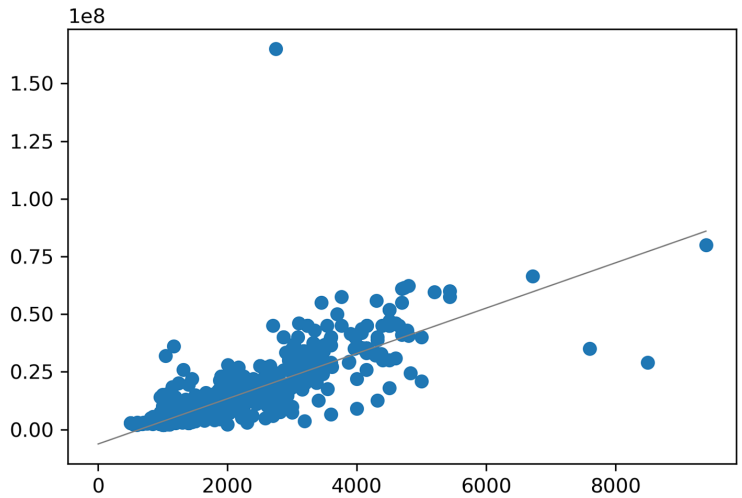


Figure 3.24 The Hyderabad housing prices dataset restricted to area and price. The line is the model we've obtained using only the area feature to predict the price.

We could do a lot more in this dataset, and I invite you to continue exploring. For example, try to explore what features are more important than others by looking at the weights of the model. I encourage you to take a look at the Turi Create documentation (https://apple.github.io/turicreate/docs/api/) for other functions and tricks you can do to improve this model.

What if the data is not in a line? Polynomial regression

In the previous sections, we learned how to find the best line fit for our data, assuming our data closely resembles a line. But what happens if our data doesn't resemble a line? In this section, we learn a powerful extension to linear regression called *polynomial regression*, which helps us deal with cases in which the data is more complex.

A special kind of curved functions: Polynomials

To learn polynomial regression, first we need to learn what polynomials are. *Polynomials* are a class of functions that are helpful when modeling nonlinear data.

We've already seen polynomials, because every line is a polynomial of degree 1. Parabolas are examples of polynomials of degree 2. Formally, a polynomial is a function in one variable that can be expressed as a sum of multiples of powers of this variable. The powers of a variable x are $1, x, x^2, x^3, \ldots$. Note that the two first are $x^0 = 1$ and $x^1 = x$. Therefore, the following are examples of polynomials:

- $y = 4$
- $y = 3x + 2$
- $y = x^2 - 2x + 5$
- $y = 2x^3 + 8x^2 - 40$

We define the *degree* of the polynomial as the exponent of the highest power in the expression of the polynomial. For example, the polynomial $y = 2x^3 + 8x^2 - 40$ has degree 3, because 3 is the highest exponent that the variable x is raised to. Notice that in the example, the polynomials have degree 0, 1, 2, and 3. A polynomial of degree 0 is always a constant, and a polynomial of degree 1 is a linear equation like the ones we've seen previously in this chapter.

The graph of a polynomial looks a lot like a curve that oscillates several times. The number of times it oscillates is related to the degree of the polynomial. If a polynomial has degree d, then the graph of that polynomial is a curve that oscillates at most $d - 1$ times (for $d > 1$). In figure 3.25 we can see the plots of some examples of polynomials.

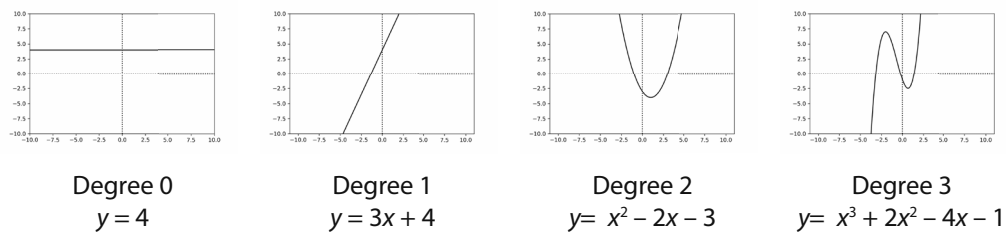


Figure 3.25 Polynomials are functions that help us model our data better. Here are the plots of four polynomials of degrees 0 to 3. Note that the polynomial of degree 0 is a horizontal line, the polynomial of degree 1 is any line, the polynomial of degree 2 is a parabola, and the polynomial of degree 3 is a curve that oscillates twice.

From the plot, notice that polynomials of degree 0 are flat lines. Polynomials of degree 1 are lines with slopes different from 0. Polynomials of degree 2 are quadratics (parabolas). Polynomials of degree 3 look like a curve that oscillates twice (although they could potentially oscillate fewer times). How would the plot of a polynomial of degree 100 look like? For example, the plot of $y = x^{100} - 8x^{62} + 73x^{27} - 4x + 38$? We'd have to plot it to find out, but for sure, we know that it is a curve that oscillates at most 99 times.

Nonlinear data? No problem: Let's try to fit a polynomial curve to it

In this section, we see what happens if our data is not linear (i.e., does not look like it forms a line), and we want to fit a polynomial curve to it. Let's say that our data looks like the left side of figure 3.26. No matter how much we try, we can't really find a good line that fits this data. No problem! If we decide to fit a polynomial of degree 3 (also called a cubic), then we get the curve shown at the right of figure 3.26, which is a much better fit to the data.

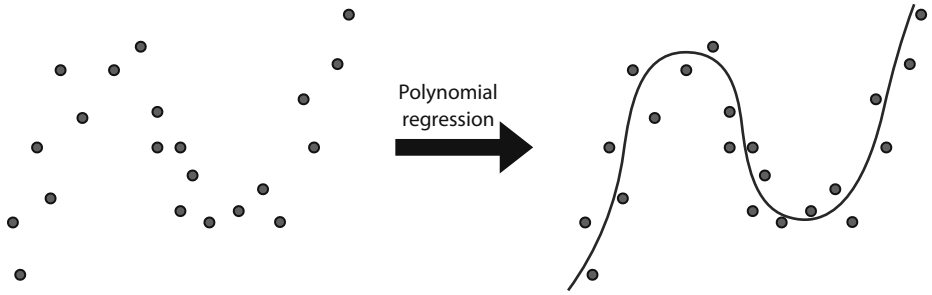


Figure 3.26 Polynomial regression is useful when it comes to modeling nonlinear data. If our data looks like the left part of the figure, it will be hard to find a line that fits it well. However, a curve will fit the data well, as you can see in the right part of the figure. Polynomial regression helps us find this curve.

The process to train a polynomial regression model is similar to the process of training a linear regression model. The only difference is that we need to add more columns to our dataset before we apply linear regression. For example, if we decide to fit a polynomial of degree 3 to the data in figure 3.26, we need to add two columns: one corresponding to the square of the feature and one corresponding to the cube of the feature. If you'd like to study this in more detail, please check out the section "Polynomial regression, testing, and regularization with Turi Create" in chapter 4, in which we learn an example of polynomial regression in a parabolic dataset.

A small caveat with training a polynomial regression model is that we must decide the degree of the polynomial before the training process. How do we decide on this degree? Do we want a line (degree 1), a parabola (degree 2), a cubic (degree 3), or some curve of degree 50? This question is important, and we deal with it in chapter 4, when we learn overfitting, underfitting, and regularization!

Parameters and hyperparameters

Parameters and hyperparameters are some of the most important concepts in machine learning, and in this section, we learn what they are and how to tell them apart.

As we saw in this chapter, regression models are defined by their weights and bias—the *parameters* of the model. However, we can twist many other knobs before training a model, such as the learning rate, the number of epochs, the degree (if considering a polynomial regression model), and many others. These are called *hyperparameters*.

Each machine learning model that we learn in this book has some well-defined parameters and hyperparameters. They tend to be easily confused, so the rule of thumb to tell them apart follows:

- Any quantity that you set *before* the training process is a hyperparameter.
- Any quantity that the model creates or modifies *during* the training process is a parameter.

Applications of regression

The impact of machine learning is measured not only by the power of its algorithms but also by the breadth of useful applications it has. In this section, we see some applications of linear regression in real life. In each of the examples, we outline the problem, learn some features to solve it, and then let linear regression do its magic.

Recommendation systems

Machine learning is used widely to generate good recommendations in some of the most well-known apps, including YouTube, Netflix, Facebook, Spotify, and Amazon. Regression plays a key part in most of these recommender systems. Because regression predicts a quantity, all we have to do to generate good recommendations is figure out what quantity is the best at indicating user interaction or user satisfaction. Following are some more specific examples of this.

Video and music recommendations

One of the ways used to generate video and music recommendations is to predict the amount of time a user will watch a video or listen to a song. For this, we can create a linear regression model where the labels on the data are the amount of minutes that each song is watched by each user. The features can be demographics on the user, such as their age, location, and occupation, but they can also be behavioral, such as other videos or songs they have clicked on or interacted with.

Product recommendations

Stores and ecommerce websites also use linear regression to predict their sales. One way to do this is to predict how much a customer will spend in the store. We can do this using linear regression. The label to predict can be the amount the user spent, and the features can be demographic and behavioral, in a similar way to the video and music recommendations.

Health care

Regression has numerous applications in health care. Depending on what problem we want to solve, predicting the right label is the key. Here are a couple of examples:

- Predicting the life span of a patient, based on their current health conditions
- Predicting the length of a hospital stay, based on current symptoms

Summary

- Regression is an important part of machine learning. It consists of training an algorithm with labeled data and using it to make predictions on future (unlabeled) data.
- Labeled data is data that comes with labels, which in the regression case, are numbers. For example, the numbers could be prices of houses.
- In a dataset, the features are the properties that we use to predict the label. For example, if we want to predict housing prices, the features are anything that describes the house and which could determine the price, such as size, number of rooms, school quality, crime rate, age of the house, and distance to the highway.
- The linear regression method for predicting consists in assigning a weight to each of the features and adding the corresponding weights multiplied by the features, plus a bias.
- Graphically, we can see the linear regression algorithm as trying to pass a line as close as possible to a set of points.
- The way the linear regression algorithm works is by starting with a random line and then slowly moving it closer to each of the points that is misclassified, to attempt to classify them correctly.
- Polynomial regression is a generalization of linear regression, in which we use curves instead of lines to model our data. This is particularly useful when our dataset is nonlinear.
- Regression has numerous applications, including recommendation systems, ecommerce, and health care.

Exercises

Exercise 3.1

A website has trained a linear regression model to predict the amount of minutes that a user will spend on the site. The formula they have obtained is

$$\hat{t} = 0.8d + 0.5m + 0.5y + 0.2a + 1.5$$

where $\hat{t}$ is the predicted time in minutes, and d, m, y, and a are indicator variables (namely, they take only the values 0 or 1) defined as follows:

- d is a variable that indicates if the user is on desktop.
- m is a variable that indicates if the user is on mobile device.
- y is a variable that indicates if the user is young (under 21 years old).
- a is a variable that indicates if the user is an adult (21 years old or older).

Example: If a user is 30 years old and on a desktop, then $d = 1$, $m = 0$, $y = 0$, and $a = 1$.

If a 45-year-old user looks at the website from their phone, what is the expected time they will spend on the site?

Exercise 3.2

Imagine that we trained a linear regression model in a medical dataset. The model predicts the expected life span of a patient. To each of the features in our dataset, the model would assign a weight.

a) For the following quantities, state if you believe the weight attached to this quantity is a positive number, a negative number, or zero. Note: if you believe that the weight is a very small number, whether positive or negative, you can say zero.

1. Number of hours of exercise the patient gets per week
2. Number of cigarettes the patient smokes per week
3. Number of family members with heart problems
4. Number of siblings of the patient
5. Whether or not the patient has been hospitalized

b) The model also has a bias. Do you think the bias is positive, negative, or zero?

Exercise 3.3

The following is a dataset of houses with sizes (in square feet) and prices (in dollars).

	Size (s)	Prize (p)
House 1	100	200
House 2	200	475
House 3	200	400
House 4	250	520
House 5	325	735

Suppose we have trained the model where the prediction for the price of the house based on size is the following:

$$\hat{p} = 2s + 50$$

a. Calculate the predictions that this model makes on the dataset.

b. Calculate the mean absolute error of this model.

c. Calculate the root mean square error of this model.

Exercise 3.4

Our goal is to move the line with equation $\hat{y} = 2x + 3$ closer to the point $(x, y) = (5, 15)$ using the tricks we've learned in this chapter. For the following two problems, use the learning rate $\eta = 0.01$.

a. Apply the absolute trick to modify the line above to be closer to the point.

b. Apply the square trick to modify the line above to be closer to the point.