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.

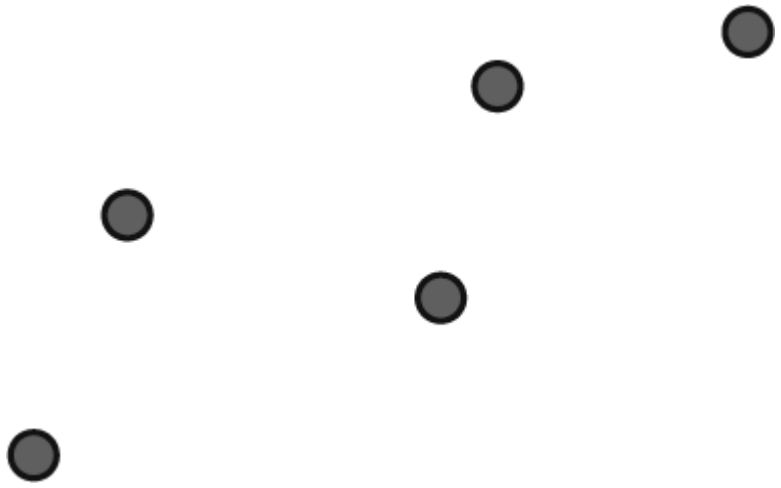


Figure 3.1 Some points that roughly look like they are forming a line

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.

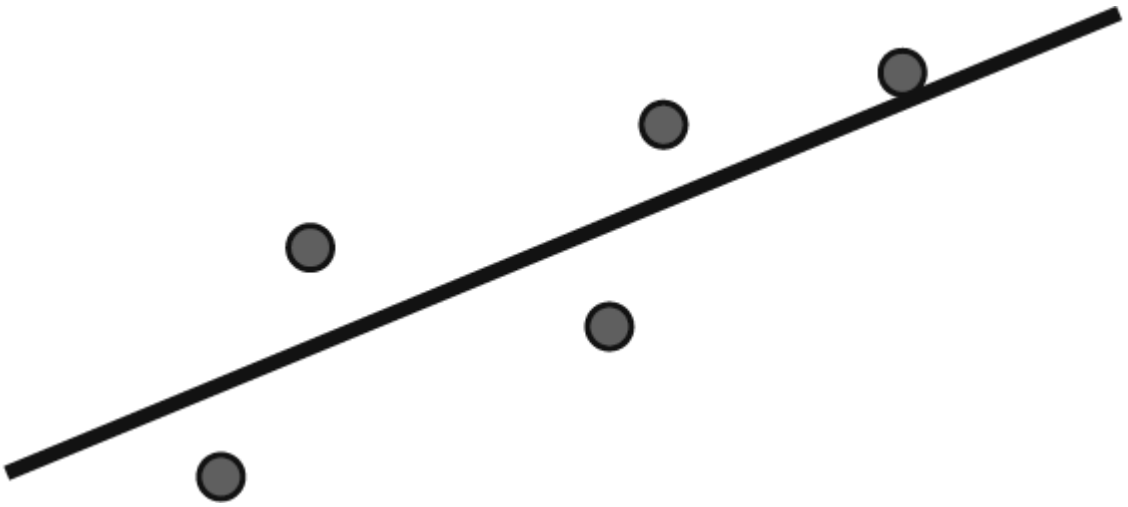


Figure 3.2 A line that passes close to the points

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:

Price = 100 + 50(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.

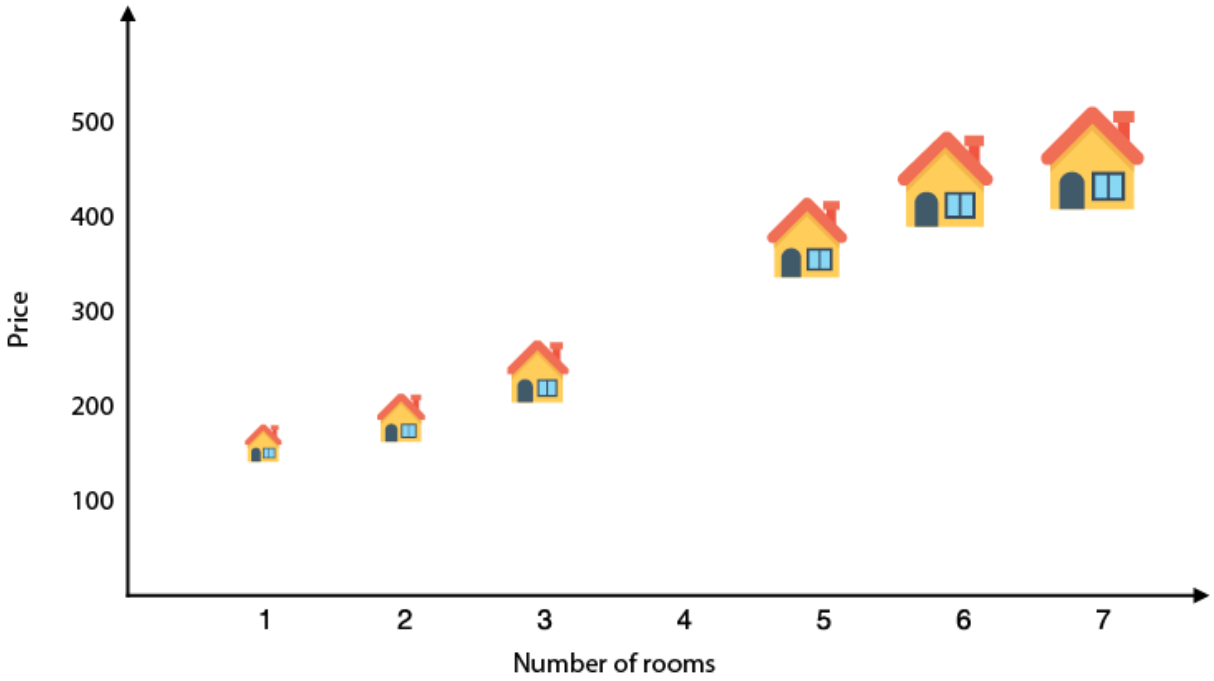


Figure 3.3 Plot of the dataset in table 3.2. The horizontal axis represents the number of rooms, and the vertical axis represents the price of the house.

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:

Price = 100 + 50(Number of rooms) + (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:

Price = 100 + 50 · 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.

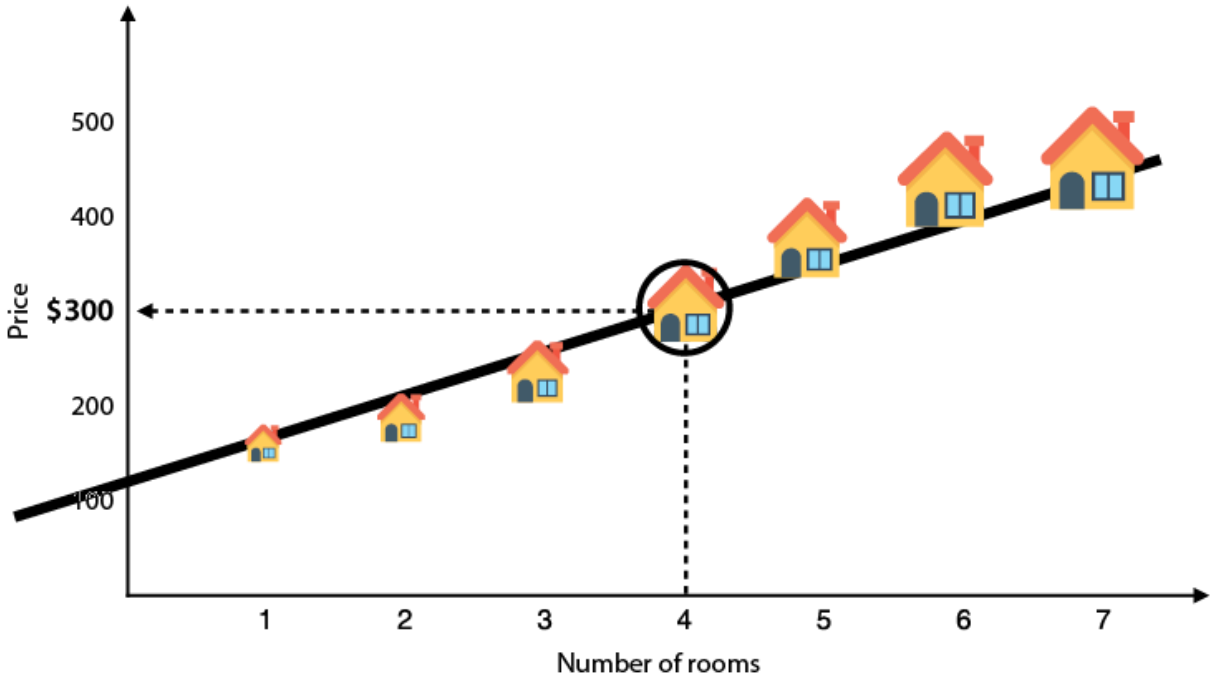


Figure 3.5 Our task is now to predict the price of the house with four rooms. Using the model (line), we deduce that the predicted price of this house is $300.

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:

Price = 30(number of rooms) + 1.5(size) + 10(quality of the schools) – 2(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.

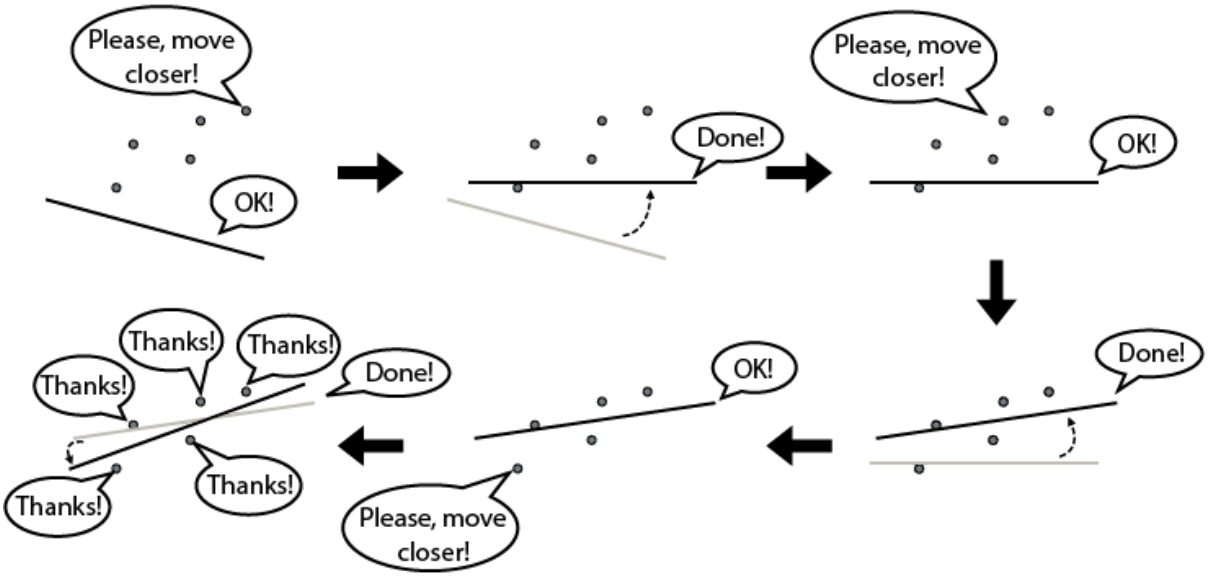


Figure 3.6 An illustration of the linear regression algorithm. We start at the top left with a random line and end in the bottom left with a line that fits the dataset well. At each stage, two things happen: (1) we pick a random point, and (2) the point asks the line to move closer to it. After many 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

Predicted price = (Price per room)(Number of rooms) + 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:

$y = 0.5x + 2$

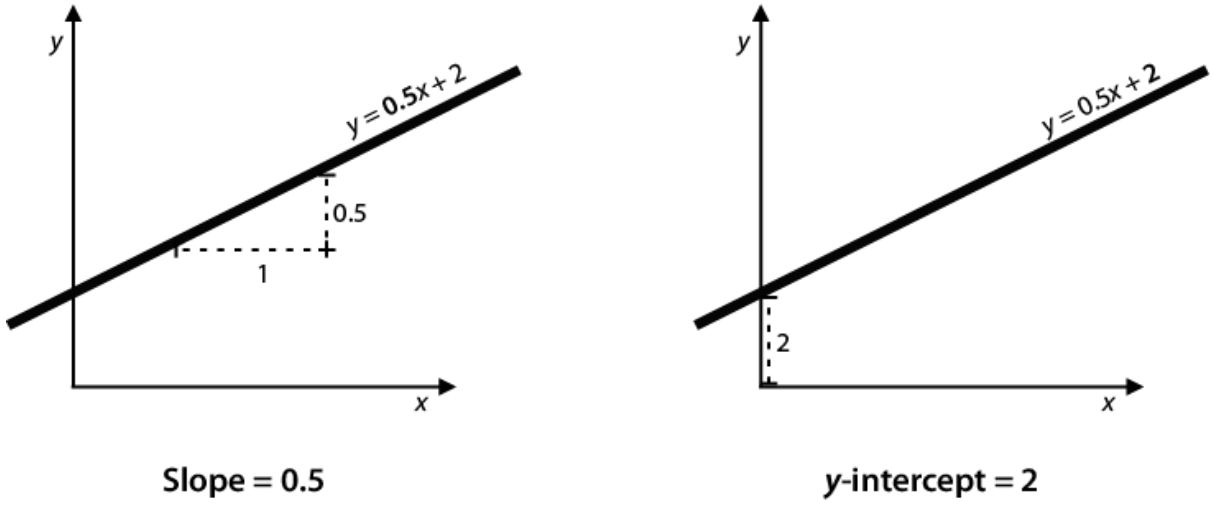


Figure 3.7 The line with equation **y** = 0.5**x** + 2 has slope 0.5 (left) and **y**-intercept 2 (right).

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.

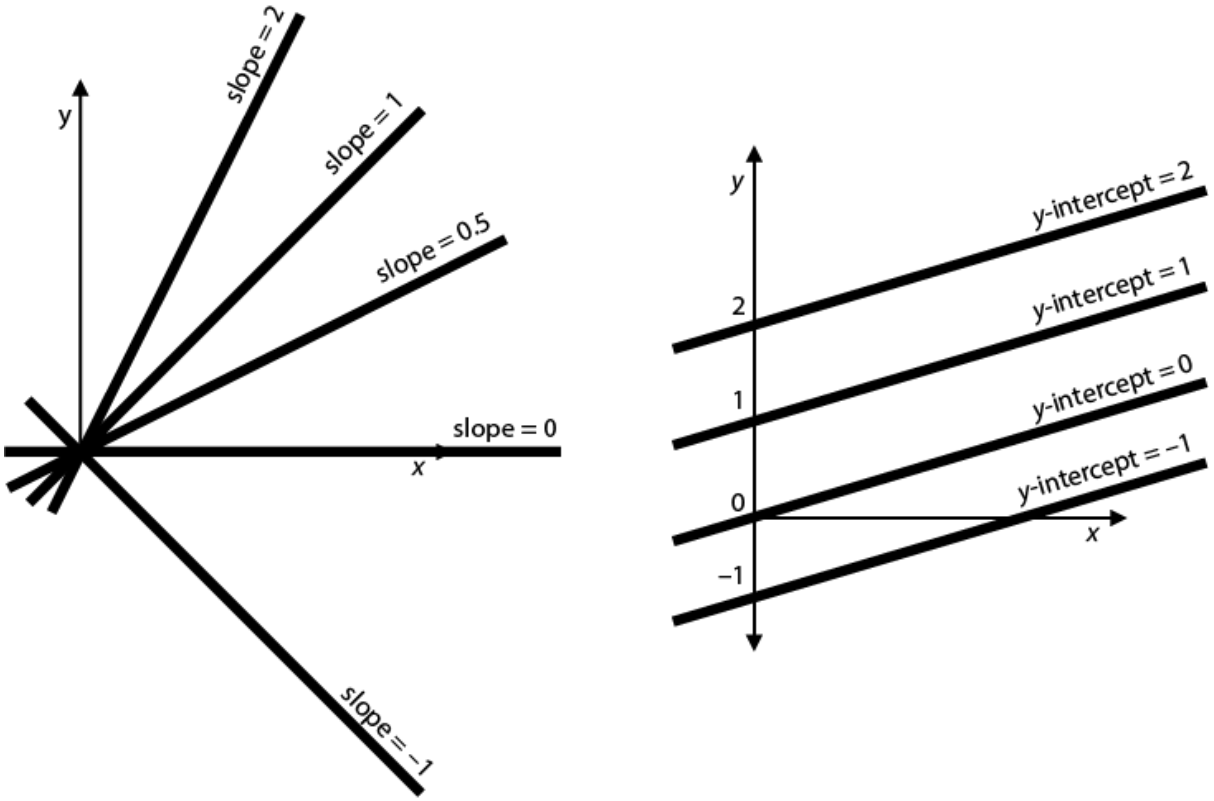


Figure 3.8 Some examples of slope and **y**-intercept. On the left, we see several lines with the same intercept and different slopes. Notice that the higher the slope, the steeper the line. On the right, we see several lines with the same slope and different **y**-intercepts. Notice that the higher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

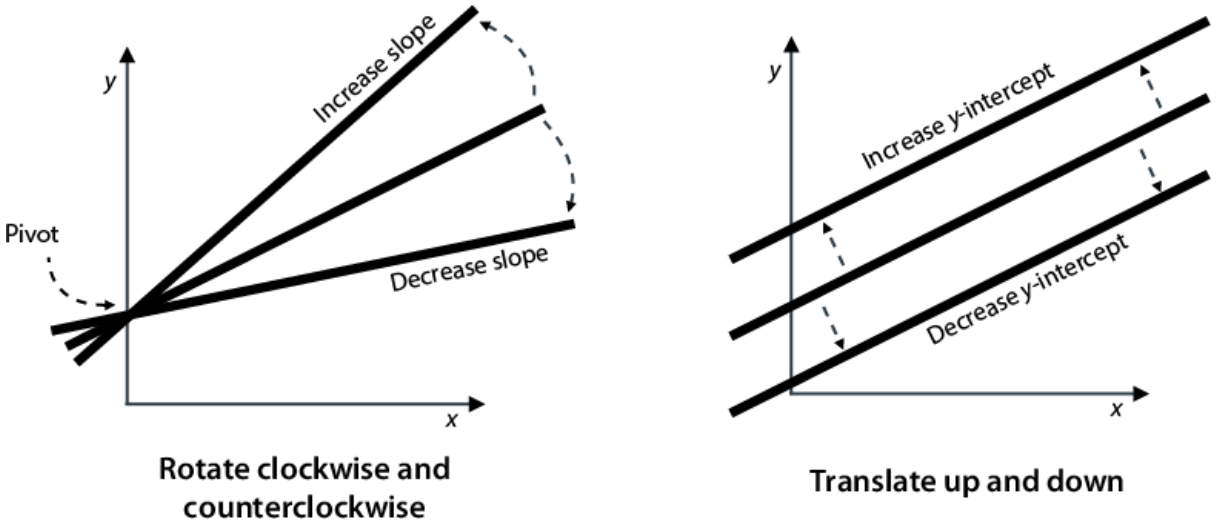


Figure 3.9 Left: Increasing the slope rotates the line counterclockwise, whereas decreasing the slope rotates it clockwise. Right: Increasing the **y**-intercept translates the line upward, whereas decreasing the **y**-intercept translates it downward.

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).

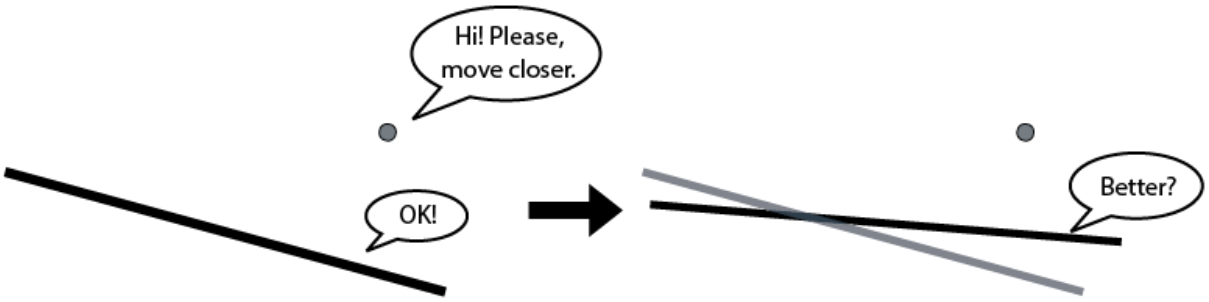


Figure 3.10 Our goal is to rotate and translate the line by a small amount to get closer to the point.

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.

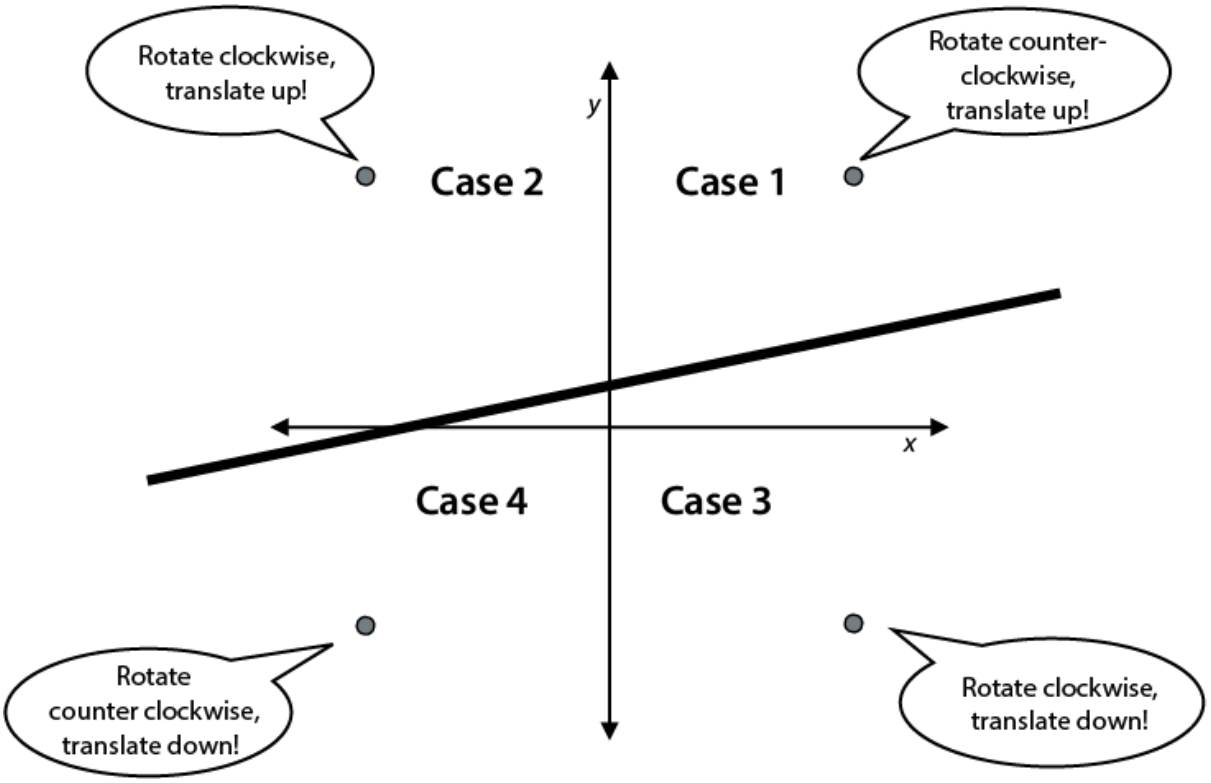


Figure 3.11 The four cases. In each of these we must rotate the line and translate it in a different way to move the line closer to the corresponding point.

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.

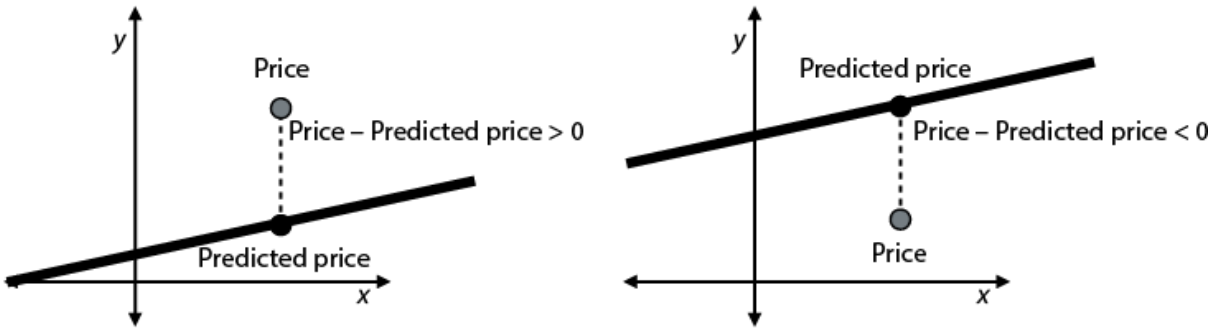


Figure 3.12 Left: When the point is above the line, the price is larger than the predicted price, so the difference is positive. Right: When the point is below the line, the price is smaller than the predicted price, so the difference is negative.

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.

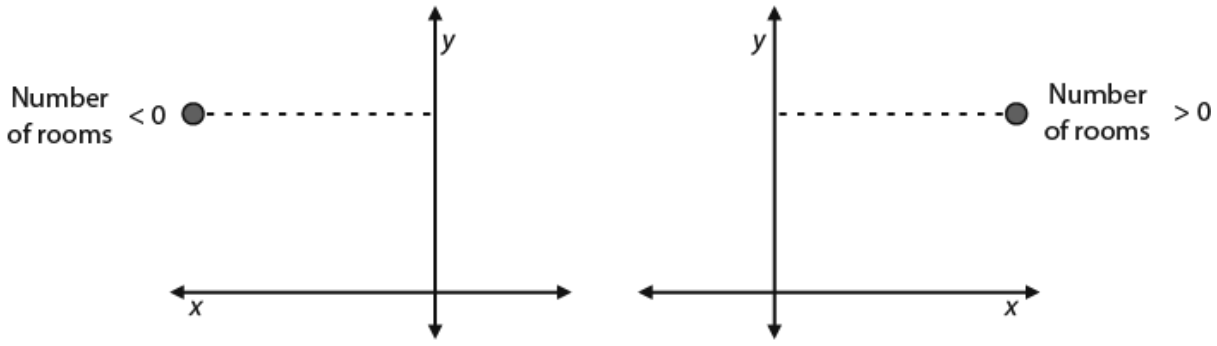


Figure 3.13 Left: When the point is to the left of the **y**-axis, the number of rooms is negative. Right: When the point is to the right of the **y**-axis, the number of rooms is positive.

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          ❶
    base_price += learning_rate*(price-predicted_price)              ❷
    price_per_room += learning_rate*num_rooms*(price-predicted_price) ❸
    return price_per_room, base_price
```

❶ Calculates the prediction

❷ Translates the line

❸ 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:

```
import random ❶
def linear_regression(features, labels, learning_rate=0.01, epochs = 1000):
    price_per_room = random.random()
    base_price = random.random() ❷
    for epoch in range(epochs):  ❸
        i = random.randint(0, len(features)-1) ❹
        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) ❺
    return price_per_room, base_price
```

❶ Imports the random package to generate (pseudo) random numbers

❷ Generates random values for the slope and the *y*-intercept

❸ Repeats the update step many times

❹ Picks a random point on our dataset

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

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.

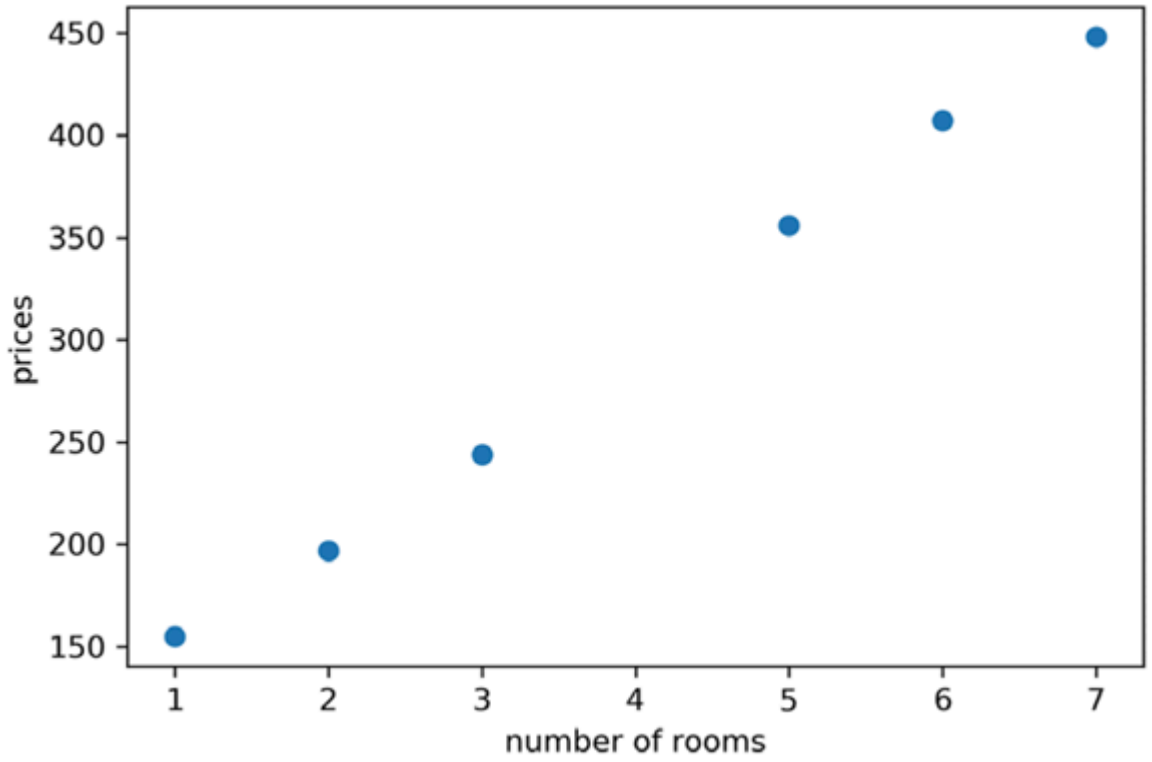


Figure 3.14 The plot of the points in table 3.2

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)
```

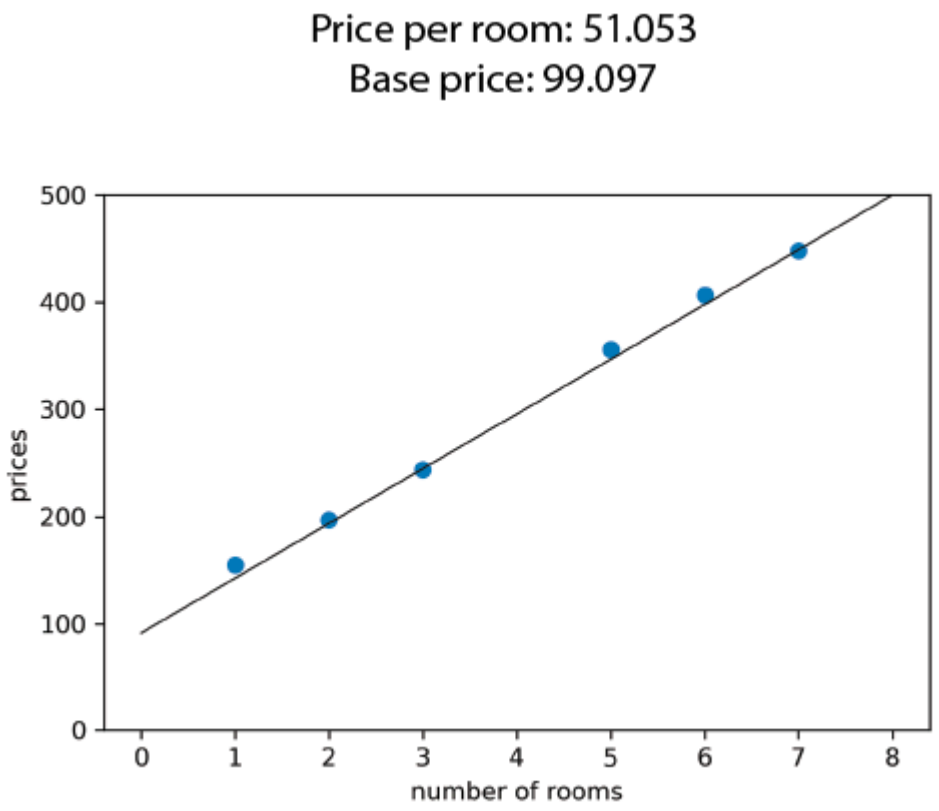


Figure 3.15 The plot of the points in table 3.2 and the line that we obtained with the linear regression algorithm

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.

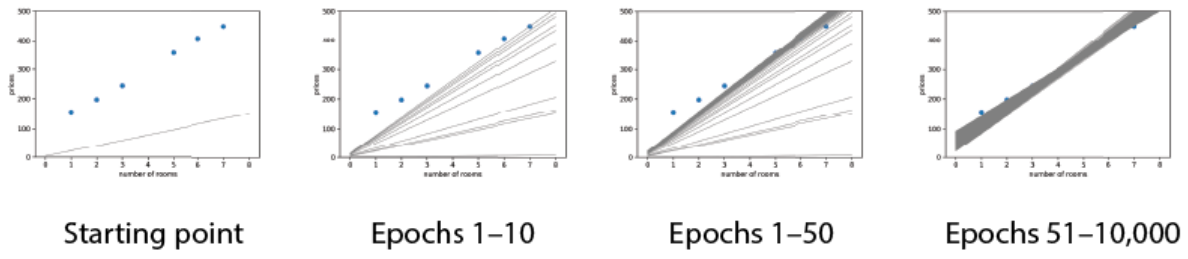


Figure 3.16 Drawing some of the lines in our algorithm, as we approach a better solution. The first graphic shows the starting point. The second graphic shows the first 10 epochs of the linear regression algorithm. Notice how the line is moving closer to fitting the points. The third graphic shows the 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 $x^{(1)}$, $x^{(2)}$, … , $x^{(m)}$. Each point is of the form $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, \dots , x_n^{(i)})$.
- The corresponding labels are y_1, y_2, … , y_m.
- The weights of the model are w_1, w_2, … , w_n.
- The bias of the model is b.

Pseudocode for the general square trick

Inputs:

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

Output:

- A model with equation $\hat{y} = w_1'x_1 + w_2'x_2 + \dots + w_n'x_n + b'$ 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 $\hat{y} = w_1'x_1 + w_2'x_2 + \ldots + w_n'x_n + b'$

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.

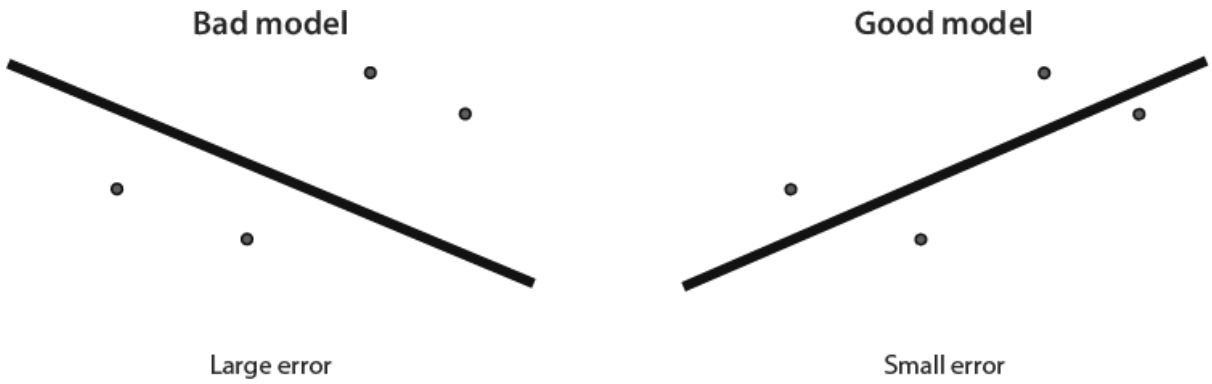


Figure 3.17 Two models, a bad one (on the left) and a good one (on the right). The bad one is assigned a large error, and the good one is assigned a small error.

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.

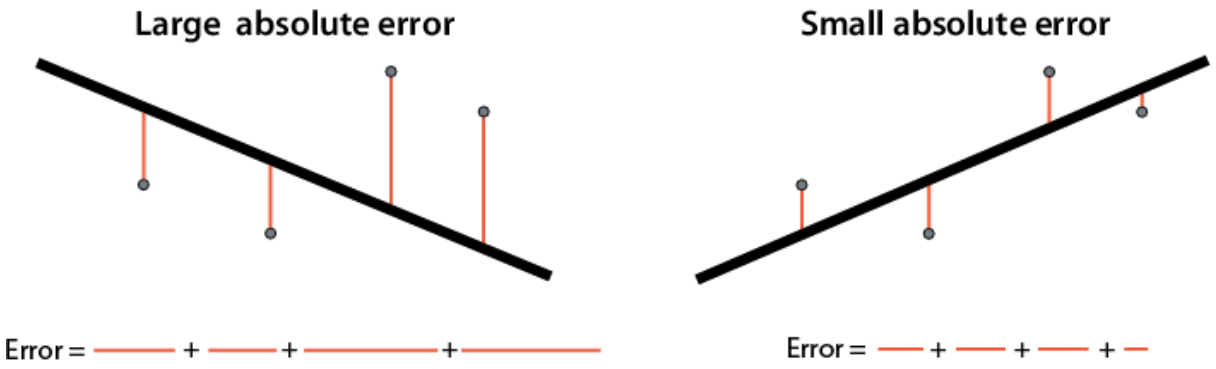


Figure 3.18 The absolute error is the sum of the vertical distances from the points to the line. Note that the absolute error is large for the bad model on the left and small for the good model on the right.

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.

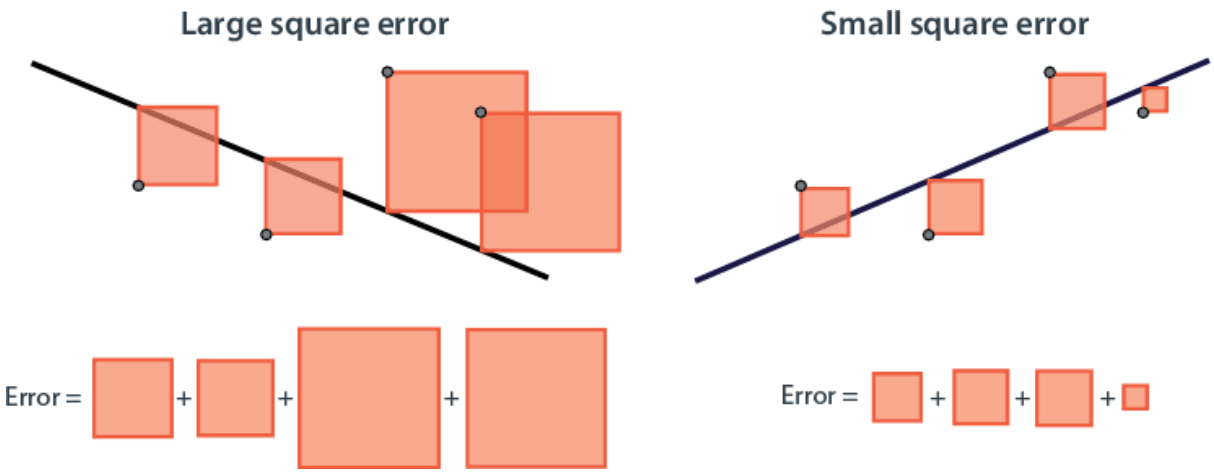


Figure 3.19 The square error is the sum of the squares of the vertical distances from the points to the line. Note that the square error is large for the bad model on the left and small for the good model on the right.

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.

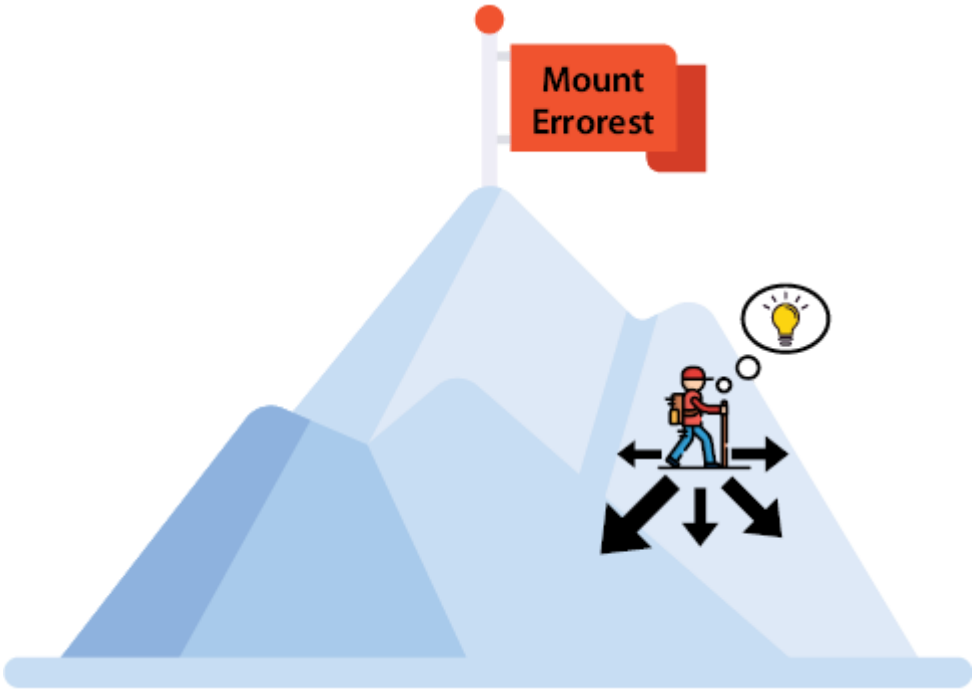


Figure 3.20 We are on top of Mount Errorest and wish to get to the bottom, but we can't see very far. A way to go down is to look at all the directions in which we can take one step and figure out which one helps us descend the most. Then we are one step closer to the bottom.

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.

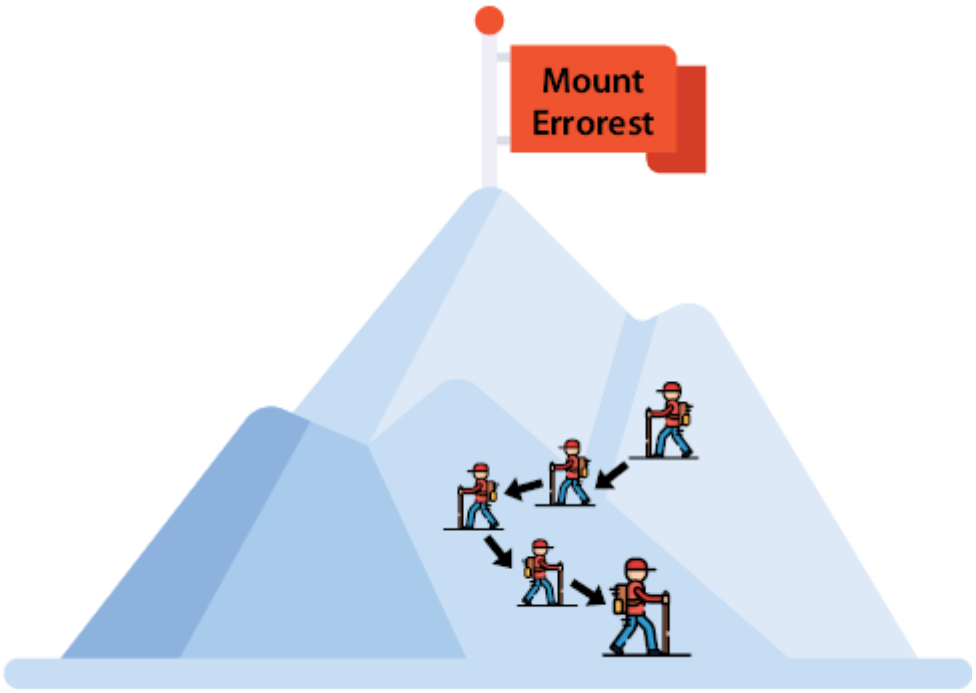


Figure 3.21 The way to descend from the mountain is to take that one small step in the direction that makes us descend the most and to continue doing this for a long time.

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.

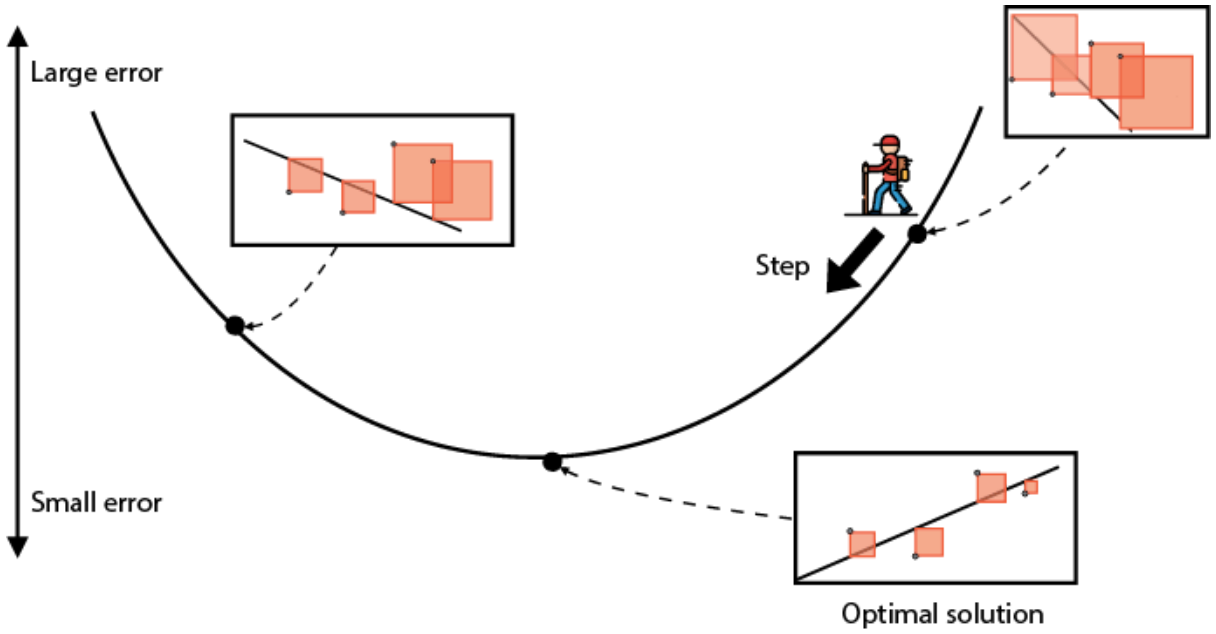


Figure 3.22 Each point on this mountain corresponds to a different model. The points below are good models with a small error, and the points above are bad models with a large error. The goal is to descend from this mountain. The way to descend is by starting somewhere and continuously taking a step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

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 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.

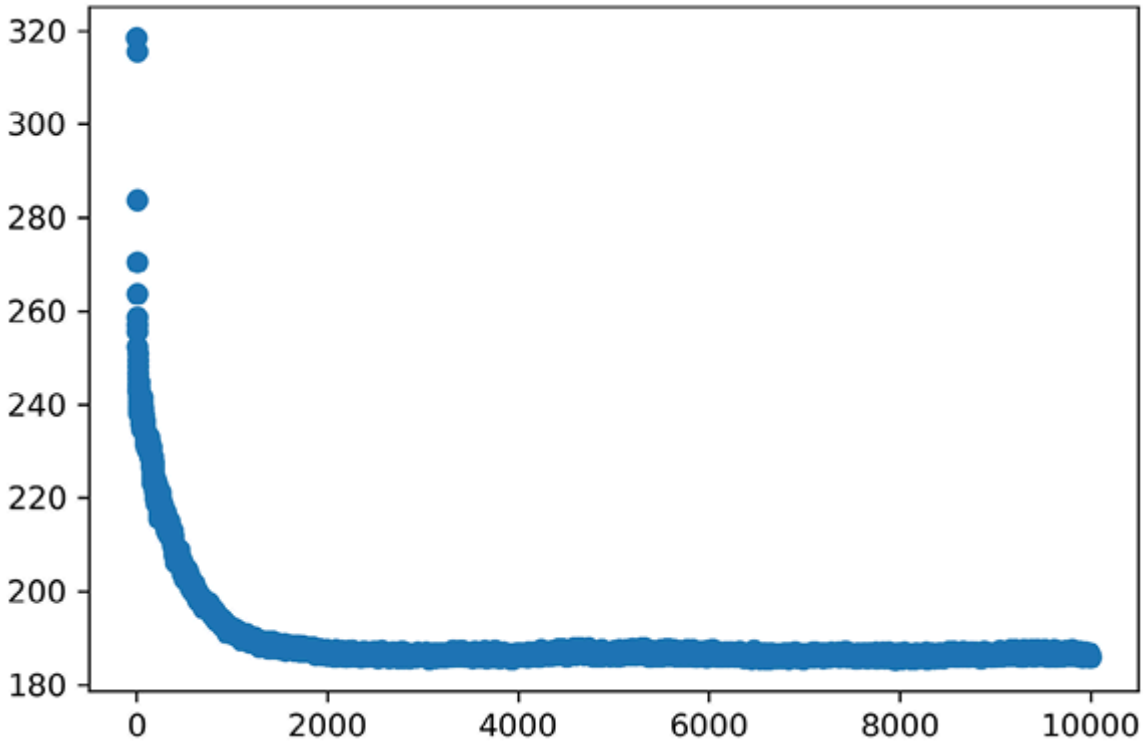


Figure 3.23 The plot of the root mean square error for our running example. Notice how the algorithm succeeded in reducing this error after a little over 1,000 iterations. This means that we don't need to keep running this algorithm for 10,000 iterations, because around 2,000 of them do the job.

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.

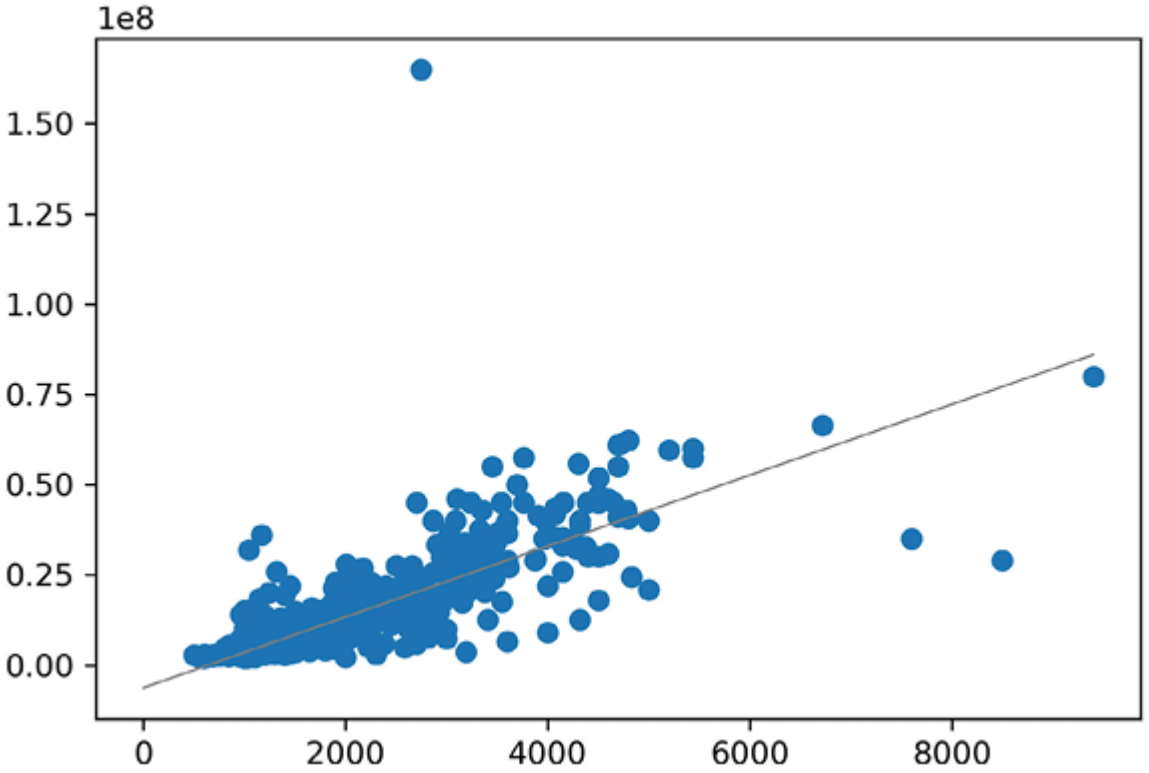


Figure 3.24 The Hyderabad housing prices dataset restricted to area and price. The line is the model we've obtained using only the area feature to predict the price.

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, …. 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.

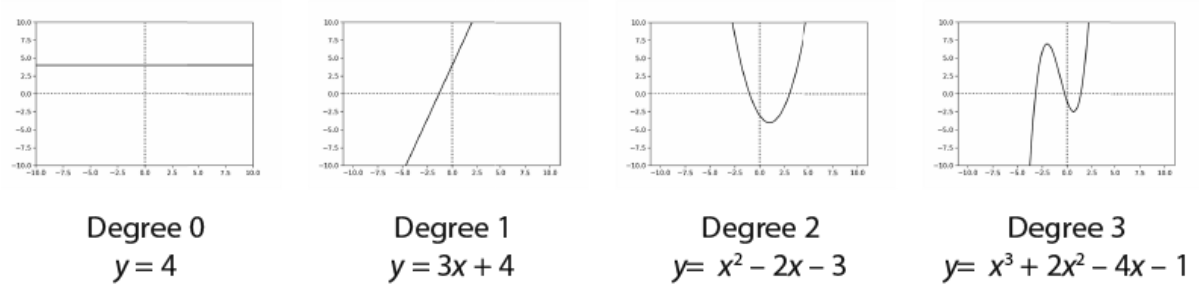


Figure 3.25 Polynomials are functions that help us model our data better. Here are the plots of four polynomials of degrees 0 to 3. Note that the polynomial of degree 0 is a horizontal line, the polynomial of degree 1 is any line, the polynomial of degree 2 is a parabola, and the polynomial of degree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

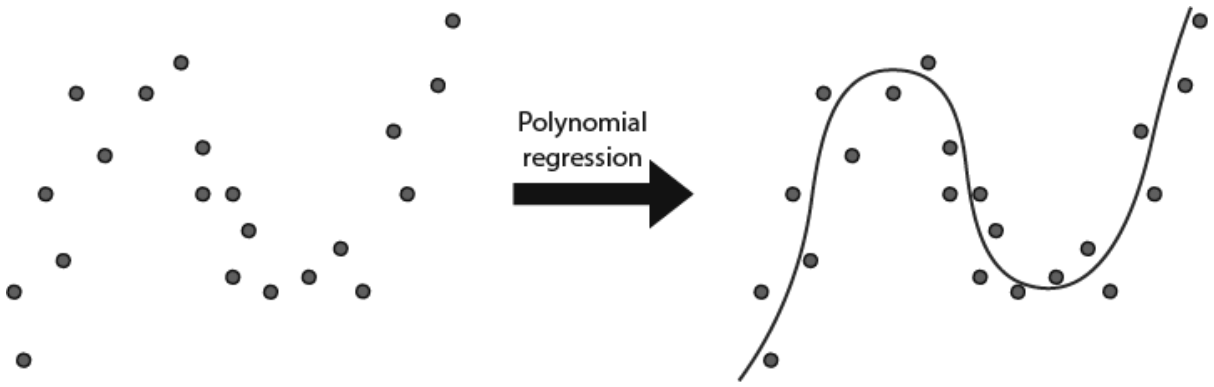


Figure 3.26 Polynomial regression is useful when it comes to modeling nonlinear data. If our data looks like the left part of the figure, it will be hard to find a line that fits it well. However, a curve will fit the data well, as you can see in the right part of the figure. Polynomial regression helps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

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

1. Calculate the predictions that this model makes on the dataset.
2. Calculate the mean absolute error of this model.
3. 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 $\eta = 0.01$.

1. Apply the absolute trick to modify the line above to be closer to the point.
2. Apply the square trick to modify the line above to be closer to the point.

Decision Tree.
ecision trees are powerful classification and regression models, which also give us a great deal of information about our dataset. Just like the previous models we've learned in this book, decision trees are trained with labeled data, where the labels that we want to predict can be classes (for classification) or values (for regression). For most of this chapter, we

focus on decision trees for classification, but near the end of the chapter, we describe decision trees for regression. However, the structure and training process of both types of tree is similar. In this chapter, we develop several use cases, including an app-recommendation system and a model for predicting admissions at a university.

Decision trees follow an intuitive process to make predictions—one that very much resembles human reasoning. Consider the following scenario: we want to decide whether we should wear a jacket today. What does the decision process look like? We may look outside and check if it's raining. If it's raining, then we definitely wear a jacket. If it's not, then maybe we check the temperature. If it is hot, then we don't wear a jacket, but if it is cold, then we wear a jacket. In figure 9.1, we can see a graph of this decision process, where the decisions are made by traversing the tree from top to bottom.

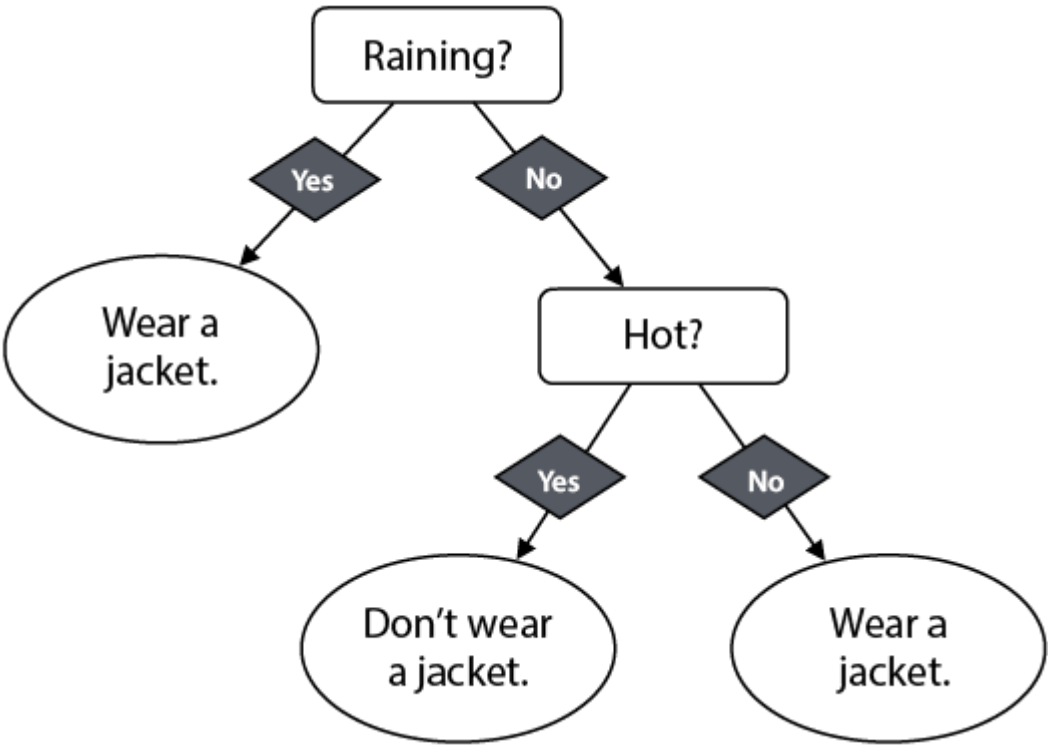


Figure 9.1 A decision tree used to decide whether we want to wear a jacket or not on a given day. We make the decision by traversing the tree down and taking the branch corresponding to each correct answer.

Our decision process looks like a tree, except it is upside down. The tree is formed of vertices, called *nodes*, and edges. On the very top, we can see the *root node*, from which two branches emanate. Each of the nodes has either two or zero branches (edges) emanating from them, and for this reason, we call it a *binary tree*. The nodes that have two branches emanating from them are called *decision nodes*, and the nodes with no branches emanating from them are called *leaf nodes*, or *leaves*. This arrangement of nodes, leaves, and edges is what we call a decision tree. Trees are natural objects in computer science, because computers break every process into a sequence of binary operations.

The simplest possible decision tree, called a *decision stump*, is formed by a single decision node (the root node) and two leaves. This represents a single yes-or-no question, based on which we immediately make a decision.

The depth of a decision tree is the number of levels underneath the root node. Another way to measure it is by the length of the longest path from the root node to a leaf, where a path is measured by the number of edges it contains. The tree in figure 9.1 has a depth of 2. A decision stump has a depth of 1.

Here is a summary of the definitions we've learned so far:

decision tree A machine learning model based on yes-or-no questions and represented by a binary tree. The tree has a root node, decision nodes, leaf nodes, and branches.

root node The topmost node of the tree. It contains the first yes-or-no question. For convenience, we refer to it as the *root*.

decision node Each yes-or-no question in our model is represented by a decision node, with two branches emanating from it (one for the "yes" answer, and one for the "no" answer).

leaf node A node that has no branches emanating from it. These represent the decisions we make after traversing the tree. For convenience, we refer to them as *leaves*.

branch The two edges emanating from each decision node, corresponding to the "yes" and "no" answers to the question in the node. In this chapter, by convention, the branch to the left corresponds to "yes" and the branch to the right to "no."

depth The number of levels in the decision tree. Alternatively, it is the number of branches on the longest path from the root node to a leaf node.

Throughout this chapter, nodes are drawn as rectangles with rounded edges, the answers in the branches as diamonds, and leaves as ovals. Figure 9.2 shows how a decision tree looks in general.

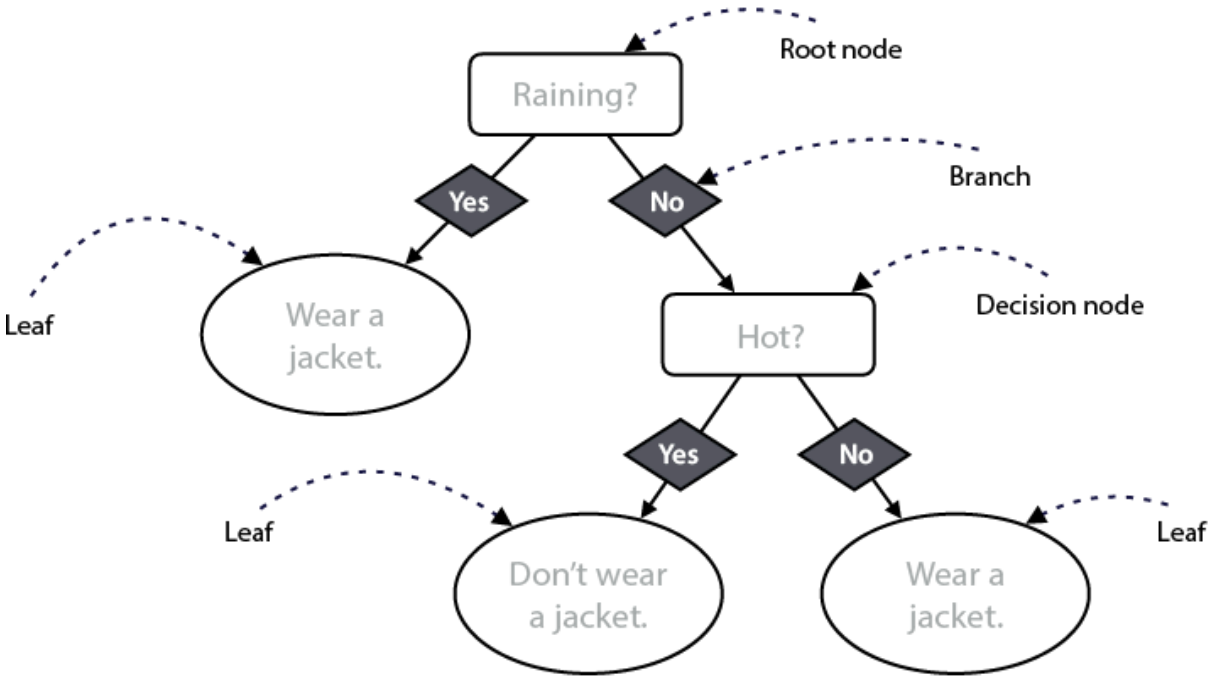


Figure 9.2 A regular decision tree with a root node, decision nodes, branches, and leaves. Note that each decision node contains a yes-or-no question. From each possible answer, one branch emanates, which can lead to another decision node or a leaf. This tree has a depth of 2, because the longest path from a leaf to the root goes through two branches.

How did we build this tree? Why were those the questions we asked? We could have also checked if it was Monday, if we saw a red car outside, or if we were hungry, and built the following decision tree:

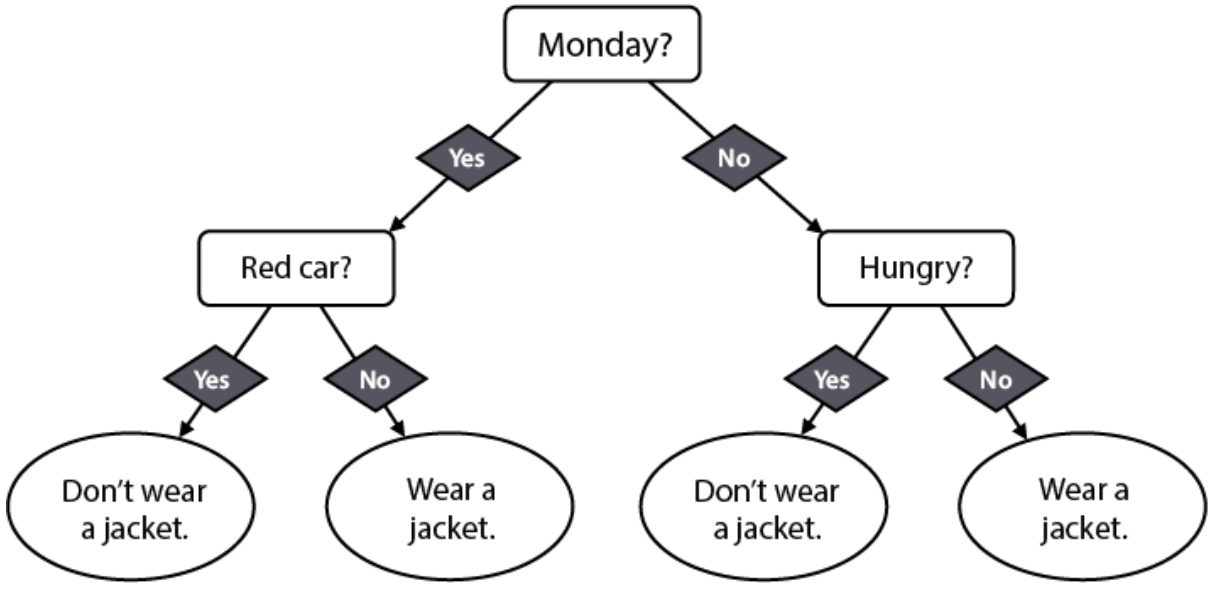


Figure 9.3 A second (maybe not as good) decision tree we could use to decide whether we want to wear a jacket on a given day

Which tree do we think is better when it comes to deciding whether or not to wear a jacket: tree 1 (figure 9.1) or tree 2 (figure 9.3)? Well, as humans, we have enough experience to

figure out that tree 1 is much better than tree 2 for this decision. How would a computer know? Computers don't have experience per se, but they have something similar, which is data. If we wanted to think like a computer, we could just go over all possible trees, try each one of them for some time—say, one year—and compare how well they did by counting how many times we made the right decision using each tree. We'd imagine that if we use tree 1, we were correct most days, whereas if we used tree 2, we may have ended up freezing on a cold day without a jacket or wearing a jacket on an extremely hot day. All a computer has to do is go over all trees, collect data, and find which one is the best one, right?

Almost! Unfortunately, even for a computer, searching over all the possible trees to find the most effective one would take a really long time. But luckily, we have algorithms that make this search much faster, and thus, we can use decision trees for many wonderful applications, including spam detection, sentiment analysis, and medical diagnosis. In this chapter, we'll go over an algorithm for constructing good decision trees quickly. In a nutshell, we build the tree one node at a time, starting from the top. To pick the right question corresponding to each node, we go over all the possible questions we can ask and pick the one that is right the highest number of times. The process goes as follows:

Picking a good first question

We need to pick a good first question for the root of our tree. What would be a good question that helps us decide whether to wear a jacket on a given day? Initially, it can be anything. Let's say we come up with five candidates for our first question:

1. Is it raining?
2. Is it cold outside?
3. Am I hungry?
4. Is there a red car outside?
5. Is it Monday?

Out of these five questions, which one seems like the best one to help us decide whether we should wear a jacket? Our intuition says that the last three questions are useless to help us decide. Let's say that from experience, we've noticed that among the first two, the first one is more useful. We use that question to start building our tree. So far, we have a simple decision tree, or a decision stump, consisting of that single question, as illustrated in Figure 9.4.

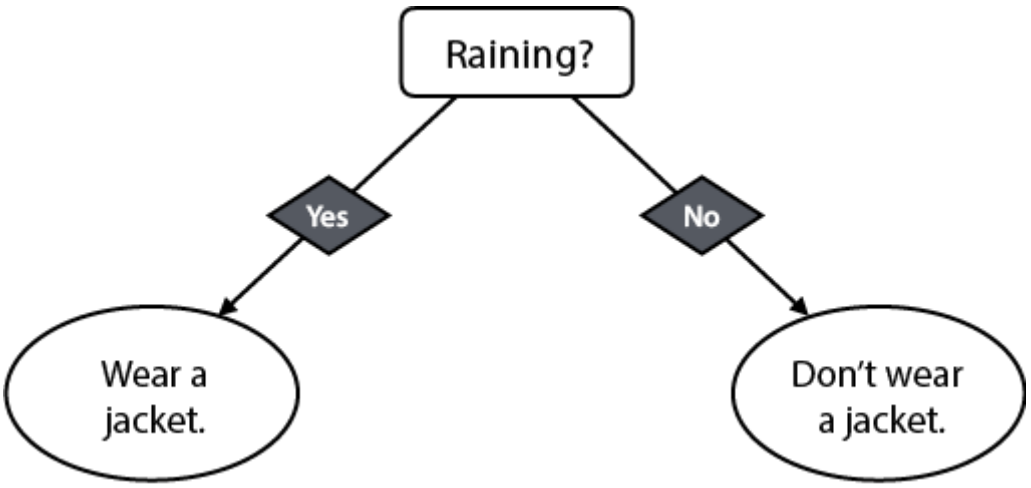


Figure 9.4 A simple decision tree (decision stump) that consists of only the question, “Is it raining?” If the answer is yes, the decision we make is to wear a jacket.

Can we do better? Imagine that we start noticing that when it rains, wearing a jacket is always the correct decision. However, there are days on which it doesn't rain, and not wearing a jacket is not the correct decision. This is where question 2 comes to our rescue. We use that question to help us in the following way: after we check that it is not raining, *then* we check the

temperature, and if it is cold, we decide to wear a jacket. This turns the left leaf of the tree into a node, with two leaves emanating from it, as shown in figure 9.5.

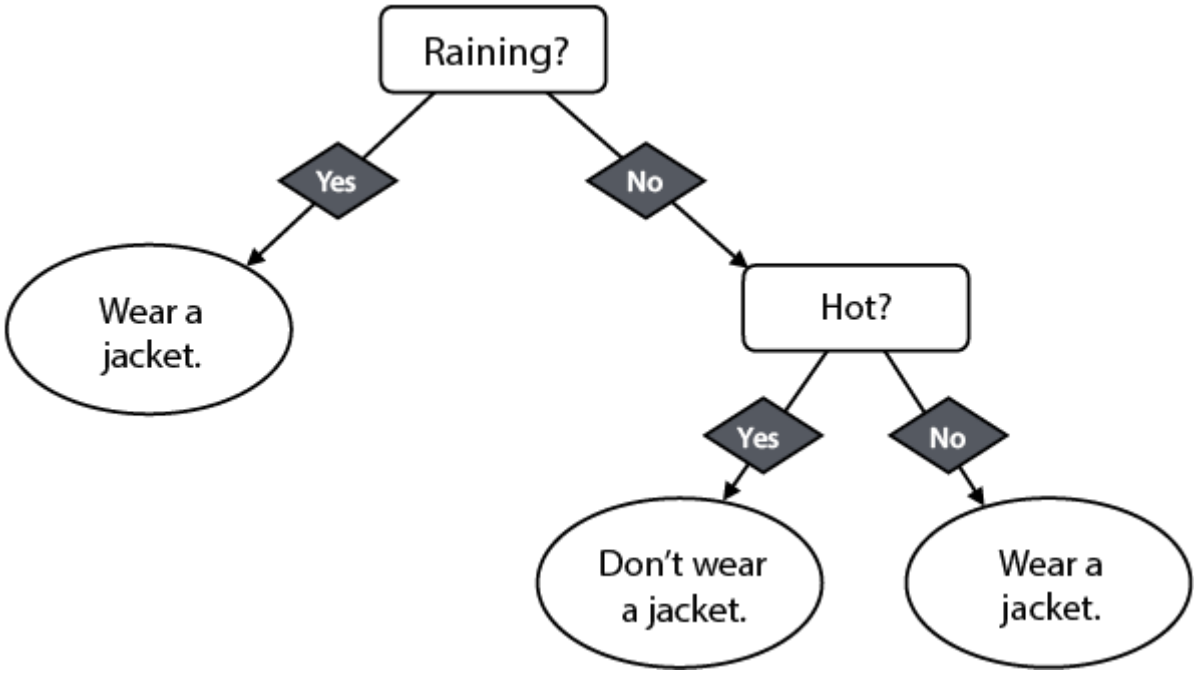


Figure 9.5 A slightly more complicated decision tree than the one in figure 9.4, where we have picked one leaf and split it into two further leaves. This is the same tree as in figure 9.1.

Now we have our decision tree. Can we do better? Maybe we can if we add more nodes and leaves to our tree. But for now, this one works very well. In this example, we made our decisions using our intuition and our experience. In this chapter, we learn an algorithm that builds these trees solely based on data.

Many questions may arise in your head, such as the following:

1. How exactly do you decide which is the best possible question to ask?
2. Does the process of always picking the best possible question actually get us to build *the* best decision tree?
3. Why don't we instead build all the possible decision trees and pick the best one from there?
4. Will we code this algorithm?
5. Where can we find decision trees in real life?
6. We can see how decision trees work for classification, but how do they work for regression?

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

1. **How exactly do you decide which is the best possible question to ask?**
   We have several ways to do this. The simplest one is using accuracy, which means: which question helps me be correct more often? However, in this chapter, we also learn other methods, such as Gini index or entropy.

2. **Does the process of always picking the best possible question actually get us to build *the* best decision tree?**
   Actually, this process does not guarantee that we get the best possible tree. This is what we call a *greedy algorithm*. Greedy algorithms work as follows: at every point, the algorithm makes the best possible available move. They tend to work well, but it's not always the case that making the best possible move at each timestep gets you to the best overall outcome. There may be times in which asking a weaker question groups our data in a way that we end up with a better tree at the end of the day. However, the algorithms for building decision trees tend to work very well and very quickly, so we'll live with this. Look at the algorithms that we see in this chapter, and try to figure out ways to improve them by removing the greedy property!

3. **Why don't we instead build all the possible decision trees and pick the best one from there?**
   The number of possible decision trees is very large, especially if our dataset has many features. Going through all of them would be very slow. Here, finding each node requires only a linear search across the features and not across all the possible trees, which makes it much faster.

4. **Will we code this algorithm?**
   This algorithm can be coded by hand. However, we'll see that because it is recursive, the coding can get a bit tedious. Thus, we'll use a useful package called Scikit-Learn to build decision trees with real data.

5. **Where can we find decision trees in real life?**
   In many places! They are used extensively in machine learning, not only because they work very well but also because they give us a lot of information on our data. Some places in which decision trees are used are in recommendation systems (to recommend videos, movies, apps, products to buy, etc)., in spam classification (to decide whether or not an email is spam), in sentiment analysis (to decide whether a sentence is happy or sad), and in biology (to decide whether or not a patient is sick or to help identify certain hierarchies in species or in types of genomes).

6. **We can see how decision trees work for classification, but how do they work for regression?**
   A regression decision tree looks exactly like a classification decision tree, except for the leaves. In a classification decision tree, the leaves have classes, such as yes and no. In a regression decision tree, the leaves have values, such as 4, 8.2, or –199. The prediction our model makes is given by the leaf at which we arrived when traversing the tree in a downward fashion.

The first use case that we'll study in this chapter is a popular application in machine learning, and one of my favorites: recommendation systems.

The code for this chapter is available in this GitHub repository: https://github.com/luisguiserrano/manning/tree/master/Chapter_9_Decision_Trees.

The problem: We need to recommend apps to users according to what they are likely to download

Recommendation systems are one of the most common and exciting applications in machine learning. Ever wonder how Netflix recommends movies, YouTube guesses which videos you may watch, or Amazon shows you products you might be interested in buying? These are all examples of recommendation systems. One simple and interesting way to see recommendation problems is to consider them classification problems. Let's start with an easy example: our very own app-recommendation system using decision trees.

Let's say we want to build a system that recommends to users which app to download among the following options. We have the following three apps in our store (figure 9.6):

- **Atom Count**: an app that counts the number of atoms in your body
- **Beehive Finder**: an app that maps your location and finds the closest beehives
- **Check Mate Mate**: an app for finding Australian chess players

Figure 9.6 The three apps we are recommending: Atom Count, an app for counting the number of atoms in your body; Beehive Finder, an app for locating the nearest beehives to your location; and Check Mate Mate, an app for finding Australian chess players in your area

The training data is a table with the platform used by the user (iPhone or Android), their age, and the app they have downloaded (in real life there are many more platforms, but for simplicity we'll assume that these are the only two options). Our table contains six people, as shown in table 9.1.

Table 9.1 A dataset with users of an app store. For each customer, we record their platform, age, and the app they downloaded.

Platform	Age	App
iPhone	15	Atom Count
iPhone	25	Check Mate Mate
Android	32	Beehive Finder
iPhone	35	Check Mate Mate
Android	12	Atom Count
Android	14	Atom Count

Given this table, which app would you recommend to each of the following three customers?

- **Customer 1**: a 13-year-old iPhone user
- **Customer 2**: a 28-year-old iPhone user
- **Customer 3**: a 34-year-old Android user

What we should do follows:

Customer 1: a 13-year-old iPhone user. To this customer, we should recommend Atom Count, because it seems (looking at the three customers in their teens) that young people tend to download Atom Count.

Customer 2: a 28-year-old iPhone user. To this customer, we should recommend Check Mate Mate, because looking at the two iPhone users in the dataset (aged 25 and 35), they both downloaded Check Mate Mate.

Customer 3: a 34-year-old Android user. To this customer, we should recommend Beehive Finder, because there is one Android user in the dataset who is 32 years old, and they downloaded Beehive Finder.

However, going customer by customer seems like a tedious job. Next, we'll build a decision tree to take care of all customers at once.

The solution: Building an app-recommendation system

In this section, we see how to build an app-recommendation system using decision trees. In a nutshell, the algorithm to build a decision tree follows:

1. Figure out which of the data is the most useful to decide which app to recommend.
2. This feature splits the data into two smaller datasets.
3. Repeat processes 1 and 2 for each of the two smaller datasets.

In other words, what we do is decide which of the two features (platform or age) is more successful at determining which app the users will download and pick this one as our root of the decision tree. Then, we iterate over the branches, always picking the most determining feature for the data in that branch, thus building our decision tree.

First step to build the model: Asking the best question

The first step to build our model is to figure out the most useful feature: in other words, the most useful question to ask. First, let's simplify our data a little bit. Let's call everyone under 20 years old "Young" and everyone 20 or older "Adult" (don't worry—we'll go back to the original dataset soon, in the section "Splitting the data using continuous features, such as age"). Our modified dataset is shown in table 9.2.

Table 9.2 A simplified version of the dataset in table 9.1, where the age column has been simplified to two categories, "young" and "adult"

Platform	Age	App
iPhone	Young	Atom Count
iPhone	Adult	Check Mate Mate
Android	Adult	Beehive Finder
iPhone	Adult	Check Mate Mate
Android	Young	Atom Count

Android	Young	Atom Count

The building blocks of decision trees are questions of the form “Does the user use an iPhone?” or “Is the user young?” We need one of these to use as our root of the tree. Which one should we pick? We should pick the one that best determines the app they downloaded. To decide which question is better at this, let’s compare them.

First question: Does the user use an iPhone or Android?

This question splits the users into two groups, the iPhone users and Android users. Each group has three users in it. But we need to keep track of which app each user downloaded. A quick look at table 9.2 helps us notice the following:

- Of the iPhone users, one downloaded Atom Count and two downloaded Check Mate Mate.
- Of the Android users, two downloaded Atom Count and one downloaded Beehive Finder.

The resulting decision stump is shown in figure 9.7.

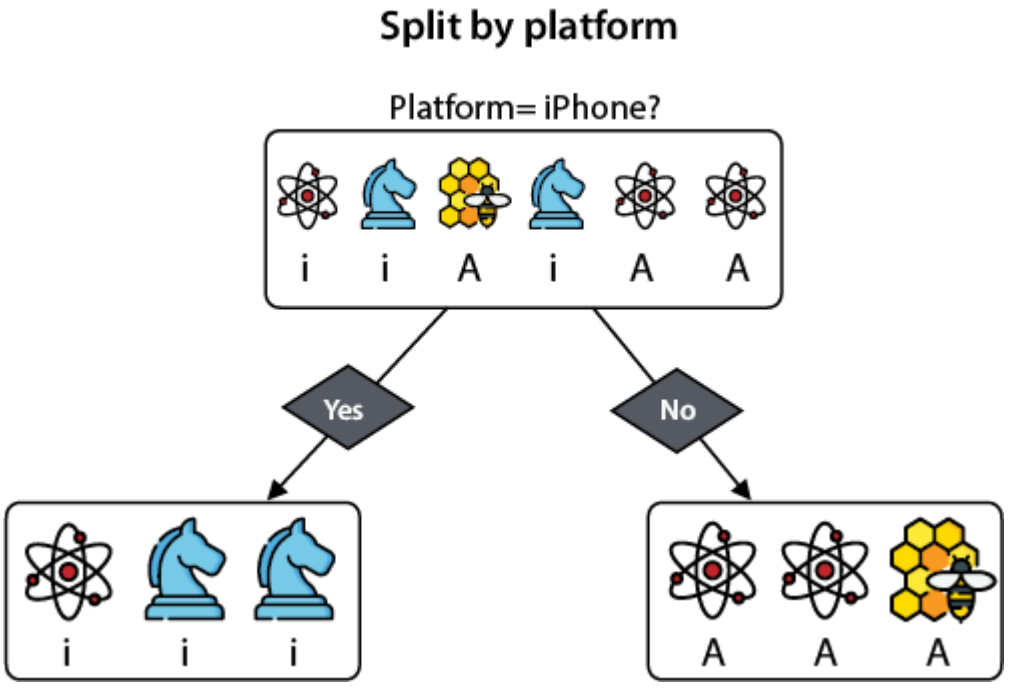


Figure 9.7 If we split our users by platform, we get this split: the iPhone users are on the left, and the Android users on the right. Of the iPhone users, one downloaded Atom Count and two downloaded Check Mate Mate. Of the Android users, two downloaded Atom Count and one downloaded Beehive Finder.

Now let’s see what happens if we split them by age.

Second question: Is the user young or adult?

This question splits the users into two groups, the young and the adult. Again, each group has three users in it. A quick look at table 9.2 helps us notice what each user downloaded, as follows:

- The young users all downloaded Atom Count.
- Of the adult users, two downloaded Atom Count and one downloaded Beehive Finder.

The resulting decision stump is shown in figure 9.8.

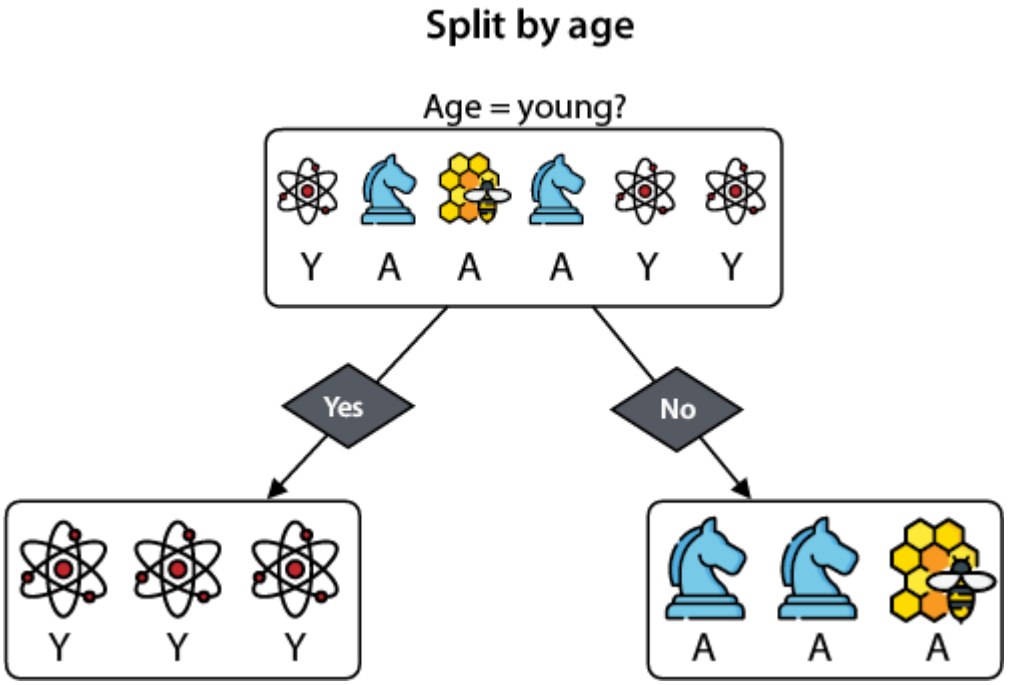


Figure 9.8 If we split our users by age, we get this split: the young are on the left, and the adults on the right. Out of the young users, all three downloaded Atom Count. Out of the adult users, one downloaded Beehive Finder and two downloaded Check Mate Mate.

From looking at figures 9.7 and 9.8, which one looks like a better split? It seems that the second one (based on age) is better, because it has picked up on the fact that all three young people downloaded Atom Count. But we need the computer to figure out that age is a better feature, so we'll give it some numbers to compare. In this section, we learn three ways to compare these two splits: accuracy, Gini impurity, and entropy. Let's start with the first one: accuracy.

Accuracy: How often is our model correct?

We learned about accuracy in chapter 7, but here is a small recap. Accuracy is the fraction of correctly classified data points over the total number of data points.

Suppose that we are allowed only one question, and with that one question, we must determine which app to recommend to our users. We have the following two classifiers:

- **Classifier** 1: asks the question “What platform do you use?” and from there, determines what app to recommend
- **Classifier 2**: asks the question “What is your age?” and from there, determines what app to recommend

Let’s look more carefully at the classifiers. The key observation follows: if we must recommend an app by asking only one question, our best bet is to look at all the people who answered with the same answer and recommend the most common app among them.

Classifier 1: What platform do you use?

- If the answer is “iPhone,” then we notice that of the iPhone users, the majority downloaded Check Mate Mate. Therefore, we recommend Check Mate Mate to all the iPhone users. We are correct **two times out of three**
- If the answer is “Android,” then we notice that of the Android users, the majority downloaded Atom Count, so that is the one we recommend to all the Android users. We are correct **two times out of three**.

Classifier 2: What is your age?

- If the answer is “young,” then we notice that all the young people downloaded Atom Count, so that is the recommendation we make. We are correct **three times out of three**.
- If the answer is “adult,” then we notice that of the adults, the majority downloaded Check Mate Mate, so we recommend that one. We are correct **two times out of three**.

Notice that classifier 1 is correct **four times out of six**, and classifier 2 is correct **five times out of six.** Therefore, for this dataset, classifier 2 is better. In figure 9.9, you can see the two classifiers with their accuracy. Notice that the questions are reworded so that they have yes-or-no answers, which doesn’t change the classifiers or the outcome.

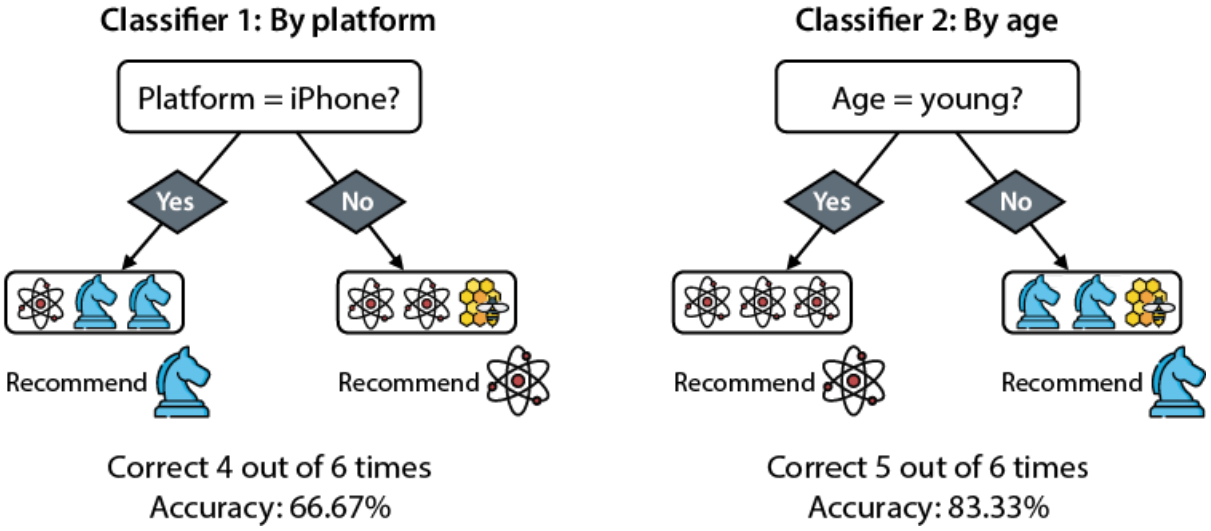


Figure 9.9 Classifier 1 uses platform, and classifier 2 uses age. To make the prediction at each leaf, each classifier picks the most common label among the samples in that leaf.

Classifier 1 is correct four out of six times, and classifier 2 is correct five out of six times. Therefore, based on accuracy, classifier 2 is better.

Gini impurity index: How diverse is my dataset?

The *Gini impurity index,* or *Gini index*, is another way we can compare the platform and age splits. The Gini index is a measure of diversity in a dataset. In other words, if we have a set in which all the elements are similar, this set has a low Gini index, and if all the elements are different, it has a large Gini index. For clarity, consider the following two sets of 10 colored balls (where any two balls of the same color are indistinguishable):

- **Set 1**: eight red balls, two blue balls
- **Set 2**: four red balls, three blue balls, two yellow balls, one green ball

Set 1 looks more pure than set 2, because set 1 contains mostly red balls and a couple of blue ones, whereas set 2 has many different colors. Next, we devise a measure of impurity that assigns a low value to set 1 and a high value to set 2. This measure of impurity relies on probability. Consider the following question:

If we pick two random elements of the set, what is the probability that they have a different color? The two elements don't need to be distinct; we are allowed to pick the same element twice.

For set 1, this probability is low, because the balls in the set have similar colors. For set 2, this probability is high, because the set is diverse, and if we pick two balls, they're likely to be of

different colors. Let's calculate these probabilities. First, notice that by the law of complementary probabilities (see the section "What the math just happened?" in chapter 8), the probability that we pick two balls of different colors is 1 minus the probability that we pick two balls of the same color:

P(picking two balls of different color) = 1 – P(picking two balls of the same color)

Now let's calculate the probability that we pick two balls of the same color. Consider a general set, where the balls have n colors. Let's call them color 1, color 2, all the way up to color n. Because the two balls must be of one of the n colors, the probability of picking two balls of the same color is the sum of probabilities of picking two balls of each of the n colors:

P(picking two balls of the same color) = P(both balls are color 1) + P(both balls are color 2) + … + P(both balls are color n)

What we used here is the sum rule for disjoint probabilities, that states the following:

sum rule for disjoint probabilities If two events E and F are disjoint, namely, they never occur at the same time, then the probability of either one of them happening (the union of the events) is the sum of the probabilities of each of the events. In other words,

$P(E \cup F) = P(E) + P(F)$

Now, let's calculate the probability that two balls have the same color, for each of the colors. Notice that we're picking each ball completely independently from the others. Therefore, by the product rule for independent probabilities (section "What the math just happened?" in chapter 8), the probability that both balls have color 1 is the square of the probability that we pick one ball and it is of color 1. In general, if p_i is the probability that we pick a random ball and it is of color i, then

P(both balls are color i) = p_i^2.

Putting all these formulas together (figure 9.10), we get that

P(picking two balls of different colors) = $1 - p_1^2 - p_2^2 - \ldots - p_n^2$.

This last formula is the Gini index of the set.

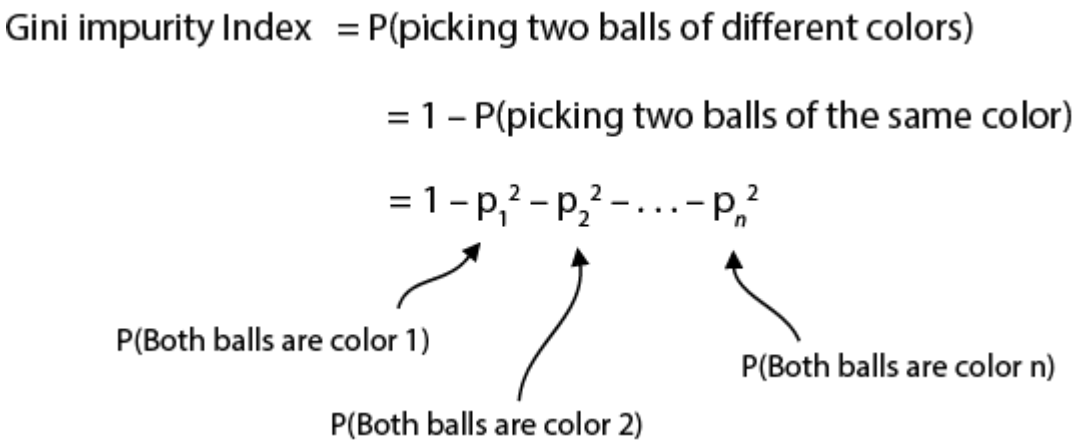


Figure 9.10 Summary of the calculation of the Gini impurity index

Finally, the probability that we pick a random ball of color *i* is the number of balls of color *i* divided by the total number of balls. This leads to the formal definition of the Gini index.

gini impurity index In a set with *m* elements and *n* classes, with a_i elements belonging to the *i*-th class, the Gini impurity index is

Gini = $1 - p_1^2 - p_2^2 - \ldots - p_n^2$,

where $p_i = a_i / m$. This can be interpreted as the probability that if we pick two random elements out of the set, they belong to different classes.

Now we can calculate the Gini index for both of our sets. For clarity, the calculation of the Gini index for set 1 is illustrated in figure 9.11 (with red and blue replaced by black and white).

Set 1: {red, red, red, red, red, red, red, red, blue, blue} (eight red balls, two blue balls)

$$\text{Gini impurity index} = 1 - \left(\frac{8}{10}\right)^2 - \left(\frac{2}{10}\right)^2 = 1 - \frac{68}{100} = 0.32$$

Set 2: {red, red, red, red, blue, blue, blue, yellow, yellow, green}

$$\text{Gini impurity index} = 1-\left(\frac{4}{10}\right)^2-\left(\frac{3}{10}\right)^2-\left(\frac{2}{10}\right)^2-\left(\frac{1}{10}\right)^2 = 1-\frac{30}{100}=0.7$$

Notice that, indeed, the Gini index of set 1 is larger than that of set 2.

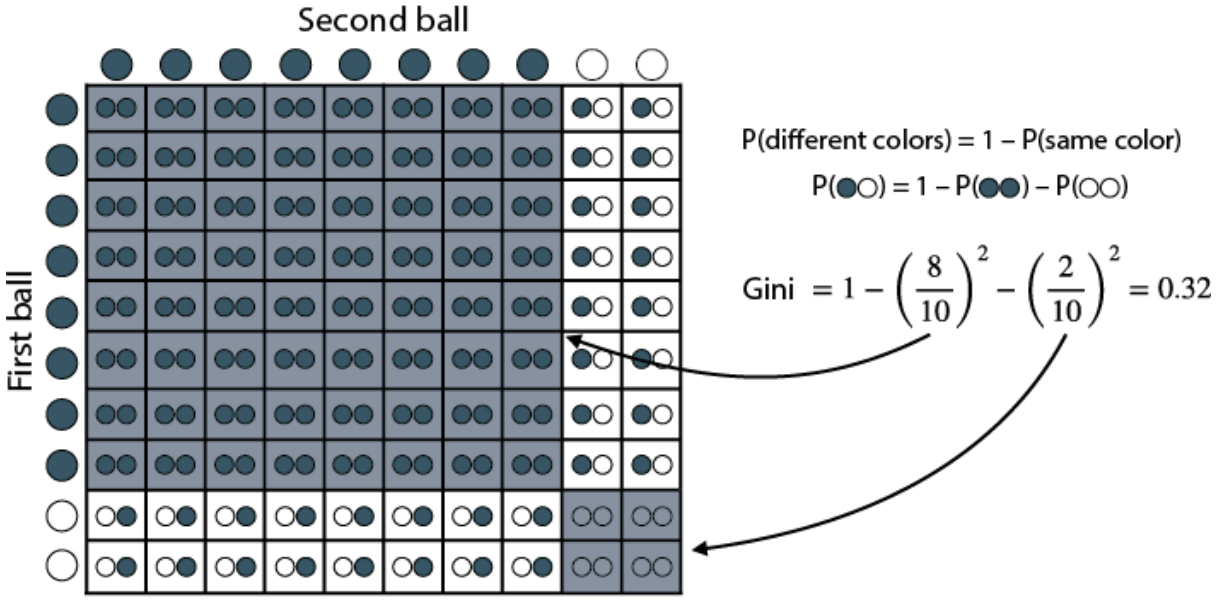


Figure 9.11 The calculation of the Gini index for the set with eight black balls and two white balls. Note that if the total area of the square is 1, the probability of picking two black balls is 0.8^2, and the probability of picking two white balls is 0.2^2 (these two are represented by the shaded squares). Thus, the probability of picking two balls of a different color is the remaining area, which is $1 – 0.8^2 – 0.2^2 =0.32$. That is the Gini index.

How do we use the Gini index to decide which of the two ways to split the data (age or platform) is better? Clearly, if we can split the data into two purer datasets, we have performed a better split. Thus, let's calculate the Gini index of the set of labels of each of the leaves. Looking at figure 9.12, here are the labels of the leaves (where we abbreviate each app by the first letter in its name):

Classifier 1 (by platform):

- Left leaf (iPhone): {A, C, C}
- Right leaf (Android): {A, A, B}

Classifier 2 (by age):

- Left leaf (young): {A, A, A}
- Right leaf (adult): {B, C, C}

The Gini indices of the sets {A, C, C}, {A, A, B}, and {B, C, C} are all the same: $1-\left(\frac{2}{3}\right)^2-\left(\frac{1}{3}\right)^2=0.444$. The Gini index of the set {A, A, A} is $1-\left(\frac{3}{3}\right)^2=0$. In general, the Gini index of a pure set is always 0. To measure the purity of the split, we average the Gini indices of the two leaves. Therefore, we have the following calculations:

Classifier 1 (by platform):

Average Gini index = 1/2(0.444+0.444) = 0.444

Classifier 2 (by age):

Average Gini index = 1/2(0.444+0) = 0.222

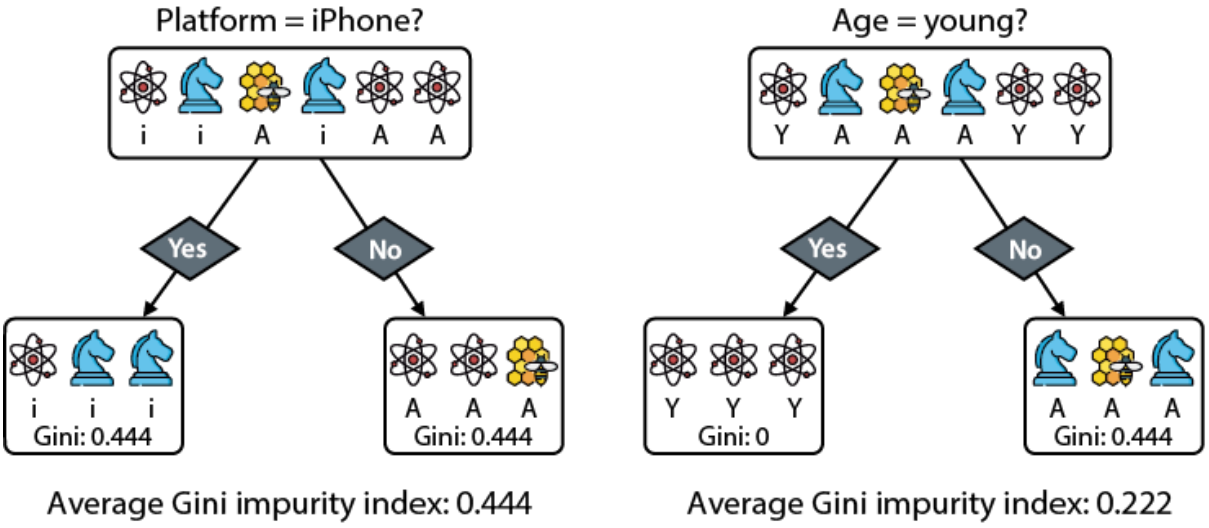


Figure 9.12 The two ways to split the dataset, by platform and age, and their Gini index calculations. Notice that splitting the dataset by age gives us two smaller datasets with a lower average Gini index. Therefore, we choose to split the dataset by age.

We conclude that the second split is better, because it has a lower average Gini index.

aside The Gini impurity index should not be confused with the Gini coefficient. The Gini coefficient is used in statistics to calculate the income or wealth inequality in countries. In this book, whenever we talk about the Gini index, we are referring to the Gini impurity index.

Entropy: Another measure of diversity with strong applications in information theory

In this section, we learn another measure of homogeneity in a set—its entropy—which is based on the physical concept of entropy and is highly important in probability and information theory. To understand entropy, we look at a slightly strange probability question. Consider the same two sets of colored balls as in the previous section, but think of the colors as an ordered set.

- **Set 1**: {red, red, red, red, red, red, red, red, blue, blue} (eight red balls, two blue balls)

- **Set 2**: {red, red, red, red, blue, blue, blue, yellow, yellow, green} (four red balls, three blue balls, two yellow balls, one green ball)

Now, consider the following scenario: we have set 1 inside a bag, and we start picking balls out of this bag and immediately return each ball we just picked back to the bag. We record the colors of the balls we picked. If we do this 10 times, imagine that we get the following sequence:

- Red, red, red, blue, red, blue, blue, red, red, red

Here is the main question that defines entropy:

What is the probability that, by following the procedure described in the previous paragraph, we get the exact sequence that defines set 1, which is {red, red, red, red, red, red, red, red, blue, blue}?

This probability is not very large, because we must be really lucky to get this sequence. Let's calculate it. We have eight red balls and two blue balls, so the probability that we get a red ball is $\frac{8}{10}$ and the probability that we get a blue ball is $\frac{2}{10}$. Because all the draws are independent, the probability that we get the desired sequence is

$$P(r, r, r, r, r, r, r, r, b, b) = \frac{8}{10}\cdot\frac{8}{10}\cdot\frac{8}{10}\cdot\frac{8}{10}\cdot\frac{8}{10}\cdot\frac{8}{10}\cdot\frac{8}{10}\cdot\frac{8}{10}\cdot\frac{2}{10}\cdot\frac{2}{10}$$

$$= \left(\frac{8}{10}\right)^8\left(\frac{2}{10}\right)^2 = 0.0067108864.$$

This is tiny, but can you imagine the corresponding probability for set 2? For set 2, we are picking balls out of a bag with four red balls, three blue balls, two yellow balls, and one green ball and hoping to obtain the following sequence:

- Red, red, red, red, blue, blue, blue, yellow, yellow, green.

This is nearly impossible, because we have many colors and not many balls of each color. This probability, which is calculated in a similar way, is

$$P(r, r, r, r, b, b, b, y, y, g) = \frac{4}{10}\cdot\frac{4}{10}\cdot\frac{4}{10}\cdot\frac{4}{10}\cdot\frac{3}{10}\cdot\frac{3}{10}\cdot\frac{3}{10}\cdot\frac{2}{10}\cdot\frac{2}{10}\cdot\frac{1}{10}$$

$$= \left(\frac{4}{10}\right)^4\left(\frac{3}{10}\right)^3\left(\frac{2}{10}\right)^2\left(\frac{1}{10}\right)^1 = 0.0000027648.$$

The more diverse the set, the more unlikely we'll be able to get the original sequence by picking one ball at a time. In contrast, the most pure set, in which all balls are of the same color, is easy to obtain this way. For example, if our original set has 10 red balls, each time

we pick a random ball, the ball is red. Thus, the probability of getting the sequence {red, red, red, red, red, red, red, red, red, red} is 1.

These numbers are very small for most cases—and this is with only 10 elements. Imagine if our dataset had one million elements. We would be dealing with tremendously small numbers. When we have to deal with really small numbers, using logarithms is the best method, because they provide a convenient way to write small numbers. For instance, 0.000000000000001 is equal to 10^{-15}, so its logarithm in base 10 is –15, which is a much nicer number to work with.

The entropy is defined as follows: we start with the probability that we recover the initial sequence by picking elements in our set, one at a time, with repetition. Then we take the logarithm, and divide by the total number of elements in the set. Because decision trees deal with binary decisions, we'll be using logarithms in base 2. The reason we took the negative of the logarithm is because logarithms of very small numbers are all negative, so we multiply by –1 to turn it into a positive number. Because we took a negative, the more diverse the set, the higher the entropy.

Now we can calculate the entropies of both sets and expand them using the following two identities:

- $log(ab) = log(a) + log(b)$
- $log(a^c) = c\ log(a)$

Set 1: {red, red, red, red, red, red, red, red, blue, blue} (eight red balls, two blue balls)

$$Entropy = -\frac{1}{10}\log_2 = \left[\left(\frac{8}{10}\right)^8\left(\frac{2}{10}\right)^2\right] = -\frac{8}{10}\log_2\left(\frac{8}{10}\right) - \frac{2}{10}\log_2\left(\frac{2}{10}\right) = 0.722$$

Set 2: {red, red, red, red, blue, blue, blue, yellow, yellow, green}

$$Entropy = -\frac{1}{10}\log_2\left[\left(\frac{4}{10}\right)^4\left(\frac{3}{10}\right)^3\left(\frac{2}{10}\right)^2\left(\frac{1}{10}\right)^1\right]$$

$$= -\frac{4}{10}\log_2\left(\frac{4}{10}\right) - \frac{3}{10}\log_2\left(\frac{3}{10}\right) - \frac{2}{10}\log_2\left(\frac{2}{10}\right) - \frac{1}{10}\log_2\left(\frac{1}{10}\right) = 1.846$$

Notice that the entropy of set 2 is larger than the entropy of set 1, which implies that set 2 is more diverse than set 1. The following is the formal definition of entropy:

entropy In a set with m elements and n classes, with a_i elements belonging to the i-th class, the entropy is

$$Entropy = -p_1 \log_2(p_1) - p_2 \log_2(p_2) - \cdots - p_n \log_2(p_n),$$

$$\text{where } p_i = \frac{a_i}{m}.$$

We can use entropy to decide which of the two ways to split the data (platform or age) is better in the same way as we did with the Gini index. The rule of thumb is that if we can split the data into two datasets with less combined entropy, we have performed a better split. Thus, let's calculate the entropy of the set of labels of each of the leaves. Again, looking at figure 9.12, here are the labels of the leaves (where we abbreviate each app by the first letter in its name):

Classifier 1 (by platform):

Left leaf: {A, C, C}

Right leaf: {A, A, B}

Classifier 2 (by age):

Left leaf: {A, A, A}

Right leaf: {B, C, C}

The entropies of the sets {A, C, C}, {A, A, B}, and {B, C, C} are all the same: $-\frac{2}{3}\log_2\left(\frac{2}{3}\right) - \frac{1}{3}\log_2\left(\frac{1}{3}\right)$. The entropy of the set {A, A, A} is $-\log\left(\frac{3}{3}\right) = -\log_2(1) = 0$. In general, the entropy of a set in which all elements are the same is always 0. To measure the purity of the split, we average the entropy of the sets of labels of the two leaves, as follows (illustrated in figure 9.13):

Classifier 1 (by platform):

Average entropy = 1/2(0.918 + 0.918) = 0.918

Classifier 2 (by age):

Average entropy = 1/2(0.918+0) = 0.459

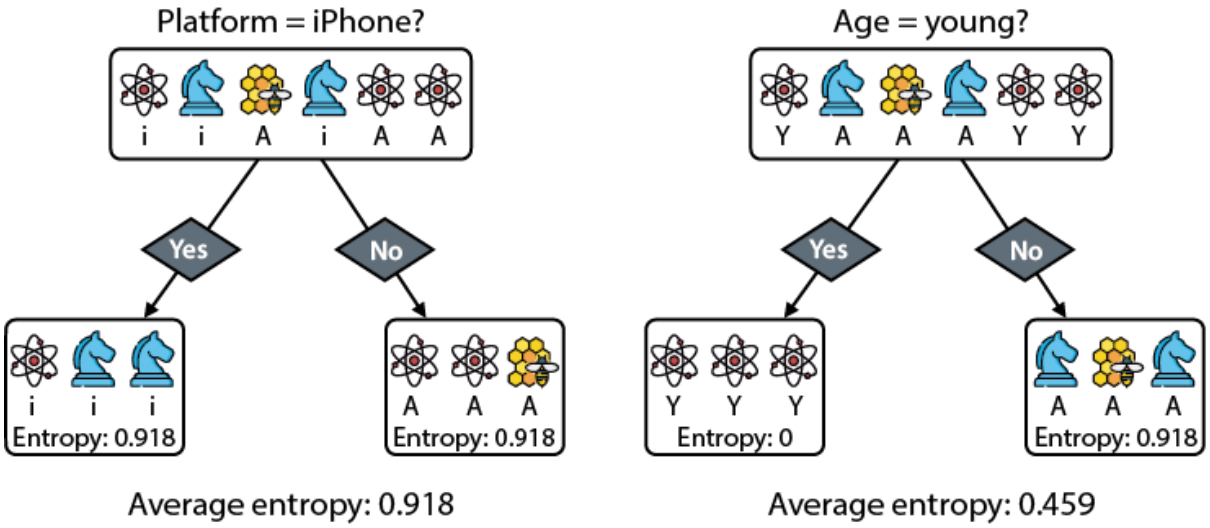


Figure 9.13 The two ways to split the dataset, by platform and age, and their entropy calculations. Notice that splitting the dataset by age gives us two smaller datasets with a lower average entropy. Therefore, we again choose to split the dataset by age.

Thus, again we conclude that the second split is better, because it has a lower average entropy.

Entropy is a tremendously important concept in probability and statistics, because it has strong connections with information theory, mostly thanks to the work of Claude Shannon. In fact, an important concept called *information gain* is precisely the change in entropy. To learn more on the topic, please see appendix C for a video and a blog post which covers this topic in much more detail.

Classes of different sizes? No problem: We can take weighted averages

In the previous sections we learned how to perform the best possible split by minimizing average Gini impurity index or entropy. However, imagine that you have a dataset with eight data points (which when training the decision tree, we also refer to as samples), and you split it into two datasets of sizes six and two. As you may imagine, the larger dataset should count for more in the calculations of Gini impurity index or entropy. Therefore, instead of considering the average, we consider the weighted average, where at each leaf, we assign the proportion of points corresponding to that leaf. Thus, in this case, we would weigh the first Gini impurity index (or entropy) by 6/8, and the second one by 2/8. Figure 9.14 shows an example of a weighted average Gini impurity index and a weighted average entropy for a sample split.

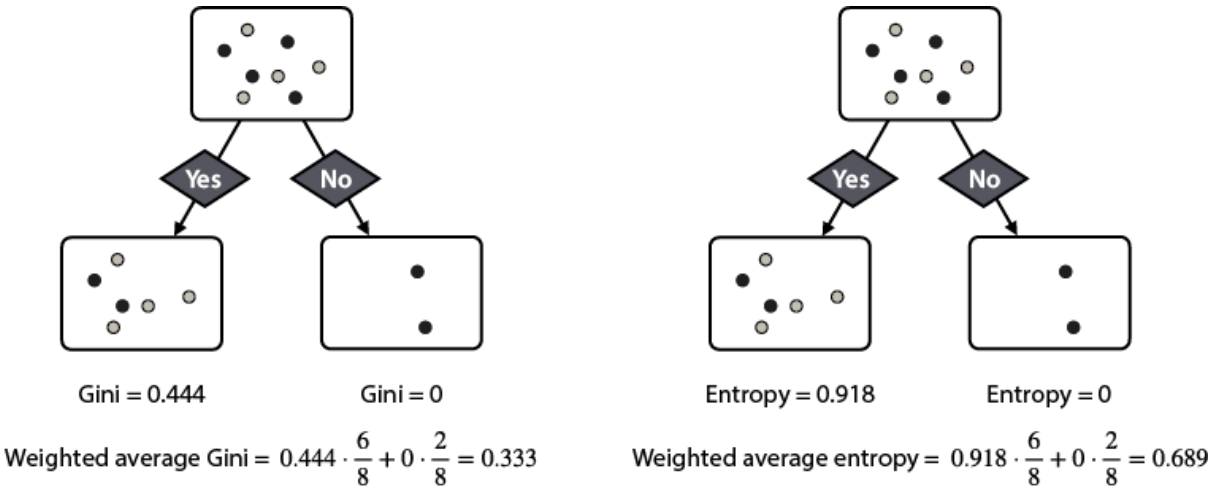


Figure 9.14 A split of a dataset of size eight into two datasets of sizes six and two. To calculate the average Gini index and the average entropy, we weight the index of the left dataset by 6/8 and that of the right dataset by 2/8. This results in a weighted Gini index of 0.333 and a weighted entropy of 0.689.

Now that we've learned three ways (accuracy, Gini index, and entropy) to pick the best split, all we need to do is iterate this process many times to build the decision tree! This is detailed in the next section.

Second step to build the model: Iterating

In the previous section, we learned how to split the data in the best possible way using one of the features. That is the bulk of the training process of a decision tree. All that is left to finish building our decision tree is to iterate on this step many times. In this section we learn how to do this.

Using the three methods, accuracy, Gini index, and entropy, we decided that the best split was made using the "age" feature. Once we make this split, our dataset is divided into two datasets. The split into these two datasets, with their accuracy, Gini index, and entropy, is illustrated in figure 9.15.

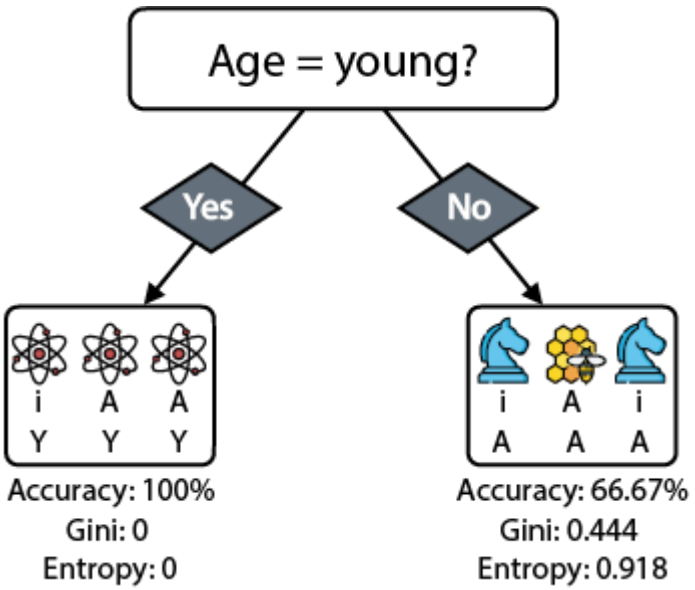


Figure 9.15 When we split our dataset by age, we get two datasets. The one on the left has three users who downloaded Atom Count, and the one on the right has one user who downloaded Beehive Count and two who downloaded Check Mate Mate.

Notice that the dataset on the left is pure—all the labels are the same, its accuracy is 100%, and its Gini index and entropy are both 0. There's nothing more we can do to split this dataset or to improve the classifications. Thus, this node becomes a leaf node, and when we get to that leaf, we return the prediction "Atom Count."

The dataset on the right can still be divided, because it has two labels: "Beehive Count" and "Check Mate Mate." We've used the age feature already, so let's try using the platform feature. It turns out that we're in luck, because the Android user downloaded Beehive Count, and the two iPhone users downloaded Check Mate Mate. Therefore, we can split this leaf using the platform feature and obtain the decision node shown in figure 9.16.

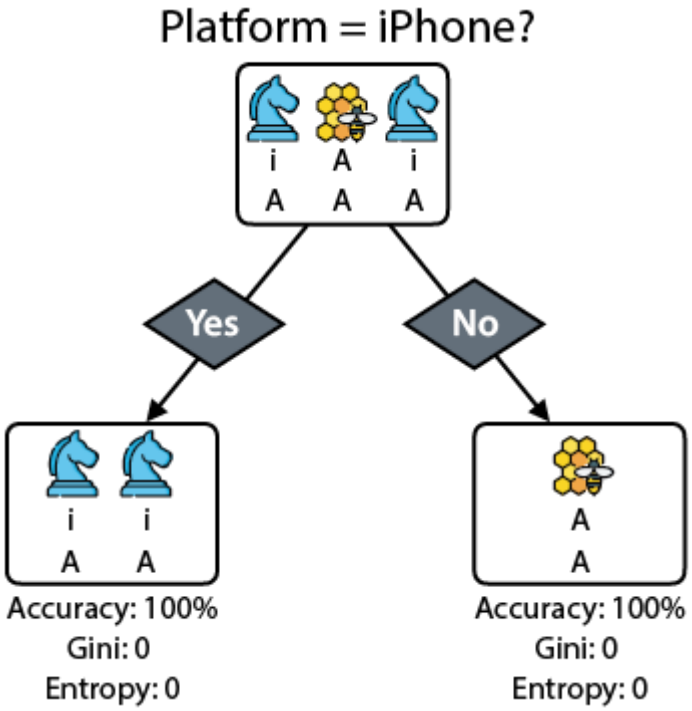


Figure 9.16 We can split the right leaf of the tree in figure 9.15 using platform and obtain two pure datasets. Each one of them has an accuracy of 100% and a Gini index and entropy of 0.

After this split, we are done, because we can't improve our splits any further. Thus, we obtain the tree in figure 9.17.

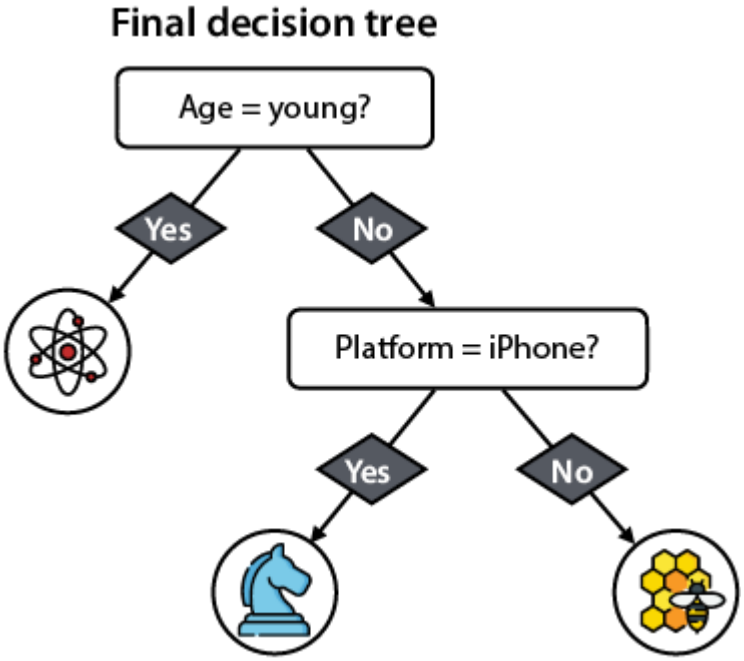


Figure 9.17 The resulting decision tree has two nodes and three leaves. This tree predicts every point in the original dataset correctly.

This is the end of our process, and we have built a decision tree that classifies our entire dataset. We almost have all the pseudocode for the algorithm, except for some final details which we see in the next section.

Last step: When to stop building the tree and other hyperparameters

In the previous section, we built a decision tree by recursively splitting our dataset. Each split was performed by picking the best feature to split. This feature was found using any of the following metrics: accuracy, Gini index, or entropy. We finish when the portion of the dataset corresponding to each of the leaf nodes is pure—in other words, when all the samples on it have the same label.

Many problems can arise in this process. For instance, if we continue splitting our data for too long, we may end up with an extreme situation in which every leaf contains very few samples, which can lead to serious overfitting. The way to prevent this is to introduce a stopping condition. This condition can be any of the following:

1. Don't split a node if the change in accuracy, Gini index, or entropy is below some threshold.
2. Don't split a node if it has less than a certain number of samples.
3. Split a node only if both of the resulting leaves contain at least a certain number of samples.
4. Stop building the tree after you reach a certain depth.

All of these stopping conditions require a hyperparameter. More specifically, these are the hyperparameters corresponding to the previous four conditions:

1. The minimum amount of change in accuracy (or Gini index, or entropy)
2. The minimum number of samples that a node must have to split it
3. The minimum number of samples allowed in a leaf node
4. The maximum depth of the tree

The way we pick these hyperparameters is either by experience or by running an exhaustive search where we look for different combinations of hyperparameters and choose the one that performs best in our validation set. This process is called *grid search*, and we'll study it in more detail in the section "Tuning the hyperparameters to find the best model: Grid search" in chapter 13.

The decision tree algorithm: How to build a decision tree and make predictions with it

Now we are finally ready to state the pseudocode for the decision tree algorithm, which allows us to train a decision tree to fit a dataset.

Pseudocode for the decision tree algorithm

Inputs:

- A training dataset of samples with their associated labels
- A metric to split the data (accuracy, Gini index, or entropy)
- One (or more) stopping condition

Output:

- A decision tree that fits the dataset

Procedure:

- Add a root node, and associate it with the entire dataset. This node has level 0. Call it a leaf node.
- Repeat until the stopping conditions are met at every leaf node:
    - Pick one of the leaf nodes at the highest level.
    - Go through all the features, and select the one that splits the samples corresponding to that node in an optimal way, according to the selected metric. Associate that feature to the node.
    - This feature splits the dataset into two branches. Create two new leaf nodes, one for each branch, and associate the corresponding samples to each of the nodes.
    - If the stopping conditions allow a split, turn the node into a decision node, and add two new leaf nodes underneath it. If the level of the node is i, the two new leaf nodes are at level $i + 1$.
    - If the stopping conditions don't allow a split, the node becomes a leaf node. To this leaf node, associate the most common label among its samples. That label is the prediction at the leaf.

Return:

- The decision tree obtained.

To make predictions using this tree, we simply traverse down it, using the following rules:

- Traverse the tree downward. At every node, continue in the direction that is indicated by the feature.
- When arriving at a leaf, the prediction is the label associated with the leaf (the most common among the samples associated with that leaf in the training process).

This is how we make predictions using the app-recommendation decision tree we built previously. When a new user comes, we check their age and their platform, and take the following actions:

- If the user is young, then we recommend them Atom Count.
- If the user is an adult, then we check their platform.
    - If the platform is Android, then we recommend Beehive Count.
    - If the platform is iPhone, then we recommend Check Mate Mate.

aside The literature contains terms like Gini gain and information gain when training decision trees. The Gini gain is the difference between the weighted Gini impurity index of the leaves and the Gini impurity index (entropy) of the decision node we are splitting. In a similar way, the information gain is the difference between the weighted entropy of the leaves and the entropy of the root. The more common way to train decision trees is by maximizing the Gini gain or the information gain. However, in this chapter, we train decision trees by, instead,

minimizing the weighted Gini index or the weighted entropy. The training process is exactly the same, because the Gini impurity index (entropy) of the decision node is constant throughout the process of splitting that particular decision node.

Beyond questions like yes/no

In the section “The solution: Building an app-recommendation system,” we learned how to build a decision tree for a very specific case in which every feature was categorical and binary (meaning that it has only two classes, such as the platform of the user). However, almost the same algorithm works to build a decision tree with categorical features with more classes (such as dog/cat/bird) and even with numerical features (such as age or average income). The main step to modify is the step in which we split the dataset, and in this section, we show you how.

Splitting the data using non-binary categorical features, such as dog/cat/bird

Recall that when we want to split a dataset based on a binary feature, we simply ask one yes-or-no question of the form, “Is the feature X?” For example, when the feature is the platform, a question to ask is “Is the user an iPhone user?” If we have a feature with more than two classes, we just ask several questions. For example, if the input is an animal that could be a dog, a cat, or a bird, then we ask the following questions:

- Is the animal a dog?
- Is the animal a cat?
- Is the animal a bird?

No matter how many classes a feature has, we can split it into several binary questions (figure 9.18).

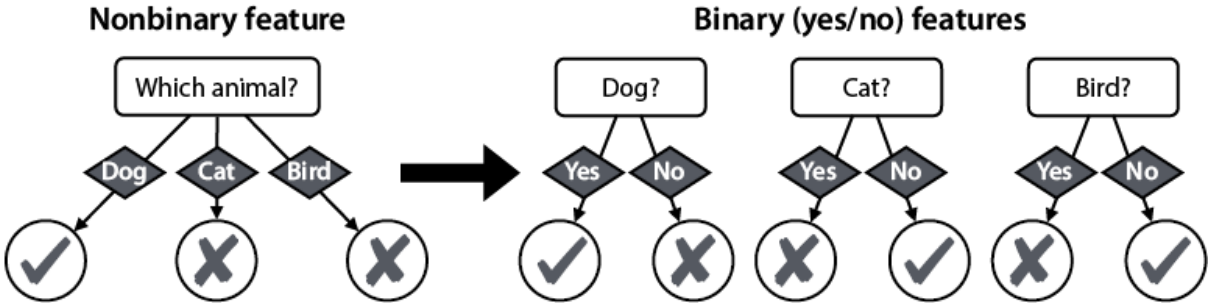


Figure 9.18 When we have a nonbinary feature, for example, one with three or more possible categories, we instead turn it into several binary (yes-or-no) features, one for each category. For example, if the feature is a dog, the answers to the three questions “Is it a dog?,” “Is it a cat?,” and “Is it a bird?” are “yes,” “no,” and “no.”

Each of the questions splits the data in a different way. To figure out which of the three questions gives us the best split, we use the same methods as in the section “First step to build the model”: accuracy, Gini index, or entropy. This process of turning a nonbinary categorical feature into several binary features is called *one-hot encoding*. In the section “Turning categorical data into numerical data” in chapter 13, we see it used in a real dataset.

Splitting the data using continuous features, such as age

Recall that before we simplified our dataset, the “age” feature contained numbers. Let’s get back to our original table and build a decision tree there (table 9.3).

Table 9.3 Our original app recommendation dataset with the platform and (numerical) age of the users. This is the same as table 9.1.

Platform	Age	App
iPhone	15	Atom Count
iPhone	25	Check Mate Mate
Android	32	Beehive Finder
iPhone	35	Check Mate Mate
Android	12	Atom Count
Android	14	Atom Count

The idea is to turn the Age column into several questions of the form, “Is the user younger than X?” or “Is the user older than X?” It seems like we have infinitely many questions to ask, because there are infinitely many numbers, but notice that many of these questions split the data in the same way. For example, asking, “Is the user younger than 20?” and “Is the user younger than 21,” gives us the same split. In fact, only seven splits are possible, as illustrated in figure 9.19.

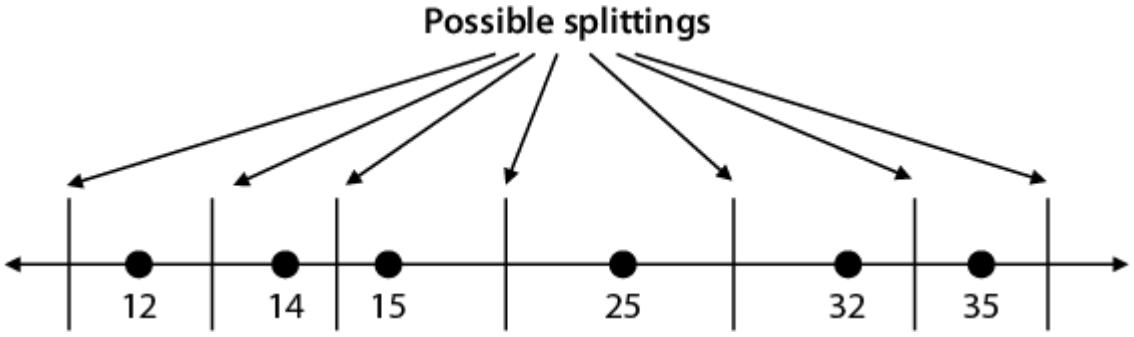


Figure 9.19 A graphic of the seven possible ways to split the users by age. Note that it doesn’t matter where we put the cutoffs, as long as they lie between consecutive ages (except for the first and last cutoff).

As a convention, we’ll pick the midpoints between consecutive ages to be the age for splitting. For the endpoints, we can pick any random value that is out of the interval. Thus, we have seven possible questions that split the data into two sets, as shown in table 9.4. In this table, we have also calculated the accuracy, the Gini impurity index, and the entropy of each of the splits.

Notice that the fourth question (“Is the user younger than 20?”) gives the highest accuracy, the lowest weighted Gini index, and the lowest weighted entropy and, therefore, is the best split that can be made using the “age” feature.

Table 9.4 The seven possible questions we can pick, each with the corresponding splitting. In the first set, we put the users who are younger than the cutoff, and in the second set, those who are older than the cutoff.

Question	First set (yes)	Second set (no)	Labels	Weighted accuracy	Weighted Gini impurity index	Weighted entropy
Is the user younger than 7?	empty	12, 14, 15, 25, 32, 35	{}, {A,A,A,C,B,C}	3/6	0.611	1.459
Is the user younger than 13?	12	14, 15, 25, 32, 35	{A}, {A,A,C,B,C}	3/6	0.533	1.268
Is the user younger than 14.5?	12, 14	15, 25, 32, 35	{A,A} {A,C,B,C}	4/6	0.417	1.0
Is the user younger than 20?	12, 14, 15	25, 32, 35	{A,A,A}, {C,B,C}	5/6	0.222	0.459
Is the user younger than 28.5?	12, 14, 15, 25	32, 35	{A,A,A,C}, {B,C}	4/6	0.416	0.874
Is the user younger than 33.5?	12, 14, 15, 25, 32	35	{A,A,A,C,B}, {C}	4/6	0.467	1.145

Is the user younger than 100?	12, 14, 15, 25, 32, 35	empty	{A,A,A,C,B,C}, {}	3/6	0.611	1/459

Carry out the calculations in the table, and verify that you get the same answers. The entire calculation of these Gini indices is in the following notebook: https://github.com/luisguiserrano/manning/blob/master/Chapter_9_Decision_Trees/Gini_entropy_calculations.ipynb.

For clarity, let's carry out the calculations of accuracy, weighted Gini impurity index, and weighted entropy for the third question. Notice that this question splits the data into the following two sets:

- **Set 1** (younger than 14.5)
  - Ages: 12, 14
  - Labels: {A, A}
- **Set 2** (14.5 and older):
  - Ages: 15, 25, 32, 25
  - Labels: {A, C, B, C}

Accuracy calculation

The most common label in set 1 is "A" and in set 2 is "C," so these are the predictions we'll make for each of the corresponding leaves. In set 1, every element is predicted correctly, and in set 2, only two elements are predicted correctly. Therefore, this decision stump is correct in four out of the six data points, for an accuracy of 4/6 = 0.667.

For the next two calculations, notice the following:

- Set 1 is pure (all its labels are the same), so its Gini impurity index and entropy are both 0.
- In set 2, the proportions of elements with labels "A," "B," and "C" are 1/4, 1/4, and 2/4 =1/2, respectively.

Weighted Gini impurity index calculation

The Gini impurity index of the set {A, A} is 0.

The Gini impurity index of the set {A, C, B, C} is $1-\left(\frac{1}{4}\right)^2-\left(\frac{1}{4}\right)^2-\left(\frac{1}{2}\right)^2=0.625$

The weighted average of the two Gini impurity indices is $\frac{2}{6}\cdot 0+\frac{4}{6}\cdot 0.625=0.417$

Accuracy calculation

The entropy of the set {A, A} is 0.

The entropy of the set {A, C, B, C} is $-\frac{1}{4}\log_2\left(\frac{1}{4}\right)-\frac{1}{4}\log_2\left(\frac{1}{4}\right)-\frac{1}{2}\log_2\left(\frac{1}{2}\right)=1.5$

The weighted average of the two entropies is $\frac{2}{6}\cdot 0+\frac{4}{6}\cdot 1.5=1.0$

A numerical feature becomes a series of yes-or-no questions, which can be measured and compared with the other yes-or-no questions coming from other features, to pick the best one for that decision node.

aside This app-recommendation model is very small, so we could do it all by hand. However, to see it in code, please check this notebook: https://github.com/luisguiserrano/manning/blob/master/Chapter_9_Decision_Trees/App_recommendations.ipynb. The notebook uses the Scikit-Learn package, which we introduce in more detail in the section “Using Scikit-Learn to build a decision tree.”

The graphical boundary of decision trees

In this section, I show you two things: how to build a decision tree geometrically (in two dimensions) and how to code a decision tree in the popular machine learning package Scikit-Learn.

Recall that in classification models, such as the perceptron (chapter 5) or the logistic classifier (chapter 6), we plotted the boundary of the model that separated the points with labels 0 and 1, and it turned out to be a straight line. The boundary of a decision tree is also nice, and when the data is two-dimensional, it is formed by a combination of vertical and horizontal lines. In this section, we illustrate this with an example. Consider the dataset in figure 9.20, where the points with label 1 are triangles, and the points with label 0 are squares. The horizontal and vertical axes are called x_0 and x_1, respectively.

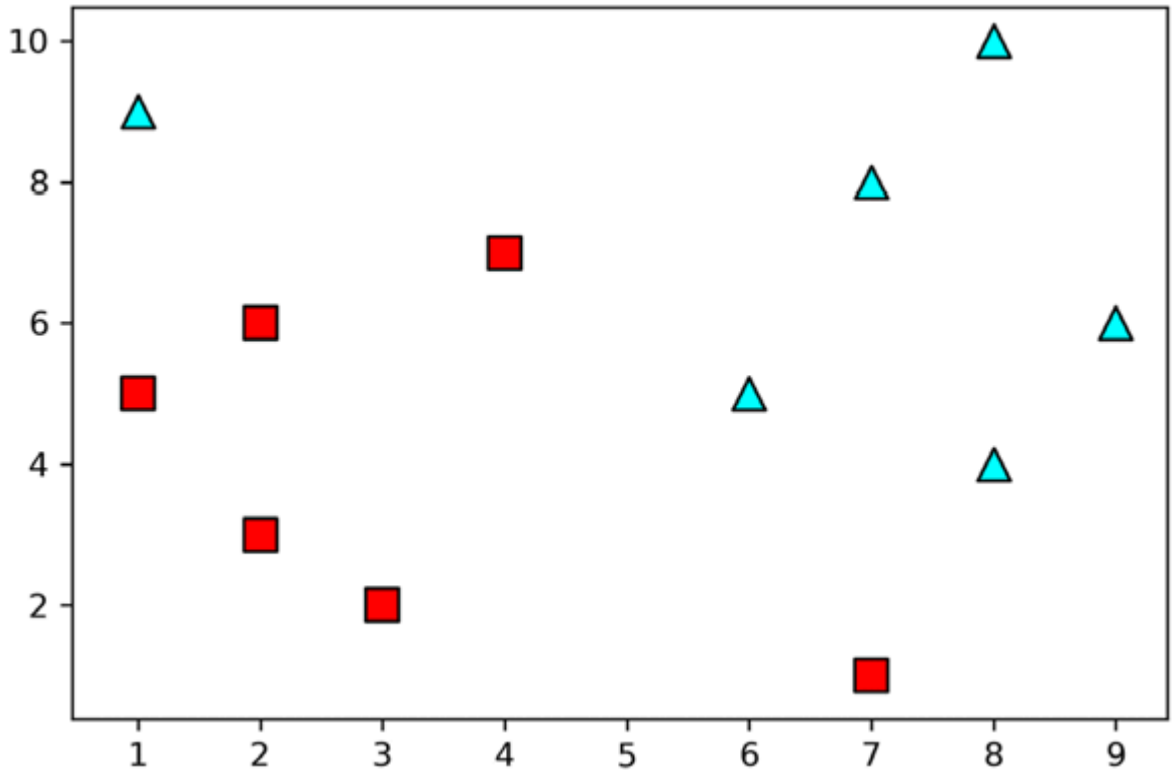


Figure 9.20 A dataset with two features (x_0 and x_1) and two labels (triangle and square) in which we will train a decision tree

If you had to split this dataset using only one horizontal or vertical line, what line would you pick? There could be different lines, according to the criteria you would use to measure the effectiveness of a solution. Let's go ahead and select a vertical line at $x_0 = 5$. This leaves mostly triangles to the right of it and mostly squares to the left of it, with the exception of two misclassified points, one square and one triangle (figure 9.21). Try checking all the other possible vertical and horizontal lines, compare them using your favorite metric (accuracy, Gini index, and entropy), and verify that this is the line that best divides the points.

Now let's look at each half separately. This time, it's easy to see that two horizontal lines at $x_1 = 8$ and $x_1 = 2.5$ will do the job on the left and the right side, respectively. These lines completely divide the dataset into squares and triangles. Figure 9.22 illustrates the result.

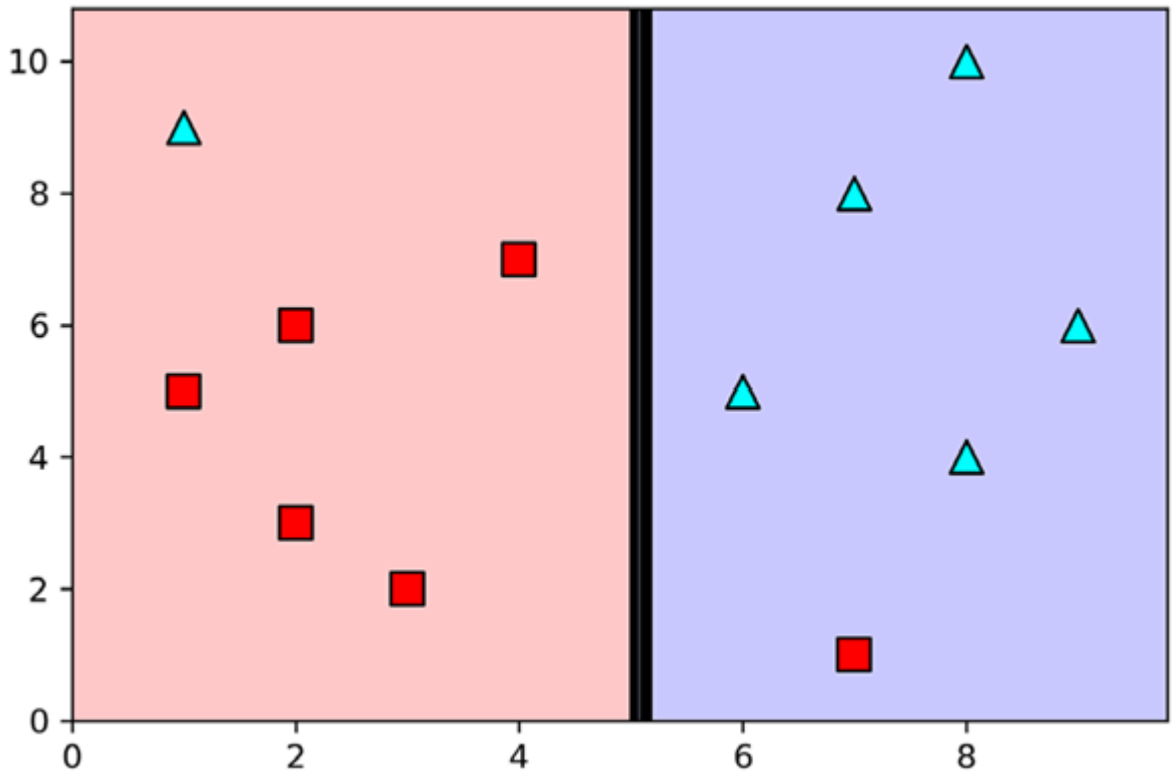


Figure 9.21 If we have to use only one vertical or horizontal line to classify this dataset in the best possible way, which one would we use? Based on accuracy, the best classifier is the vertical line at $x_0 = 5$, where we classify everything to the right of it as a triangle, and everything to the left of it as a square. This simple classifier classifies 8 out of the 10 points correctly, for an accuracy of 0.8.

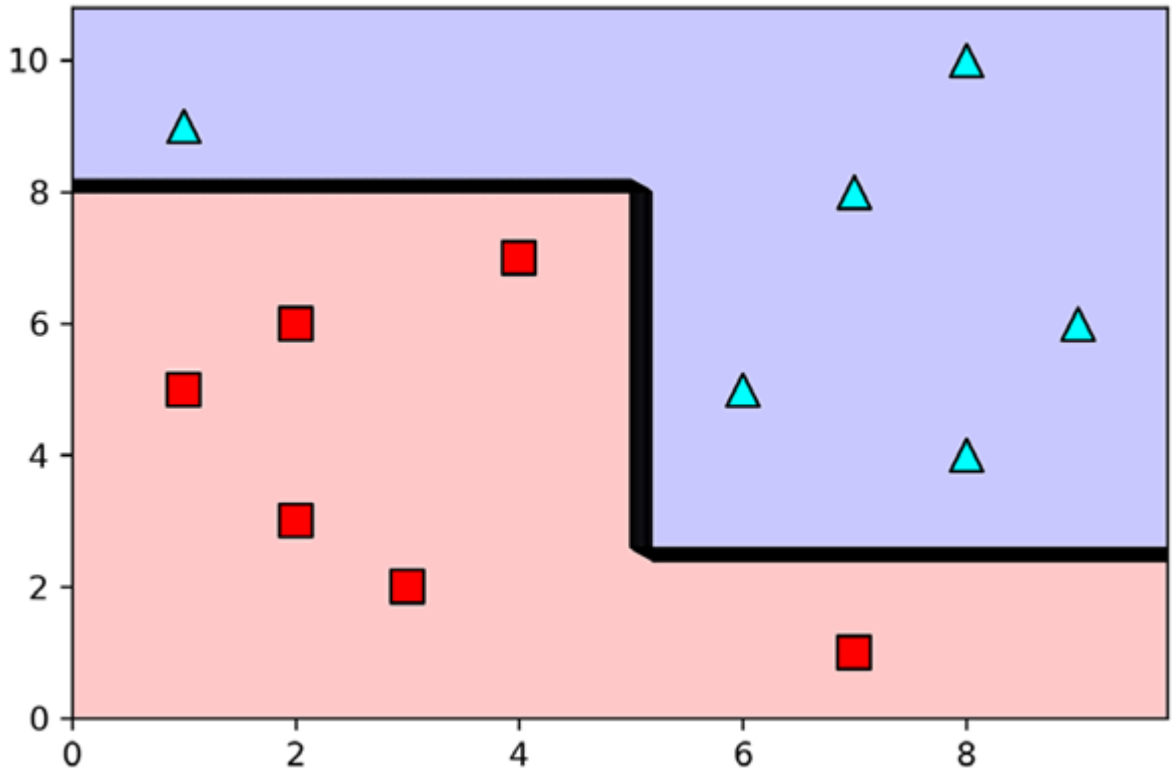


Figure 9.22 The classifier in figure 9.21 leaves us with two datasets, one at each side of the vertical line. If we had to classify each one of them, again using one vertical or horizontal line, which one would we choose? The best choices are horizontal lines at $x_1 = 8$ and $x_1 = 2.5$, as the figure shows.

What we did here was build a decision tree. At every stage, we picked from each of the two features (x_0 and x_1) and selected the threshold that best splits our data. In fact, in the next subsection, we use Scikit-Learn to build the same decision tree on this dataset.

Using Scikit-Learn to build a decision tree

In this section, we learn how to use a popular machine learning package called Scikit-Learn (abbreviated sklearn) to build a decision tree. The code for this section follows:

- **Notebook**: Graphical_example.ipynb
    - https://github.com/luisguiserrano/manning/blob/master/Chapter_9_Decision_Trees/Graphical_example.ipynb

We begin by loading the dataset as a Pandas DataFrame called `dataset` (introduced in chapter 8), with the following lines of code:

```
import pandas as pd
dataset = pd.DataFrame({
    'x_0':[7,3,2,1,2,4,1,8,6,7,8,9],
    'x_1':[1,2,3,5,6,7,9,10,5,8,4,6],
    'y': [0,0,0,0,0,0,1,1,1,1,1,1]})
```

Now we separate the features from the labels as shown here:

```
features = dataset[['x_0', 'x_1']]
labels = dataset['y']
```

To build the decision tree, we create a `DecisionTreeClassifier` object and use the `fit` function, as follows:

```
decision_tree = DecisionTreeClassifier()
decision_tree.fit(features, labels)
```

We obtained the plot of the tree, shown in figure 9.23, using the `display_tree` function in the utils.py file.

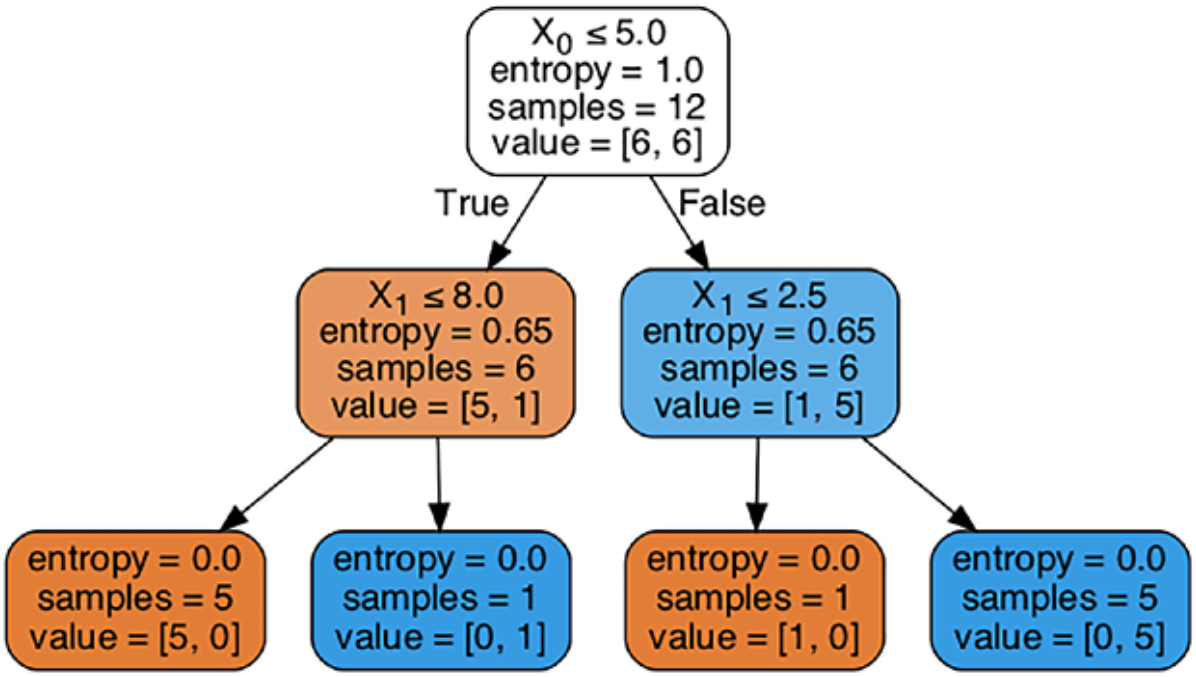


Figure 9.23 The resulting decision tree of depth 2 that corresponds to the boundary in figure 9.22. It has three nodes and four leaves.

Notice that the tree in figure 9.23 corresponds precisely to the boundary in figure 9.22. The root node corresponds to the first vertical line at $x_0 = 5$, with the points at each side of the line corresponding to the two branches. The two horizontal lines at $x_1 = 8.0$ and $x_1 = 2.5$ on the left and right halves of the plot correspond to the two branches. Furthermore, at each node we have the following information:

- **Gini**: the Gini impurity index of the labels at that node
- **Samples**: the number of data points (samples) corresponding to that node
- **Value**: the number of data points of each of the two labels at that node

As you can see, this tree has been trained using the Gini index, which is the default in Scikit-Learn. To train it using entropy, we can specify it when building the `DecisionTree` object, as follows:

```
decision_tree = DecisionTreeClassifier(criterion='entropy')
```

We can specify more hyperparameters when training the tree, which we see in the next section with a much bigger example.

Real-life application: Modeling student admissions with Scikit-Learn

In this section, we use decision trees to build a model that predicts admission to graduate schools. The dataset can be found in Kaggle (see appendix C for the link). As in the section

“The graphical boundary of decision trees,” we’ll use Scikit-Learn to train the decision tree and Pandas to handle the dataset. The code for this section follows:

- **Notebook**: University_admissions.ipynb
    - https://github.com/luisguiserrano/manning/blob/master/Chapter_9_Decision_Trees/University_Admissions.ipynb
- **Dataset**: Admission_Predict.csv

The dataset has the following features:

- **GRE score**: a number out of 340
- **TOEFL score**: a number out of 120
- **University rating**: a number from 1 to 5
- **Statement of purpose strength (SOP)**: a number from 1 to 5
- **Undergraduate grade point average (CGPA)**: a number from 1 to 10
- **Letter of recommendation strength (LOR)**: a number from 1 to 5
- **Research experience**: Boolean variable (0 or 1)

The labels on the dataset are the chance of admission, which is a number between 0 and 1. To have binary labels, we’ll consider every student with a chance of 0.75 or higher as “admitted,” and any other student as “not admitted.”

The code for loading the dataset into a Pandas DataFrame and performing this preprocessing step is shown next:

```
import pandas as pd
data = pd.read_csv('Admission_Predict.csv', index_col=0)
data['Admitted'] = data['Chance of Admit'] >= 0.75
data = data.drop(['Chance of Admit'], axis=1)
```

The first few rows of the resulting dataset are shown in table 9.5.

Table 9.5 A dataset with 400 students and their scores in standardized tests, grades, university ratings, letters of recommendations, statements of purpose, and information about their chances of being admitted to graduate school

GRE score	TOEFL score	University rating	SOP	LOR	CGPA	Research	Admitted
337	118	4	4.5	4.5	9.65	1	True
324	107	4	4.0	4.5	8.87	1	True
316	104	3	3.0	3.5	8.00	1	False
322	110	3	3.5	2.5	8.67	1	True
314	103	2	2.0	3.0	8.21	0	False

As we saw in the section “The graphical boundary of decision trees,” Scikit-Learn requires that we enter the features and the labels separately. We’ll build a Pandas DataFrame called `features` containing all the columns except the Admitted column, and a Pandas Series called `labels` containing only the Admitted column. The code follows:

```
features = data.drop(['Admitted'], axis=1)
labels = data['Admitted']
```

Now we create a `DecisionTreeClassifier` object (which we call `dt`) and use the `fit` method. We’ll train it using the Gini index, as shown next, so there is no need to specify the `criterion` hyperparameter, but go ahead and train it with entropy and compare the results with those that we get here:

```
from sklearn.tree import DecisionTreeClassifier
dt = DecisionTreeClassifier()
dt.fit(features, labels)
```

To make predictions, we can use the `predict` function. For example, here is how we make predictions for the first five students:

```
dt.predict(features[0:5])
```
Output: array([True, True, False, True, False])

However, the decision tree we just trained massively overfits. One way to see this is by using the `score` function and realizing that it scores 100% in the training set. In this chapter, we won’t test the model, but will try building a testing set and verifying that this model overfits. Another way to see the overfitting is to plot the tree and notice that its depth is 10 (see the notebook). In the next section, we learn about some hyperparameters that help us prevent overfitting.

Setting hyperparameters in Scikit-Learn

To prevent overfitting, we can use some of the hyperparameters that we learned in the section “Last step: When to stop building the tree and other hyperparameters,” such as the following:

- `max_depth`: the maximum allowed depth.
- `max_features`: the maximum number of features considered at each split (useful for when there are too many features, and the training process takes too long).
- `min_impurity_decrease`: the decrease in impurity must be higher than this threshold to split a node.
- `min_impurity_split`: when the impurity at a node is lower than this threshold, the node becomes a leaf.
- `min_samples_leaf`: the minimum number of samples required for a leaf node. If a split leaves a leaf with less than this number of samples, the split is not performed.
- `min_samples_split`: the minimum number of samples required to split a node.

Play around with these parameters to find a good model. We’ll use the following:

- `max_depth = 3`
- `min_samples_leaf = 10`
- `min_samples_split = 10`

dt_smaller = DecisionTreeClassifier(max_depth=3, min_samples_leaf=10, min_samples_split=10)
dt_smaller.fit(features, labels)

The resulting tree is illustrated in figure 9.24. Note that in this tree, all the edges to the right correspond to “False” and to the left to “True.”

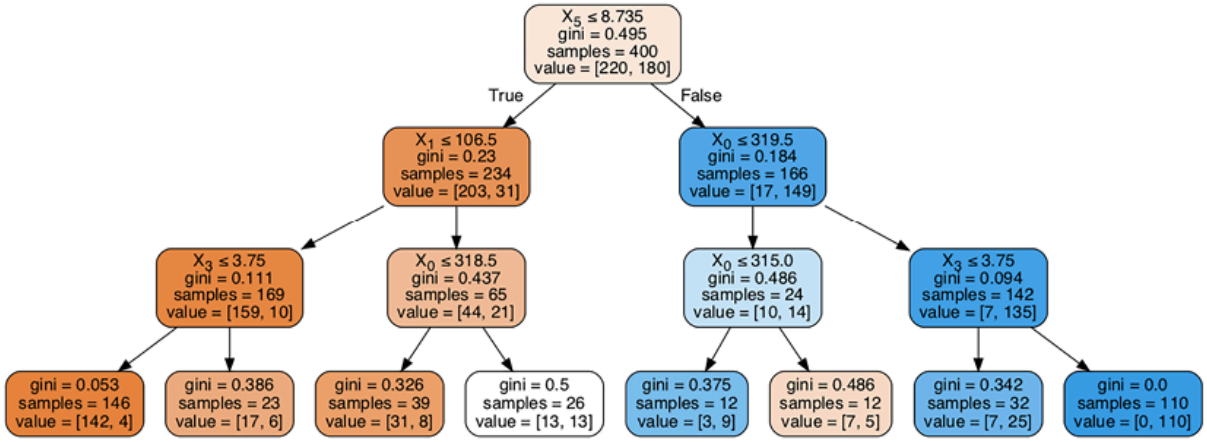


Figure 9.24 A decision tree of depth 3 trained in the student admissions dataset

The prediction given at each of the leaves is the label corresponding to the majority of the nodes in that leaf. In the notebook, each node has a color assigned to it, ranging from orange to blue. The orange nodes are those with more points with label 0, and the blue nodes are those with label 1. Notice that the white leaf, in which there are the same number of points with labels 0 and 1. For this leaf, any prediction has the same performance. In this case, Scikit-Learn defaults to the first class in the list, which in this case is false.

To make a prediction, we use the `predict` function. For example, let’s predict the admission for a student with the following numbers:

- GRE score: 320
- TOEFL score: 110
- University rating: 3
- SOP: 4.0
- LOR: 3.5
- CGPA: 8.9
- Research: 0 (no research)

dt_smaller.predict([[320, 110, 3, 4.0, 3.5, 8.9, 0]])
Output: array([True])

The tree predicts that the student will be admitted.

From this tree, we can infer the following things about our dataset:

- The most important feature is the sixth column (X_5), corresponding to the CGPA, or the grades. The cutoff grade is 8.735 out of 10. In fact, most of the predictions to the right of the root node are “admit” and to the left are “not admit,” which implies that CGPA is a very strong feature.
- After this feature, the two most important ones are GRE score (X_0) and TOEFL score (X_1), both standardized tests. In fact, among the students who got good grades, most of them are likely to be admitted, unless they did poorly on the GRE, as accounted for by the sixth leaf from the left in the tree in figure 9.24.
- Aside from grades and standardized tests, the only other feature appearing in the tree is SOP, or the strength of the statement of purpose. This is located down in the tree, and it didn’t change the predictions much.

Recall, however, that the construction of the tree is greedy in nature, namely, at each point it selects the top feature. This doesn’t guarantee that the choice of features is the best, however. For example, there could be a combination of features that is very strong, yet none of them is strong individually, and the tree may not be able to pick this up. Thus, even though we got some information about the dataset, we should not yet throw away the features that are not present in the tree. A good feature selection algorithm, such as L1 regularization, would come in handy when selecting features in this dataset.

Decision trees for regression

In most of this chapter, we’ve used decision trees for classification, but as was mentioned earlier, decision trees are good regression models as well. In this section, we see how to build a decision tree regression model. The code for this section follows:

- **Notebook**: Regression_decision_tree.ipynb
    - https://github.com/luisguiserrano/manning/blob/master/Chapter_9_Decision_Trees/Regression_decision_tree.ipynb

Consider the following problem: we have an app, and we want to predict the level of engagement of the users in terms of how many days per week they used it. The only feature we have is the user’s age. The dataset is shown in table 9.6, and its plot is in figure 9.25.

Table 9.6 A small dataset with eight users, their age, and their engagement with our app. The engagement is measured in the number of days when they opened the app in one week.

Age	Engagement
10	7
20	5
30	7

40	1
50	2
60	1
70	5
80	4

From this dataset, it seems that we have three clusters of users. The young users (ages 10, 20, 30) use the app a lot, the middle-aged users (ages 40, 50, 60) don't use it very much, and the older users (ages 70, 80) use it sometimes. Thus, a prediction like this one would make sense:

- If the user is 34 years old or younger, the engagement is 6 days per week.
- If the user is between 35 and 64, the engagement is 1 day per week.
- If the user is 65 or older, the engagement is 3.5 days per week.

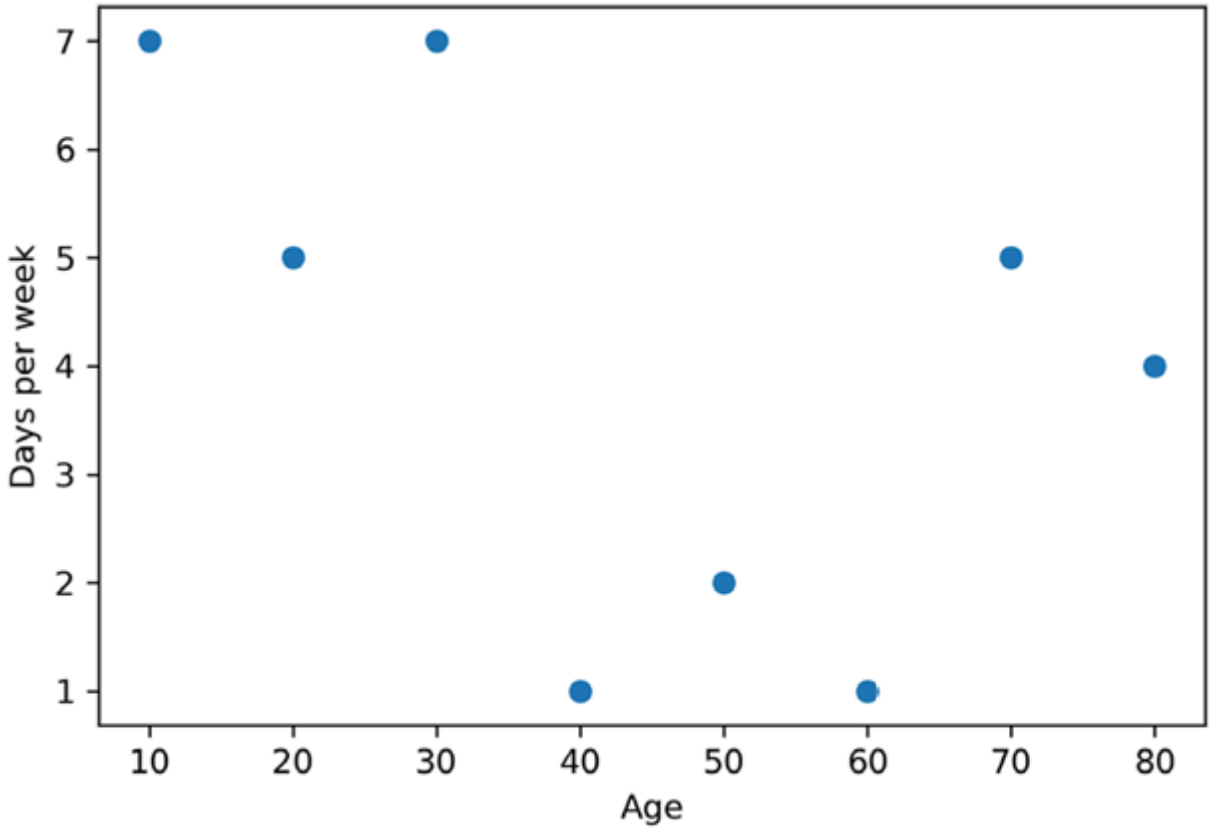


Figure 9.25 The plot of the dataset in table 9.6, where the horizontal axis corresponds to the age of the user and the vertical axis to the number of days per week that they engaged with the app

The predictions of a regression decision tree look similar to this, because the decision tree splits our users into groups and predicts a fixed value for each of the groups. The way to split the users is by using the features, exactly like we did for classification problems.

Lucky for us, the algorithm used for training a regression decision tree is very similar to the one we used for training a classification decision tree. The only difference is that for classification trees, we used accuracy, Gini index, or entropy, and for regression trees, we use the mean square error (MSE). The mean square error may sound familiar—we used it to train linear regression models in the section “How do we measure our results? The error function” in chapter 3.

Before we get into the algorithm, let’s think about it conceptually. Imagine that you have to fit a line as close as possible to the dataset in figure 9.25. But there is a catch—the line must be horizontal. Where should we fit this horizontal line? It makes sense to fit it in the “middle” of the dataset—in other words, at a height equal to the average of the labels, which is 4. That is a very simple classification model, which assigns to every point the same prediction of 4.

Now, let’s go a bit further. If we had to use two horizontal segments, how should we fit them as close as possible to the data? We might have several guesses, with one being to put a high bar for the points to the left of 35 and a low bar to the right of 35. That represents a decision stump that asks the question, “Are you younger than 35?” and assigns predictions based on how the user answered that question.

What if we could split each of these two horizontal segments into two more—where should we locate them? We can continue following this process until we have broken down the users

into several groups in which their labels are very similar. We then predict the average label for all the users in that group.

The process we just followed is the process of training a regression decision tree. Now let’s get more formal. Recall that when a feature is numerical, we consider all the possible ways to split it. Thus, the possible ways to split the age feature are using, for example, the following cutoffs: 15, 25, 35, 45, 55, 65, and 75. Each of these cutoffs gives us two smaller datasets, which we call the left dataset and the right dataset. Now we carry out the following steps:

1. For each of the smaller datasets, we predict the average value of the labels.
2. We calculate the mean square error of the prediction.
3. We select the cutoff that gives us the smallest square error.

For example, if our cutoff is 65, then the two datasets are the following:

- **Left dataset**: users younger than 65. The labels are {7, 5, 7, 1, 2, 1}.
- **Right dataset**: users 65 or older. The labels are {5,4}.

For each dataset, we predict the average of the labels, which is 3.833 for the left one and 4.5 for the right one. Thus, the prediction for the first six users is 3.833, and for the last two is 4.5. Now, we calculate the MSE as follows:

$$MSE = \frac{1}{8}[(7-3.833)^2 + (5-3.833)^2 + (7-3.833)^2 + (1-3.833)^2 + (2-3.833)^2 + (1-3.833)^2 + (5-4.5)^2 + (4-4.5)^2] = 5.167$$

In table 9.7, we can see the values obtained for each of the possible cutoffs. The full calculations are at the end of the notebook for this section.

Table 9.7 The nine possible ways to split the dataset by age using a cutoff. Each cutoff splits the dataset into two smaller datasets, and for each of these two, the prediction is given by the average of the labels. The mean square error (MSE) is calculated as the average of the squares of the differences between the labels and the prediction. Notice that the splitting with the smallest MSE is obtained with a cutoff of 35. This gives us the root node in our decision tree.

Cutoff	Labels left set	Labels right set	Prediction left set	Prediction right set	MSE
0	{}	{7,5,7,1,2,1,5,4}	None	4.0	5.25
15	{7}	{5,7,1,2,1,5,4}	7.0	3.571	3.964
25	{7,5}	{7,1,2,1,5,4}	6.0	3.333	3.917
35	{7,5,7}	{1,2,1,5,4}	6.333	2.6	1.983
45	{7,5,7,1}	{2,1,5,4}	5.0	3.0	4.25
55	{7,5,7,1,2}	{1,5,4}	4.4	3.333	4.983
65	{7,5,7,1,2,1}	{5,4}	3.833	4.5	5.167
75	{7,5,7,1,2,1,5}	{4}	4.0	4.0	5.25
100	{7,5,7,1,2,1,5,4}	{}	4.0	none	5.25

The best cutoff is at 35 years old, because it gave us the prediction with the least mean square error. Thus, we've built the first decision node in our regression decision tree. The next steps are to continue splitting the left and right datasets recursively in the same fashion. Instead of doing it by hand, we'll use Scikit-Learn as before.

First, we define our features and labels. We can use arrays for this, as shown next:

features = [[10],[20],[30],[40],[50],[60],[70],[80]]

labels = [7,5,7,1,2,1,5,4]

Now, we build a regression decision tree of maximum depth 2 using the `DecisionTreeRegressor` object as follows:

```
from sklearn.tree import DecisionTreeRegressor
dt_regressor = DecisionTreeRegressor(max_depth=2)
dt_regressor.fit(features, labels)
```

The resulting decision tree is shown in figure 9.26. The first cutoff is at 35, as we had already figured out. The next two cutoffs are at 15 and 65. At the right of figure 9.26, we can also see the predictions for each of these four resulting subsets of the data.

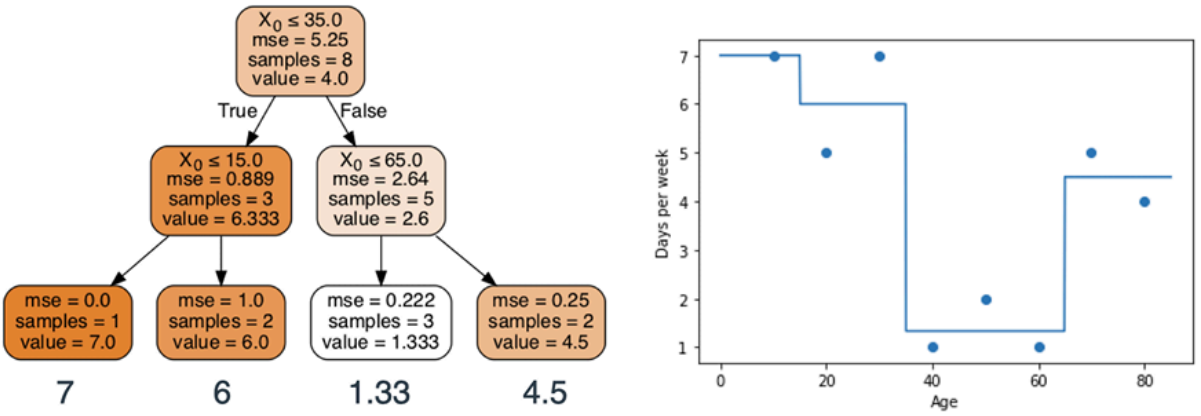


Figure 9.26 Left: The resulting decision tree obtained in Scikit-Learn. This tree has three decision nodes and four leaves. Right: The plot of the predictions made by this decision tree. Note that the cutoffs are at ages 35, 15, and 65, corresponding to the decision nodes in the tree. The predictions are 7, 6, 1.33, and 4.5, corresponding to the leaves in the tree.

Applications

Decision trees have many useful applications in real life. One special feature of decision trees is that, aside from predicting, they give us a lot of information about our data, because they organize it in a hierarchical structure. Many times, this information is of as much or even more value as the capacity of making predictions. In this section, we see some examples of decision trees used in real life in the following fields:

- Health care
- Recommendation systems

Decision trees are widely used in health care

Decision trees are widely used in medicine, not only to make predictions but also to identify features that are determinant in the prediction. You can imagine that in medicine, a black box saying “the patient is sick” or “the patient is healthy” is not good enough. However, a decision tree comes with a great deal of information about why the prediction was made.

The patient could be sick based on their symptoms, family medical history, habits, or many other factors.

Decision trees are useful in recommendation systems

In recommendation systems, decision trees are also useful. One of the most famous recommendation systems problems, the Netflix prize, was won with the help of decision trees. In 2006, Netflix held a competition that involved building the best possible recommendation system to predict user ratings of their movies. In 2009, they awarded $1,000,000 USD to the winner, who improved the Netflix algorithm by over 10%. The way they did this was using gradient-boosted decision trees to combine more than 500 different models. Other recommendation engines use decision trees to study the engagement of their users and figure out the demographic features to best determine engagement.

In chapter 12, we will learn more about gradient-boosted decision trees and random forests. For now, the best way to imagine them is as a collection of many decision trees working together to make the best predictions.

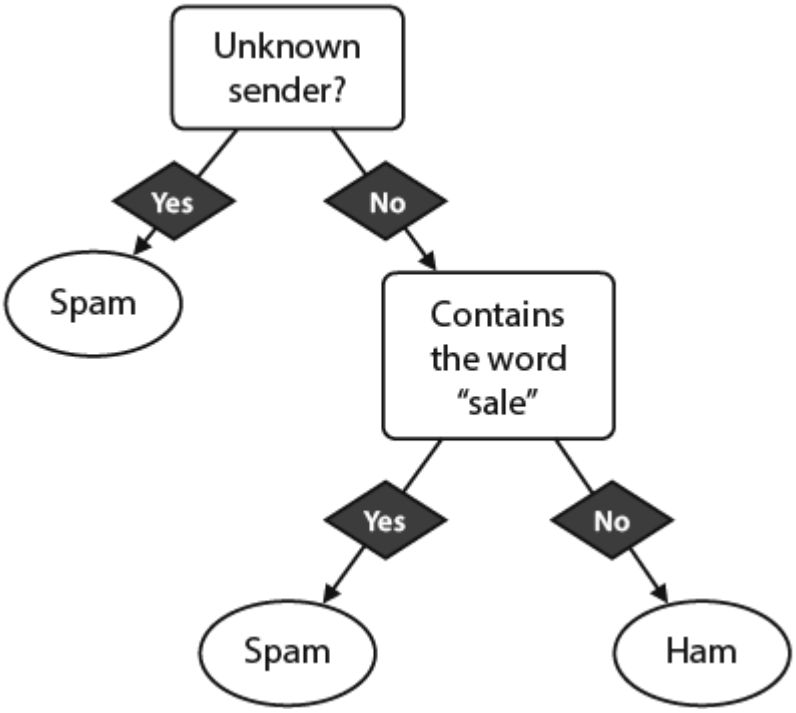
Summary

- Decision trees are important machine learning models, used for classification and regression.
- The way decision trees work is by asking binary questions about our data and making a prediction based on the answers to those questions.
- The algorithm for building decision trees for classification consists of finding the feature in our data that best determines the label and iterating over this step.
- We have several ways to tell if a feature determines the label best. The three that we learned in this chapter are accuracy, Gini impurity index, and entropy.
- The Gini impurity index measures the purity of a set. In that way, a set in which every element has the same label has a Gini impurity index of 0. A set in which every element has a different label has a Gini impurity label close to 1.
- Entropy is another measure for the purity of a set. A set in which every element has the same label has an entropy of 0. A set in which half of the elements have one label and the other half has another label has an entropy of 1. When building a decision tree, the difference in entropy before and after a split is called information gain.
- The algorithm for building a decision tree for regression is similar to the one used for classification. The only difference is that we use the mean square error to select the best feature to split the data.
- In two dimensions, regression tree plots look like the union of several horizontal lines, where each horizontal line is the prediction for the elements in a particular leaf.
- Applications of decision trees range very widely, from recommendation algorithms to applications in medicine and biology.

Exercises

Exercise 9.1

In the following spam-detection decision tree model, determine whether an email from your mom with the subject line, “Please go to the store, there’s a sale,” will be classified as spam.



Exercise 9.2

Our goal is to build a decision tree model to determine whether credit card transactions are fraudulent. We use the dataset of credit card transactions below, with the following features:

- **Value**: value of the transaction.
- **Approved vendor**: the credit card company has a list of approved vendors. This variable indicates whether the vendor is in this list.

	Value	Approved vendor	Fraudulent
Transaction 1	$100	Not approved	Yes
Transaction 2	$100	Approved	No
Transaction 3	$10,000	Approved	No
Transaction 4	$10,000	Not approved	Yes
Transaction 5	$5,000	Approved	Yes

Transaction 6	$100	Approved	No

Build the first node of the decision tree with the following specifications:

1. Using the Gini impurity index
2. Using entropy

Exercise 9.3

A dataset of patients who have tested positive or negative for COVID-19 follows. Their symptoms are cough (C), fever (F), difficulty breathing (B), and tiredness (T).

	Cough (C)	Fever (F)	Difficulty breathing (B)	Tiredness (T)	Diagnosis
Patient 1		X	X	X	Sick
Patient 2	X	X		X	Sick
Patient 3	X		X	X	Sick
Patient 4	X	X	X		Sick
Patient 5	X			X	Healthy
Patient 6		X	X		Healthy
Patient 7		X			Healthy
Patient 8				X	Healthy

Using accuracy, build a decision tree of height 1 (a decision stump) that classifies this data. What is the accuracy of this classifier on the dataset?

support vector machine (SVM for short). An SVM is similar to a perceptron, in that it separates a dataset with two classes using a linear boundary. However, the SVM aims to find the linear boundary that is located as far as possible from the points in the dataset. We also cover the kernel method, which is useful when used in conjunction with an SVM, and it can help classify datasets using highly nonlinear boundaries.

In chapter 5, we learned about linear classifiers, or perceptrons. With two-dimensional data, these are defined by a line that separates a dataset consisting of points with two labels. However, we may have noticed that many different lines can separate a dataset, and this raises the following question: how do we know which is the best line? In figure 11.1, we can see three different linear classifiers that separate this dataset. Which one do you prefer, classifier 1, 2, or 3?

Figure 11.1 Three classifiers that classify our data set correctly. Which should we prefer, classifier 1, 2, or 3?

If you said classifier 2, we agree. All three lines separate the dataset well, but the second line is better placed. The first and third lines are very close to some of the points, whereas the second line is far from all the points. If we were to wiggle the three lines around a little bit, the first and the third may go over some of the points, misclassifying some of them in the process, whereas the second one will still classify them all correctly. Thus, classifier 2 is more robust than classifiers 1 and 3.

This is where support vector machines come into play. An SVM classifier uses two parallel lines instead of one line. The goal of the SVM is twofold; it tries to classify the data correctly and also tries to space the lines as much as possible. In figure 11.2, we can see the two parallel lines for the three classifiers, together with their middle line for reference. The two external (dotted) lines in classifier 2 are the farthest from each other, which makes this classifier the best one.

Figure 11.2 We draw our classifier as two parallel lines, as far apart from each other as possible. We can see that classifier 2 is the one where the parallel lines are the farthest away from each other. This means that the middle line in classifier 2 is the one best located between the points.

We may want to visualize an SVM as the line in the middle that tries to stay as far as possible from the points. We can also imagine it as the two external parallel lines trying to stay as far away from each other as possible. In this chapter, we'll use both visualizations at different times, because each of them is useful in certain situations.

How do we build such a classifier? We can do this in a similar way as before, with a slightly different error function and a slightly different iterative step.

note In this chapter, all the classifiers are discrete, namely, their output is 0 or 1. Sometimes they are described by their prediction $\hat{y} = step(f(x))$, and other times by their boundary equation $f(x) = 0$, namely, the graph of the function that attempts to separate our data points into two classes. For example, the perceptron that makes the prediction $\hat{y} = step(3x_1 + 4x_2 - 1)$ sometimes is described only by the linear equation $3x_1 + 4x_2 - 1 = 0$. For some classifiers in this chapter, especially those in the section "Training SVMs with nonlinear boundaries: The kernel method," the boundary equation will not necessarily be a linear function.

In this chapter, we see this theory mostly on datasets of one and two dimensions (points on a line or on the plane). However, support vector machines work equally well in datasets of higher dimensions. The linear boundaries in one dimension are points and in two dimensions are lines. Likewise, the linear boundaries in three dimensions are planes, and in higher dimensions, they are hyperplanes of one dimension less than the space in which the points live. In each of these cases, we try to find the boundary that is the farthest from the points. In figure 11.3, you can see examples of boundaries for one, two, and three dimensions.

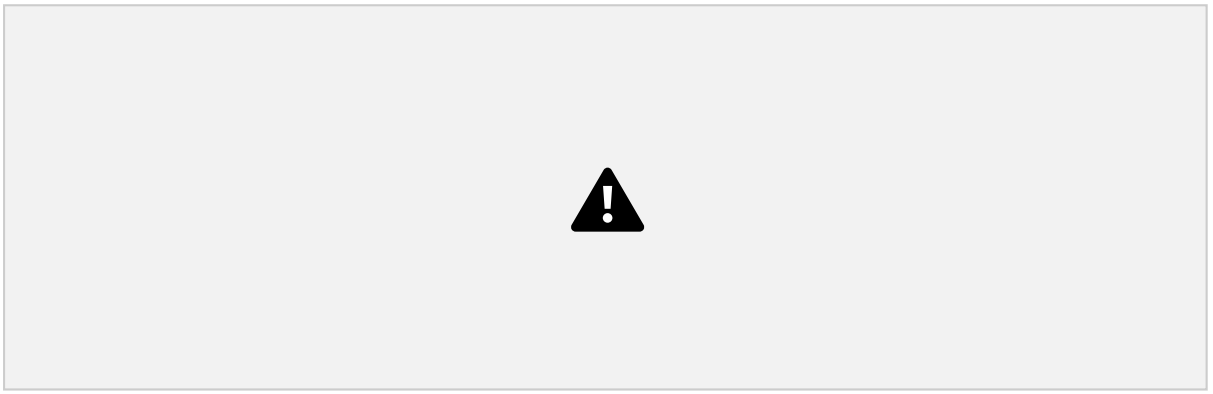


Figure 11.3 Linear boundaries for datasets in one, two, and three dimensions. In one dimension, the boundary is formed by two points, in two dimensions by two lines, and in three dimensions by two planes. In each of the cases, we try to separate these two as much as possible. The middle boundary (point, line, or plane) is illustrated for clarity.

All the code for this chapter is in this GitHub repository: https://github.com/luisguiserrano/manning/tree/master/Chapter_11_Support_Vector_Machines.

Using a new error function to build better classifiers

As is common in machine learning models, SVMs are defined using an error function. In this section, we see the error function of SVMs, which is very special, because it tries to maximize two things at the same time: the classification of the points and the distance between the lines.

To train an SVM, we need to build an error function for a classifier consisting of two lines, spaced as far apart as possible. When we think of building an error function, we should always ask ourselves: "What do we want the model to achieve?" The following are the two things we want to achieve:

- Each of the two lines should classify the points as best as possible.
- The two lines should be as far away from each other as possible.

The error function should penalize any model that doesn't achieve these things. Because we want two things, our SVM error function should be the sum of two error functions: the first one penalizes points that are misclassified, and the second one penalizes lines that are too close to each other. Therefore, our error function can look like this:

Error = Classification Error + Distance Error

In the next two sections, we develop each one of these two terms separately.

Classification error function: Trying to classify the points correctly

In this section, we learn the classification error function. This is the part of the error function that pushes the classifier to correctly classify the points. In short, this error is calculated as follows. Because the classifier is formed by two lines, we think of them as two separate discrete perceptrons (chapter 5). We then calculate the total error of this classifier as the sum of the two perceptron errors (section "How to compare classifiers? The error function" in chapter 5). Let's take a look at an example.

The SVM uses two parallel lines, and luckily, parallel lines have similar equations; they have the same weights but a different bias. Thus, in our SVM, we use the central line as a frame of reference L with equation $w_1x_1 + w_2x_2 + b = 0$, and construct two lines, one above it and one below it, with the respective equations:

- L+: $w_1x_1 + w_2x_2 + b = 1$, and
- L–: $w_1x_1 + w_2x_2 + b = -1$

As an example, figure 11.4 shows the three parallel lines, L, L+, and L–, with the following equations:

- L: $2x_1 + 3x_2 - 6 = 0$
- L+: $2x_1 + 3x_2 - 6 = 1$
- L–: $2x_1 + 3x_2 - 6 = -1$

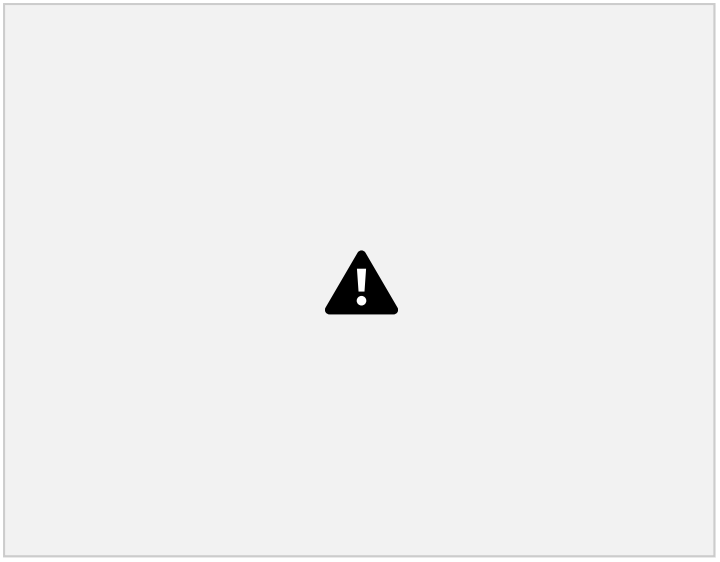


Figure 11.4 Our main line L is the one in the middle. We build the two parallel equidistant lines L+ and L– by slightly changing the equation of L.

Our classifier now consists of the lines L+ and L–. We can think of L+ and L– as two independent perceptron classifiers, and each of them has the same goal of classifying the points correctly. Each classifier comes with its own perceptron error function, so the classification function is defined as the sum of these two error functions, as illustrated in figure 11.5.

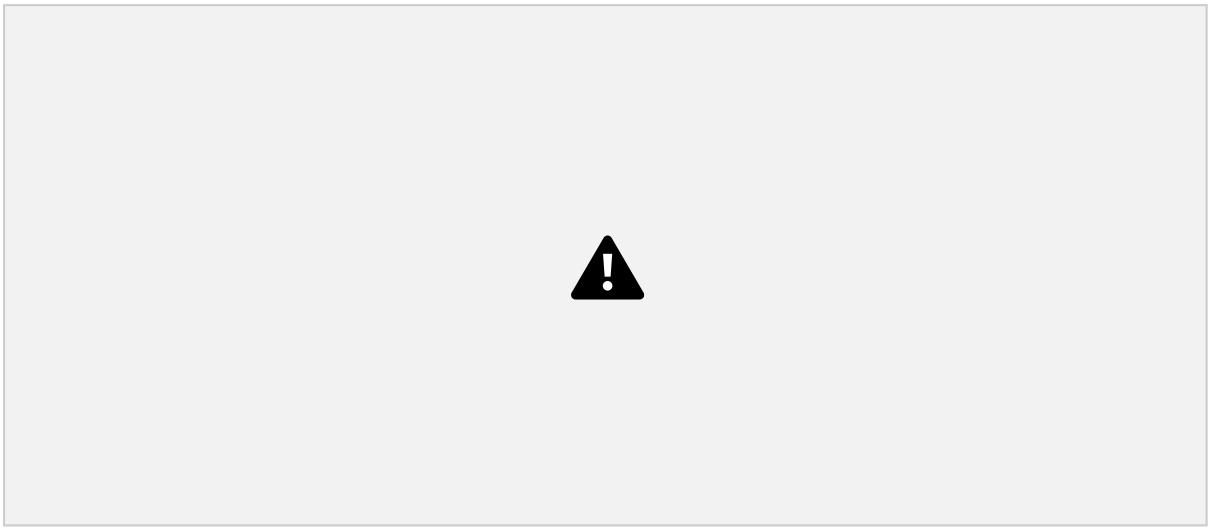


Figure 11.5 Now that our classifier consists of two lines, the error of a misclassified point is measured with respect to both lines. We then add the two errors to obtain the classification error. Note that the error is not the length of the perpendicular segment to the boundary, as illustrated, but it is proportional to it.

Notice that in an SVM, *both* lines have to classify the points well. Therefore, a point that is between the two lines is always misclassified by one of the lines, so it does not count as a correctly classified point by the SVM.

Recall from the section “How to compare classifiers? The error function” in chapter 5 that the error function for the discrete perceptron with prediction $\hat{y} = step(w_1x_1 + w_2x_2 + b)$ at the point (p, q) is given by the following:

- 0 if the point is correctly classified, and
- $|w_1x_1 + w_2x_2 + b|$ if the point is incorrectly classified

As an example, consider the point (4,3) with a label of 0. This point is incorrectly classified by both of the perceptrons in figure 11.5. Note that the two perceptrons give the following predictions:

- L+: $\hat{y} = step(2x_1 + 3x_2 – 7)$
- L–: $\hat{y} = step(2x_1 + 3x_2 – 5)$

Therefore, its classification error with respect to this SVM is

$|2 \cdot 4 + 3 \cdot 3 – 7| + |2 \cdot 4 + 3 \cdot 3 – 5| = 10 + 12 = 22.$

Distance error function: Trying to separate our two lines as far apart as possible

Now that we have created an error function that measures classification errors, we need to build one that looks at the distance between the two lines and raises an alarm if this distance is small. In this section, we discuss a surprisingly simple error function that is large when the two lines are close and small when they are far.

This error function is called the distance error function, and we’ve already seen it before; it is the regularization term we learned in the section “Modifying the error function to solve our problem” in chapter 4. More specifically, if our lines have equations $w_1x_1 + w_2x_2 + b = 1$ and $w_1x_1 + w_2x_2 + b = –1$, then the error function is $w_1^2 + w_2^2$. Why? We’ll make use of the following fact: the perpendicular distance between the two lines is precisely  , as illustrated in figure 11.6. If you’d like to work out the details of this distance calculation, please check exercise 11.1 at the end of this chapter.

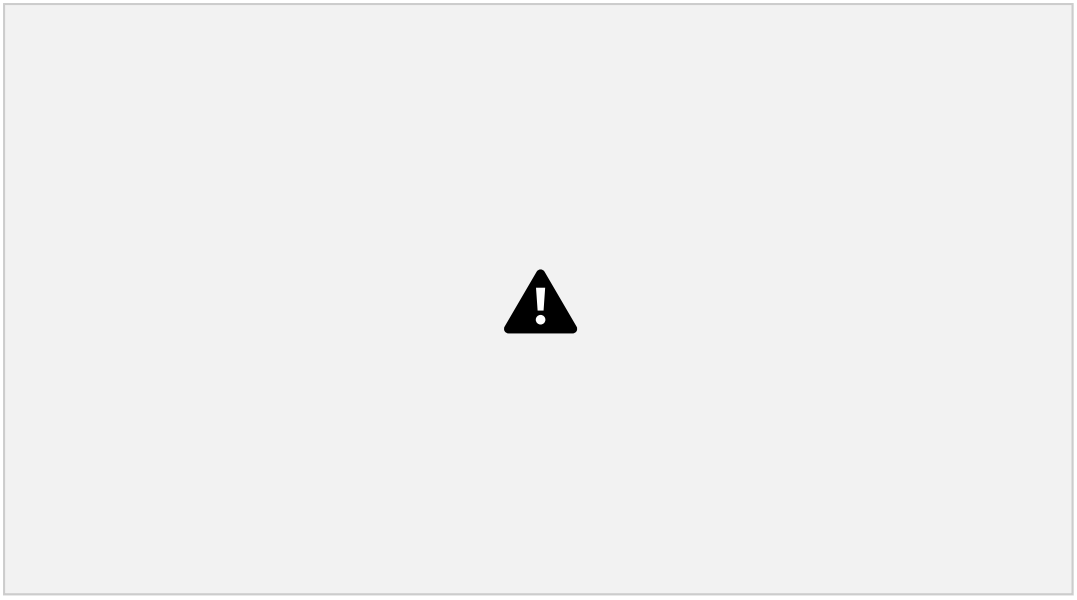


Figure 11.6 The distance between the two parallel lines can be calculated based on the equations of the lines.

Knowing this, notice the following:

- When $w_1^2 + w_2^2$ is large, is small.
- When $w_1^2 + w_2^2$ is small, is large.

Because we want the lines to be as far apart as possible, this term $w_1^2 + w_2^2$ is a good error function, as it gives us large values for the bad classifiers (those where the lines are close) and small values for the good classifiers (those where the lines are far).

In figure 11.7, we can see two examples of SVM classifiers. Their equations follow:

- SVM 1:
  - L+: $3x_1 + 4x_2 + 5 = 1$
  - L–: $3x_1 + 4x_2 + 5 = –1$
- SVM 2:
  - L+: $30x_1 + 40x_2 + 50 = 1$
  - L–: $30x_1 + 40x_2 + 50 = 1$

Their distance error functions are shown next:

- SVM 1:
  - Distance error function = $3^2 + 4^2 = 25$
- SVM 2:
  - Distance error function = $30^2 + 40^2 = = 2500$

Notice also from figure 11.7 that the lines are much closer in SVM 2 than in SVM 1, which makes SVM 1 a much better classifier (from the distance perspective). The distance between the lines in SVM 1 is , whereas in SVM 2 it is .

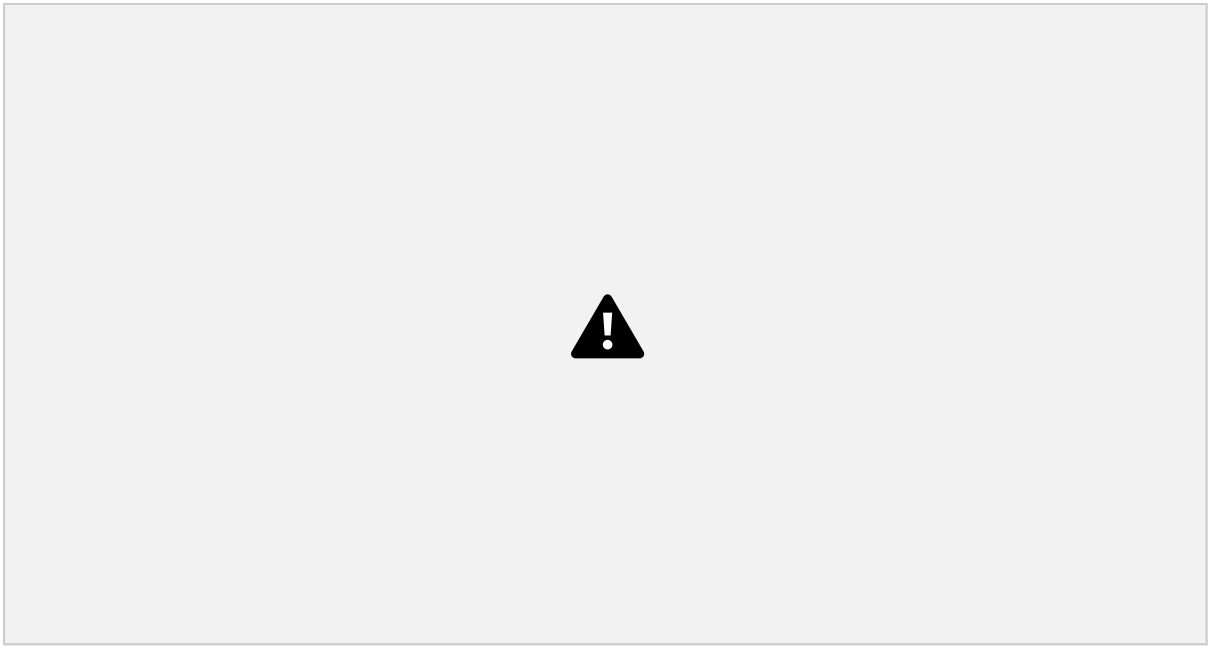


Figure 11.7 Left: An SVM where the lines are at distance 0.4 apart, with an error of 25. Right: An SVM where the lines are at distance 0.04 apart, with an error of 2500. Notice that in this comparison, the classifier on the left is much better than the one on the right, because the lines are farther apart from each other. This results in a smaller distance error.

Adding the two error functions to obtain the error function

Now that we've built a classification error function and a distance error function, let's see how to combine them to build an error function that helps us make sure that we have achieved both goals: classify our points well and with two lines that are far apart from each other.

To obtain this error function, we add the classification error function and the distance error function and get the following formula:

Error = Classification Error + Distance Error

A good SVM that minimizes this error function must then try to make as few classification errors as possible, while simultaneously trying to keep the lines as far apart as possible.

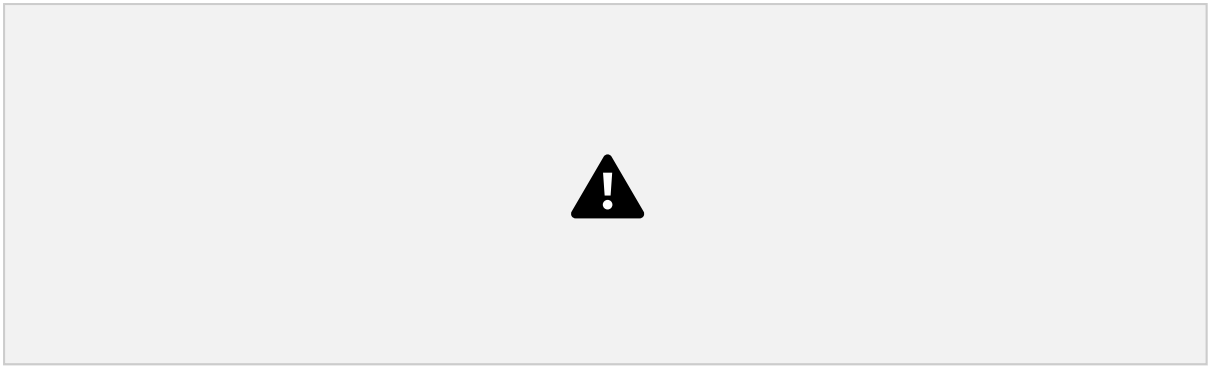


Figure 11.8 Left: A good SVM consisting of two well-spaced lines that classifies all the points correctly. Middle: A bad SVM that misclassifies two points. Right: A bad SVM that consists of two lines that are too close together.

In figure 11.8, we can see three SVM classifiers for the same dataset. The one on the left is a good classifier, because it classifies the data well and the lines are far apart, reducing the likelihood of errors. The one in the middle makes some errors (because there is a triangle underneath the top line and a square over the bottom line), so it is not a good classifier. The one on the right classifies the points correctly, but the lines are too close together, so it is also not a good classifier.

Do we want our SVM to focus more on classification or distance? The C parameter can help us

In this section, we learn a useful technique to tune and improve our model, which involves introducing the C parameter. The C parameter is used in cases where we want to train an SVM that pays more attention to classification than to distance (or the other way around).

So far it seems that all we have to do to build a good SVM classifier is to keep track of two things. We want to make sure the classifier makes as few errors as possible, while keeping the lines as far apart as possible. But what if we have to sacrifice one for the benefit of the other? In figure 11.9, we have two classifiers for the same dataset. The one on the left makes some errors, but the lines are far apart. The one on the right makes no errors, but the lines are too close together. Which one should we prefer?

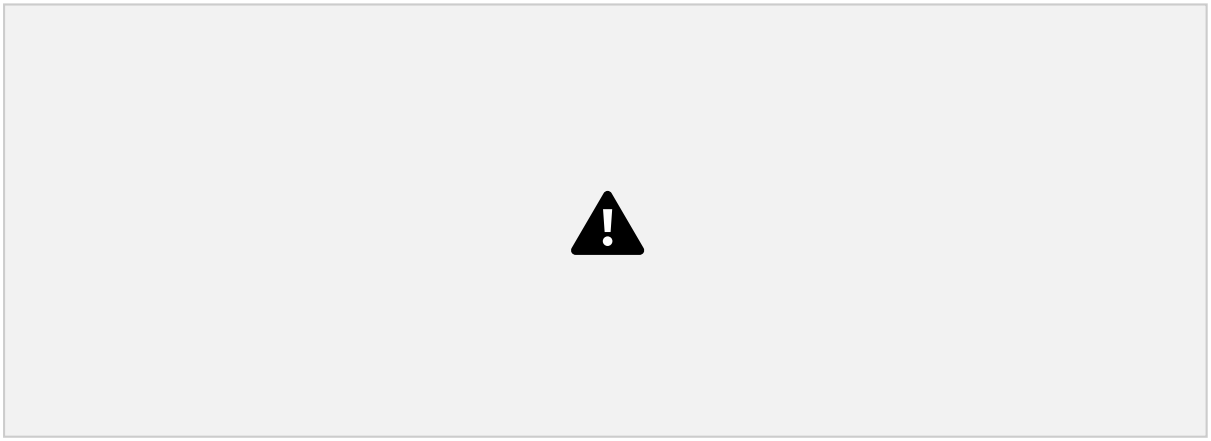


Figure 11.9 Both of these classifiers have one pro and one con. The one on the left consists of well-spaced lines (pro), but it misclassifies some points (con). The one on the right consists of lines that are too close together (con), but it classifies all the points correctly (pro).

It turns out that the answer for this depends on the problem we are solving. Sometimes we want a classifier that makes as few errors as possible, even if the lines are too close, and sometimes we want a classifier that keeps the lines apart, even if it makes a few errors. How do we control this? We use a parameter which we call the C parameter. We slightly modify the error formula by multiplying the classification error by C, to get the following formula:

Error formula = C · (Classification Error) + (Distance Error)

If C is large, then the error formula is dominated by the classification error, so our classifier focuses more on classifying the points correctly. If C is small, then the formula is dominated by the distance error, so our classifier focuses more on keeping the lines far apart.

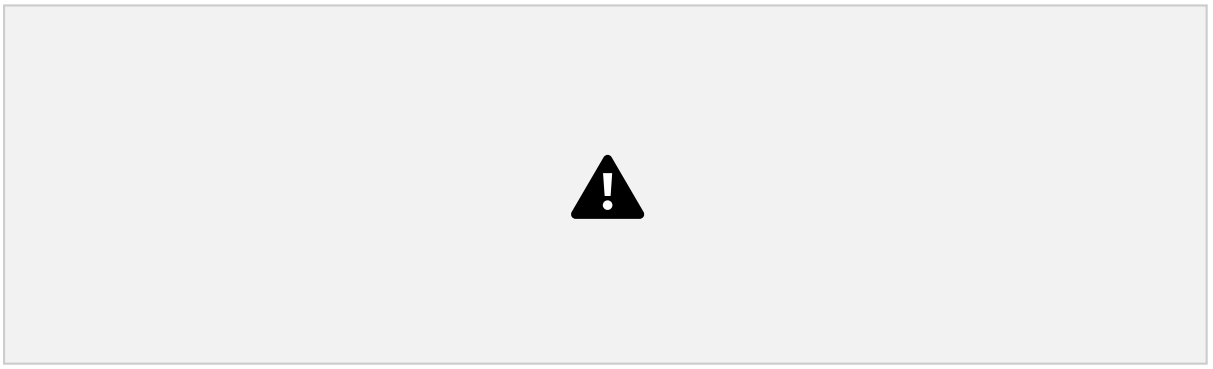


Figure 11.10 Different values of C toggle between a classifier with well-spaced lines and one that classifies points correctly. The classifier on the left has a small value of C (0.01), and the lines are well spaced, but it makes mistakes. The classifier on the right has a large value of C (100), and it classifies points correctly, but the lines are too close together. The classifier in the middle makes one mistake but finds two lines that are well spaced apart.

In figure 11.10, we can see three classifiers: one with a large value of C that classifies all points correctly, one with a small value of C that keeps the lines far apart, and one with C = 1, which tries to do both. In real life, C is a hyperparameter that we can tune using methods such as the model complexity graph (section “A numerical way to decide how complex our model should be” in chapter 4) or our own knowledge of the problem we’re solving, the data, and the model.

Coding support vector machines in Scikit-Learn

Now that we’ve learned what an SVM is, we are ready to code one and use it to model some data. In Scikit-Learn, coding an SVM is simple and that’s what we learn in this section. We also learn how to use the C parameter in our code.

Coding a simple SVM

We start by coding a simple SVM in a sample dataset and then we'll add more parameters. The dataset is called linear.csv, and its plot is shown in figure 11.11. The code for this section follows:

- **Notebook**: SVM_graphical_example.ipynb
    - https://github.com/luisguiserrano/manning/blob/master/Chapter_11_Support_Vector_Machines/SVM_graphical_example.ipynb
- **Dataset**: linear.csv

We first import from the `svm` package in Scikit-Learn and load our data as follows:

```
from sklearn.svm import SVC
```

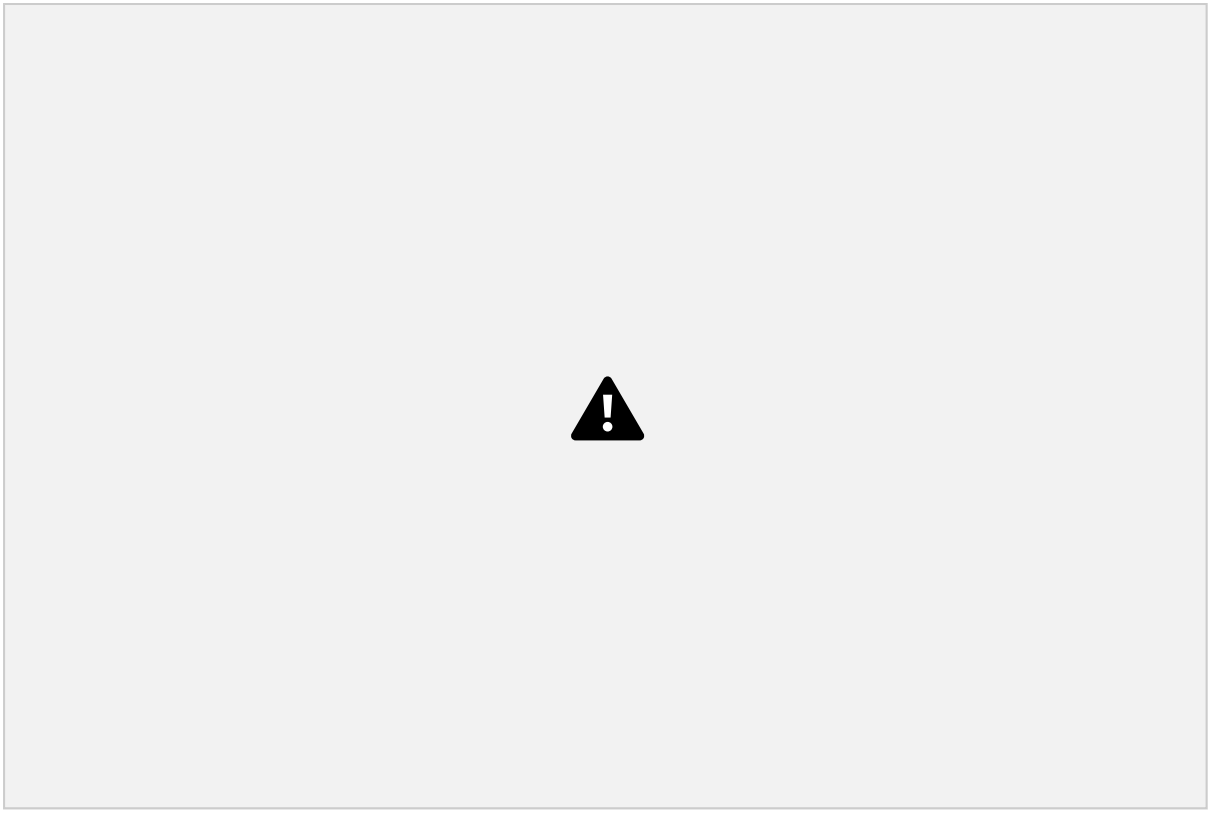


Figure 11.11 An almost linearly separable dataset, with some outliers

Then, as shown in the next code snippet, we load our data into two Pandas DataFrames called `features` and `labels`, and then we define our model called `svm_linear` and train it. The accuracy we obtain is 0.933, and the plot is shown in figure 11.12.

```
svm_linear = SVC(kernel='linear')

svm_linear.fit(features, labels)
```

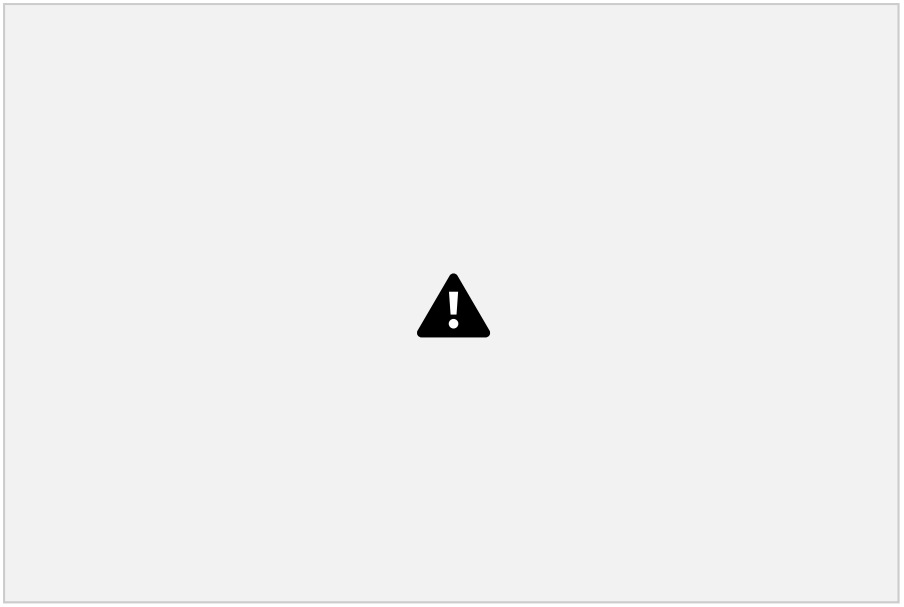


Figure 11.12 The plot of the SVM classifier we've built in Scikit-Learn consists of a line. The accuracy of this model is 0.933.

The C parameter

In Scikit-Learn, we can easily introduce the C parameter into the model. Here we train and plot two models, one with a very small value of 0.01, and another one with a large value of 100, which is shown in the following code and in figure 11.13:

```
svm_c_001 = SVC(kernel='linear', C=0.01)

svm_c_001.fit(features, labels)


svm_c_100 = SVC(kernel='linear', C=100)

svm_c_100.fit(features, labels)
```

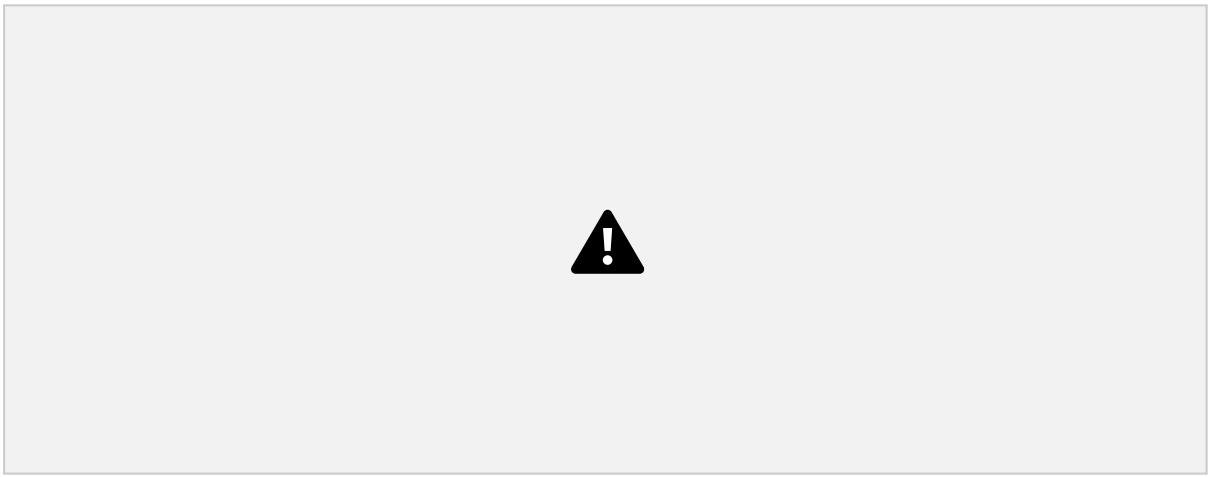


Figure 11.13 The classifier on the left has a small value of C, and it spaced the line well between the points, but it makes some mistakes. The classifier on the right has a large value of C, and it makes no mistakes, although the line passes too close to some of the points.

We can see that the model with a small value of C doesn't put that much emphasis on classifying the points correctly, and it makes some mistakes, as is evident in its low accuracy (0.867). It is hard to tell in this example, but this classifier puts a lot of emphasis on the line being as far away from the points as possible. In contrast, the classifier with the large value of C tries to classify all the points correctly, which reflects on its higher accuracy.

Training SVMs with nonlinear boundaries: The kernel method

As we've seen in other chapters of this book, not every dataset is linearly separable, and many times we need to build nonlinear classifiers to capture the complexity of the data. In this section, we study a powerful method associated with SVMs called the *kernel method*, which helps us build highly nonlinear classifiers.

If we have a dataset and find that we can't separate it with a linear classifier, what can we do? One idea is to add more columns to this dataset and hope that the richer dataset is linearly separable. The kernel method consists of adding more columns in a clever way, building a linear classifier on this new dataset and later removing the columns we added while keeping track of the (now nonlinear) classifier.

That was quite a mouthful, but we have a nice geometric way to see this method. Imagine that the dataset is in two dimensions, which means that the input has two columns. If we add a third column, the dataset is now three-dimensional, like if the points on your paper all of a sudden start flying into space at different heights. Maybe if we raise the points at different heights in a clever way, we can separate them with a plane. This is the kernel method, and it is illustrated in figure 11.14.

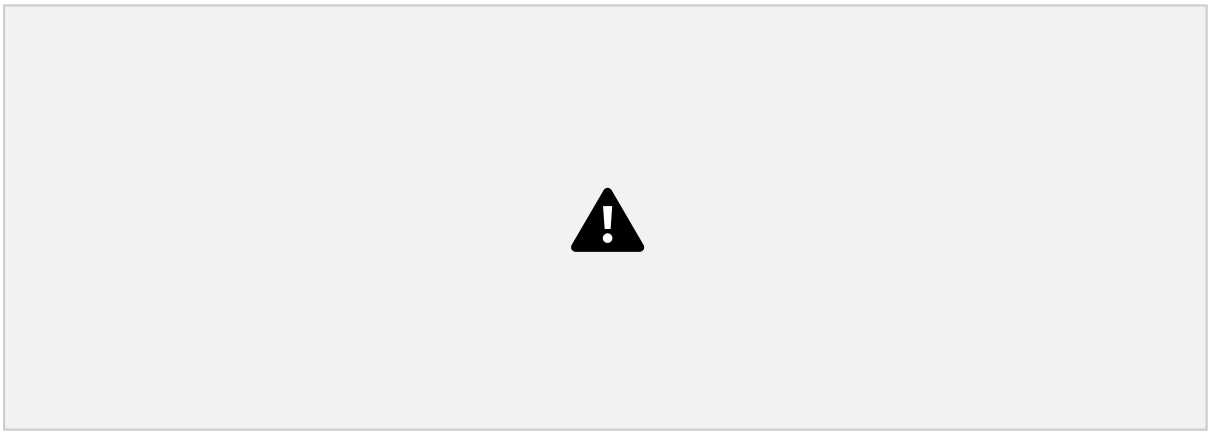


Figure 11.14 Left: The set is not separable by a line. Middle: We look at it in three dimensions, and proceed to raise the two triangles and lower the two squares. Right: Our new dataset is now separable by a plane. (Source: Image created with the assistance of Grapher™ from Golden Software, LLC; https://www.goldensoftware.com/products/grapher).

Kernels, feature maps, and operator theory

The theory behind the kernel method comes from a field in mathematics called *operator theory*. A kernel is a similarity function, which, in short, is a function that tells us if two points are similar or different (e.g., close or far). A kernel can give rise to a *feature map*, which is a map between the space where our dataset lives and a (usually) higher-dimensional space.

The full theory of kernels and feature maps is not needed to understand the classifiers. If you'd like to delve into these more, see the resources in appendix C. For the purpose of this chapter, we look at the kernel method as a way of adding columns to our dataset to make the points separable. For example, the dataset in figure 11.14 has two columns, x_1 and x_2, and we have added the third column with the value x_1x_2. Equivalently, it can also be seen as the function that sends the point (x_1, x_2) in the plane to the point (x_1, x_2, x_1x_2) in space. Once the points belong in 3-D space, we can separate them using the plane seen on the right of figure 11.14. To study this example more in detail, see exercise 11.2 at the end of the chapter.

The two kernels and their corresponding features maps we see in this chapter are the *polynomial kernel* and the *radial basis function* (RBF) *kernel*. Both of them consist of adding columns to our dataset in different, yet very effective, ways.

Using polynomial equations to our benefit: The polynomial kernel

In this section, we discuss the polynomial kernel, a useful kernel that will help us model nonlinear datasets. More specifically, the kernel method helps us model data using polynomial equations such as circles, parabolas, and hyperbolas. We'll illustrate the polynomial kernel with two examples.

Example 1: A circular dataset

For our first example, let's try to classify the dataset in table 11.1.

Table 11.1 A small dataset, depicted in figure 11.15

x_1	x_2	y
0.3	0.3	0
0.2	0.8	0
–0.6	0.4	0
0.6	–0.4	0
–0.4	–0.3	0
0	–0.8	0
–0.4	1.2	1

0.9	–0.7	1
–1.1	–0.8	1
0.7	0.9	1
–0.9	0.8	1
0.6	–1	1

The plot is shown in figure 11.15, where the points with label 0 are drawn as squares and those with label 1 are drawn as triangles.

When we look at the plot in figure 11.15, it is clear that a line won't be able to separate the squares from the triangles. However, a circle would (seen in figure 11.16). Now the question is, if a support vector machine can draw only linear boundaries, how can we draw this circle?

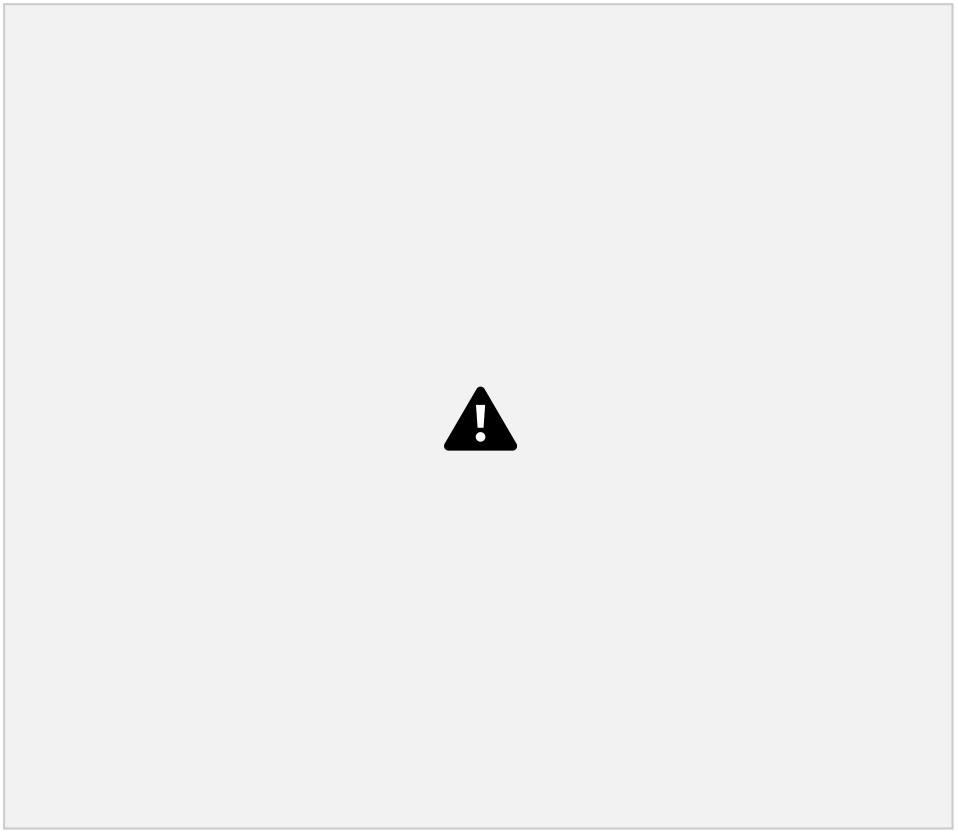


Figure 11.15 Plot of the dataset in table 11.1. Note that it is not separable by a line. Therefore, this dataset is a good candidate for the kernel method.

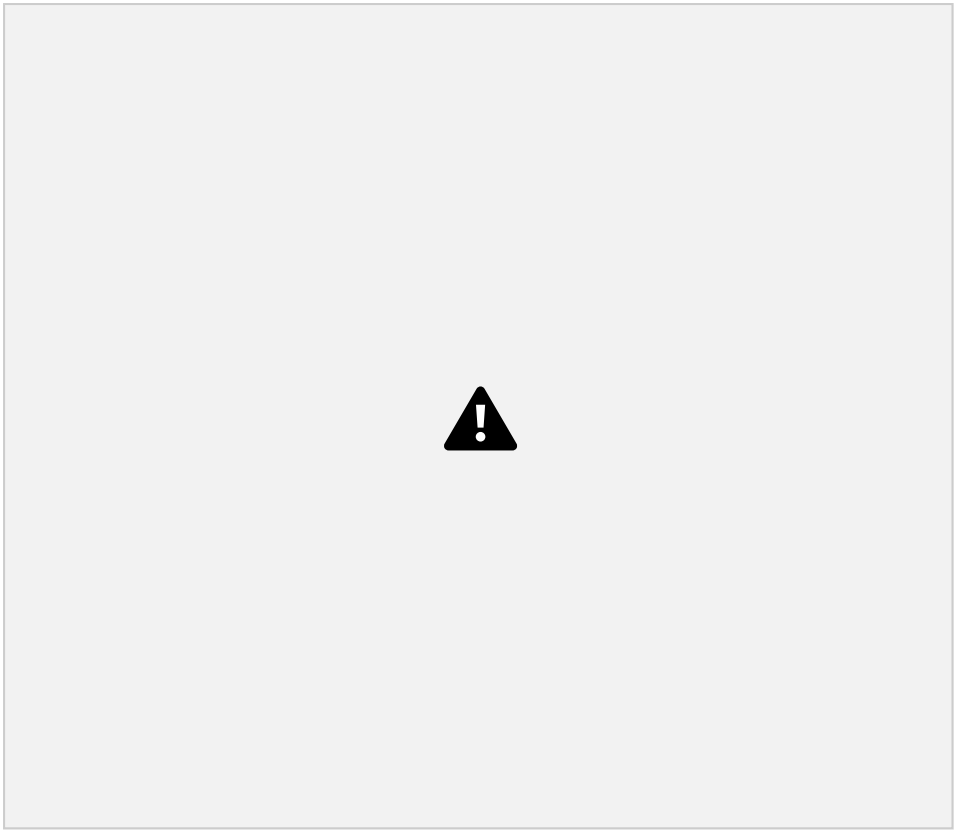


Figure 11.16 The kernel method gives us a classifier with a circular boundary, which separates these points well.

To draw this boundary, let's think. What is a characteristic that separates the squares from the triangles? From observing the plot, it seems that the triangles are farther from the origin than the circles. The formula that measures the distance to the origin is the square root of the sum of the squares of the two coordinates. If these coordinates are x_1 and x_2, then this distance is $\sqrt{x_1^2 + x_2^2}$. Let's forget about the square root, and think only of $x_1^2 + x_2^2$. Now let's add a column to table 11.1 with this value and see what happens. The resulting dataset is shown in table 11.2.

Table 11.2 We have added one more column to table 11.1. This one consists of the sum of the squares of the values of the first two columns.

x_1	x_2	$x_1^2 + x_2^2$	y
0.3	0.3	0.18	0
0.2	0.8	0.68	0
–0.6	0.4	0.52	0
0.6	–0.4	0.52	0

–0.4	–0.3	0.25	0
0	–0.8	0.64	0
–0.4	1.2	1.6	1
0.9	–0.7	1.3	1
–1.1	–0.8	1.85	1
0.7	0.9	1.3	1
–0.9	0.8	1.45	1
0.6	–1	1.36	1

After looking at table 11.2, we can see the trend. All the points labeled 0 satisfy that the sum of the squares of the coordinates is less than 1, and the points labeled 1 satisfy that this sum is greater than 1. Therefore, the equation on the coordinates that separates the points is precisely $x_1^2 + x_2^2 = 1$. Note that this is not a linear equation, because the variables are raised to a power greater than one. In fact, this is precisely the equation of a circle.

The geometric way to imagine this is depicted in figure 11.17. Our original set lives in the plane, and it is impossible to separate the two classes with a line. But if we raise each point $(x_1, x_2)$to the height $x_1^2 + x_2^2$, this is the same as putting the points in the paraboloid with equation $z = x_1^2 + x_2^2$ (drawn in the figure). The distance we raised each point is precisely the square of the distance from that point to the origin. Therefore, the squares are raised a small amount, because they are close to the origin, and the triangles are raised a large amount, because they are far away from the origin. Now the squares and triangles are far away from each other, and therefore, we can separate them with the horizontal plane at height 1—in other words, the plane with equation $z = 1$. As a final step, we project everything down to the plane. The intersection between the paraboloid and the plane becomes the circle of equation $x_1^2 + x_2^2 = 1$. Notice that this equation is not linear, because it has quadratic terms. Finally, the prediction this classifier makes is given by $\hat{y} = step(x_1^2 + x_2^2 - 1)$.

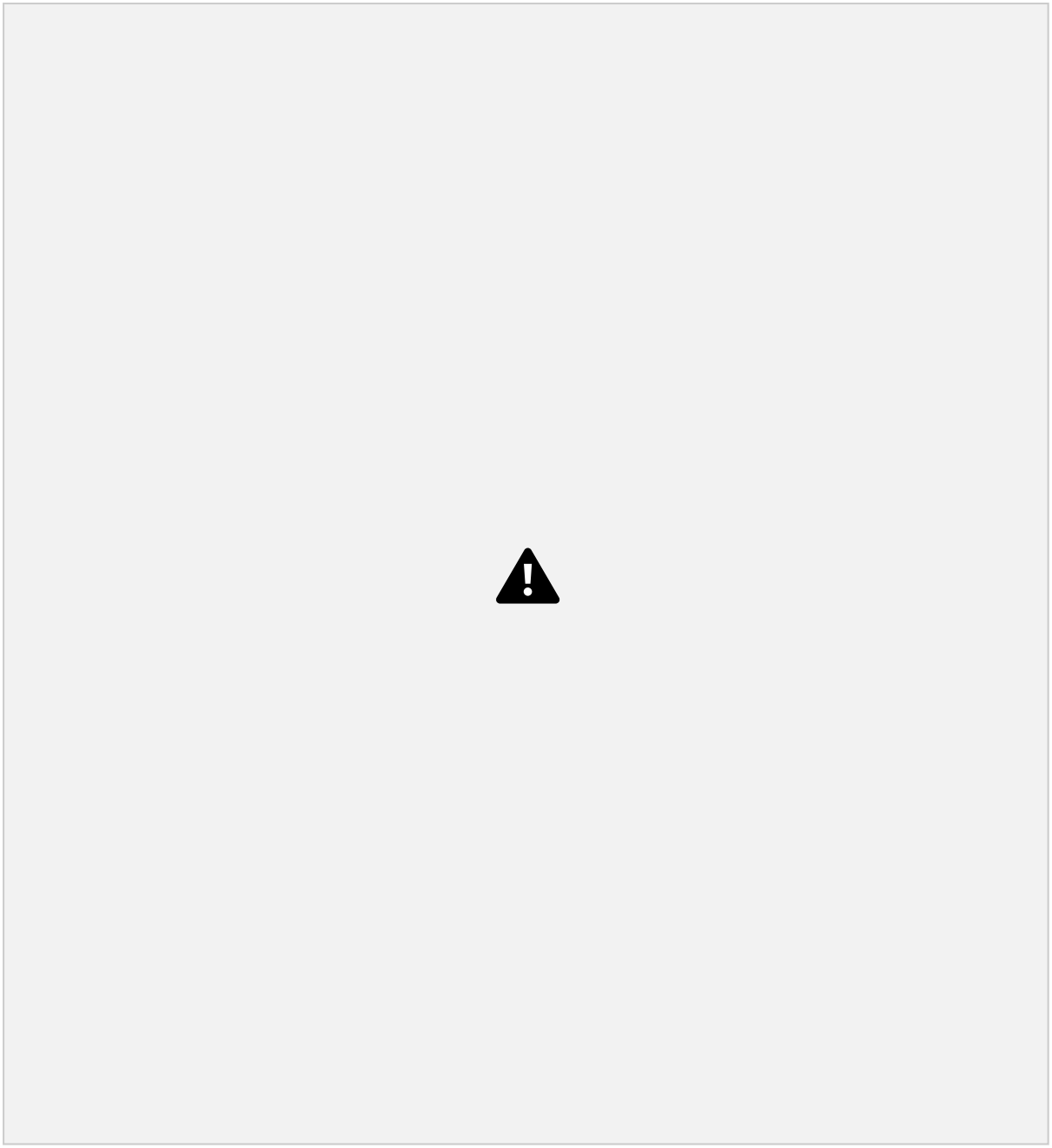


Figure 11.17 The kernel method. Step 1: We start with a dataset that is not linearly separable. Step 2: Then we raise each point by a distance that is the square of its distance to the origin. This creates a paraboloid. Step 3: Now the triangles are high, whereas the squares are low. We proceed to separate them with a plane at height 1. Step 4. We project everything down. The intersection between the paraboloid and the plane creates a circle. The projection of this circle gives us the circular boundary of our classifier. (Source: Image created with the assistance of Grapher™ from Golden Software, LLC; https://www.goldensoftware.com/products/grapher).

Example 2: The modified XOR dataset

Circles are not the only figure we can draw. Let's consider a very simple dataset, illustrated in table 11.3 and plotted in figure 11.18. This dataset is similar to the one that corresponds to

the XOR operator from exercises 5.3 and 10.2. If you'd like to solve the same problem with the original XOR dataset, you can do it in exercise 11.2 at the end of the chapter.

Table 11.3 The modified XOR dataset

x_1	x_2	y
–1	–1	1
–1	1	0
1	–1	0
1	1	1

To see that this dataset is not linearly separable, take a look at figure 11.18. The two triangles lie on opposite corners of a large square, and the two squares lie on the remaining two corners. It is impossible to draw a line that separates the triangles from the squares. However, we can use a polynomial equation to help us, and this time we'll use the product of the two features. Let's add the column corresponding to the product x_1x_2 to the original dataset. The result is shown in table 11.4.

Table 11.4 We have added a column to table 11.3, which consists of the product of the first two columns. Notice that there is a strong relation between the rightmost two columns on the table.

x_1	x_2	$x_1\ x_2$	y
–1	–1	1	1
–1	1	–1	0
1	–1	–1	0
1	1	1	1

Notice that the column corresponding to the product x_1x_2 is very similar to the column of labels. We can now see that a good classifier for this data is the one with the following boundary equation: $x_1x_2 = 1$. The plot of this equation is the union of the horizontal and vertical axes, and the reason for this is that for the product x_1x_2 to be 0, we need that $x_1 = 0$ or $x_2 = 0$. The prediction this classifier makes is given by $\hat{y} = step(x_1x_2)$, and it is 1 for points in the northeast and southwest quadrants of the plane, and 0 elsewhere.

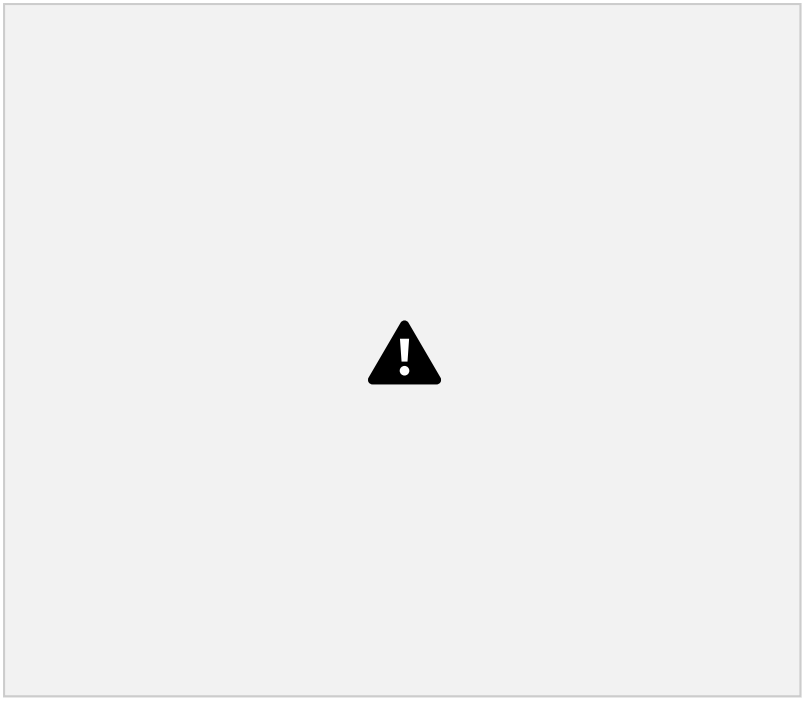


Figure 11.18 The plot of the dataset in table 11.3. The classifier that separates the squares from the triangles has boundary equation $x_1x_2 = 0$, which corresponds to the union of the horizontal and vertical axes.

Going beyond quadratic equations: The polynomial kernel

In both of the previous examples, we used a polynomial expression to help us classify a dataset that was not linearly separable. In the first example, this expression was $x_1^2 + x_2^2$, because that value is small for points near the origin and large for points far from the origin. In the second example, the expression was x_1x_2, which helped us separate points in different quadrants of the plane.

How did we find these expressions? In a more complicated dataset, we may not have the luxury to look at a plot and eyeball an expression that will help us out. We need a method or, in other words, an algorithm. What we'll do is consider all the possible monomials of degree 2 (quadratic), containing x_1 and x_2. These are the following three monomials: x_1^2, x_1x_2, and x_2^2. We call these new variables x_3, x_4, and x_5, and we treat them as if they had no relation with x_1 and x_2 whatsoever. Let's apply this to the first example (the circle). The dataset in table 11.1 with these new columns added is shown in table 11.5.

We can now build an SVM that classifies this enhanced dataset. The way to train an SVM is using the methods learned in the last section. I encourage you to build such a classifier using Scikit-Learn, Turi Create, or the package of your choice. By inspection, here is one equation of a classifier that works:

$0x_1 + 0x_2 + 1x_3 + 0x_4 + 1x_5 – 1 = 0$

Table 11.5 We have added three more rowcolumns to table 11.1, one corresponding to each of the monomials of degree 2 on the two variables x_1 and x_2. These monomials are x_1^2, $x_1\ x_2$, and x_2^2.

x_1	x_2	$x_3 = x_1^2$	$x_4 = x_1 x_2$	$x_5 = x_2^2$	y
0.3	0.3	0.09	0.09	0.09	0
0.2	0.8	0.04	0.16	0.64	0
–0.6	0.4	0.36	–0.24	0.16	0
0.6	–0.4	0.36	–0.24	0.16	0
–0.4	–0.3	0.16	0.12	0.09	0
0	–0.8	0	0	0.64	0
–0.4	1.2	0.16	–0.48	1.44	1
0.9	–0.7	0.81	–0.63	0.49	1
–1.1	–0.8	1.21	0.88	0.64	1
0.7	0.9	0.49	0.63	0.81	1
–0.9	0.8	0.81	–0.72	0.64	1
0.6	–1	0.36	–0.6	1	1

Remembering that $x_3 = x_1^2$ and $x_4 = x_2^2$, we get the desired equation of the circle, as shown next:

$x_1^2 + x_2^2 = 1$

If we want to visualize this process geometrically, like we've done with the previous ones, it gets a little more complicated. Our nice two-dimensional dataset became a five-dimensional dataset. In this one, the points labelled 0 and 1 are now far away, and can be separated with a four-dimensional hyperplane. When we project this down to two dimensions, we get the desired circle.

The polynomial kernel gives rise to the map that sends the 2-D plane to the 5-D space. The map is the one that sends the point (x_1, x_2) to the point (x_1, x_2, x_1^2, x_1x_2, x_2^2). Because the maximum degree of each monomial is 2, we say that this is the polynomial kernel of degree 2. For the polynomial kernel, we always have to specify the degree.

What columns do we add to the dataset if we are using a polynomial kernel of higher degree, say, k? We add one column for each monomial in the given set of variables, of degree less than or equal to k. For example, if we are using the degree 3 polynomial kernel on the variables x_1 and x_2, we are adding columns corresponding to the monomials {x_1, x_2, x_1^2, x_1x_2, x_2^2, x_1^3, $x_1^2x_2$, $x_1x_2^2$, x_2^3}. We can also do this for more variables in the same way. For

example, if we use the degree 2 polynomial kernel on the variables x_1, x_2, and x_3, we are adding columns with the following monomials: $\{x_1, x_2, x_3, x_1^2, x_1x_2, x_1x_3, x_2^2, x_2x_3, x_3^2\}$.

Using bumps in higher dimensions to our benefit: The radial basis function (RBF) kernel

The next kernel that we'll see is the radial basis function kernel. This kernel is tremendously useful in practice, because it can help us build nonlinear boundaries using certain special functions centered at each of the data points. To introduce the RBF kernel, let's first look at the one-dimensional example shown in figure 11.19. This dataset is not linearly separable—the square lies exactly between the two triangles.

Figure 11.19 A dataset in one dimension that can't be classified by a linear classifier. Notice that a linear classifier is a point that divides the line into two parts, and there is no point that we can locate on the line that leaves all the triangles on one side and the square on the other side.

The way we will build a classifier for this dataset is to imagine building a mountain or a valley on each of the points. For the points labeled 1 (the triangles), we'll put a mountain, and for those labeled 0 (the square), we'll put a valley. These mountains and valleys are called *radial basis functions*. The resulting figure is shown at the top of figure 11.20. Now, we draw a mountain range such that at every point, the height is the sum of all the heights of the mountains and valleys at that point. We can see the resulting mountain range at the bottom of figure 11.20. Finally, the boundary of our classifier corresponds to the points at which this mountain range is at height zero, namely, the two highlighted points in the bottom. This classifier classifies anything in the interval between those two points as a square and everything outside of the interval as a triangle.

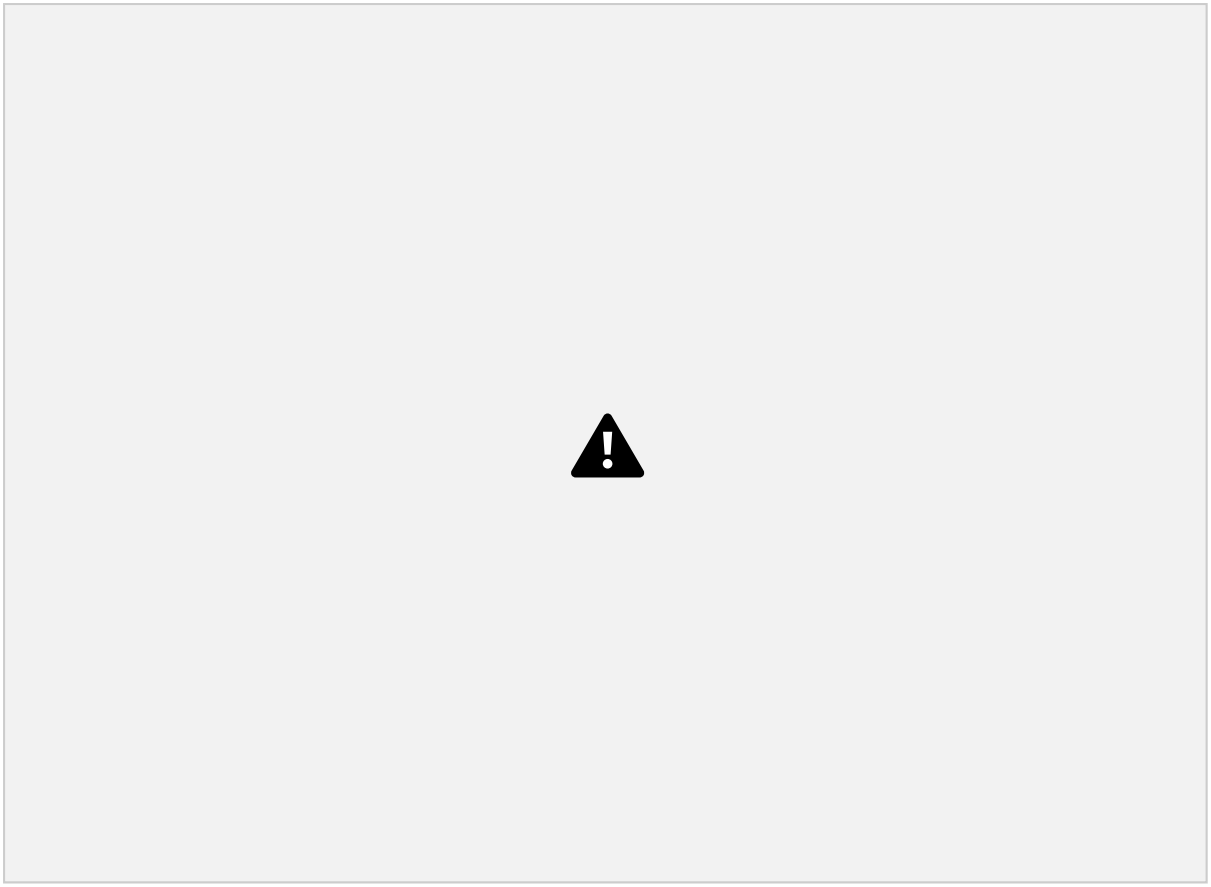


Figure 11.20 Using an SVM with the RBF kernel to separate a nonlinear dataset in one dimension. Top: We draw a mountain (radial basis function) at each point with label 1 and a valley at each point of label 0. Bottom: We add the radial basis functions from the top figure. The resulting function intersects the axis twice. The two points of intersection are the boundary of our SVM classifier. We classify each point between them as a square (label 0) and every point outside as a triangle (label 1).

This (plus some math around it which comes in the next section) is the essence of the RBF kernel. Now let's use it to build a similar classifier in a two-dimensional dataset.

To build the mountains and valleys on the plane, imagine the plane as a blanket (as illustrated in figure 11.21). If we pinch the blanket at that point and raise it, we get the mountain. If we push it down, we get the valley. These mountains and valleys are radial basis functions. They are called radial basis functions because the value of the function at a point is dependent only on the distance between the point and the center. We can raise the blanket at any point we like, and that gives us one different radial basis function for each point. The *radial basis function kernel* (also called RBF kernel) gives rise to a map that uses these radial functions to add several columns to our dataset in a way that will help us separate it.

Figure 11.21 A radial basis function consists of raising the plane at a particular point. This is the family of functions that we'll use to build nonlinear classifiers. (Source: Image created with the assistance of Grapher™ from Golden Software, LLC; https://www.goldensoftware.com/products/grapher).

How do we use this as a classifier? Imagine the following: we have the dataset on the left of figure 11.22, where, as usual, the triangles represent points with label 1, and the squares represent points with label 0. Now, we lift the plane at every triangle and push it down at every square. We get the three-dimensional plot shown on the right of figure 11.22.

To create the classifier, we draw a plane at height 0 and intersect it with our surface. This is the same as looking at the curve formed by the points at height 0. Imagine if there is a landscape with mountains and the sea. The curve will correspond to the coastline, namely, where the water and the land meet. This coastline is the curve shown on the left in figure 11.23. We then project everything back to the plane and obtain our desired classifier, shown on the right in figure 11.23.

That is the idea behind the RBF kernel. Of course, we have to develop the math, which we will do in the next few sections. But in principle, if we can imagine lifting and pushing down a blanket, and then building a classifier by looking at the boundary of the points that lie at a particular height, then we can understand what an RBF kernel is.

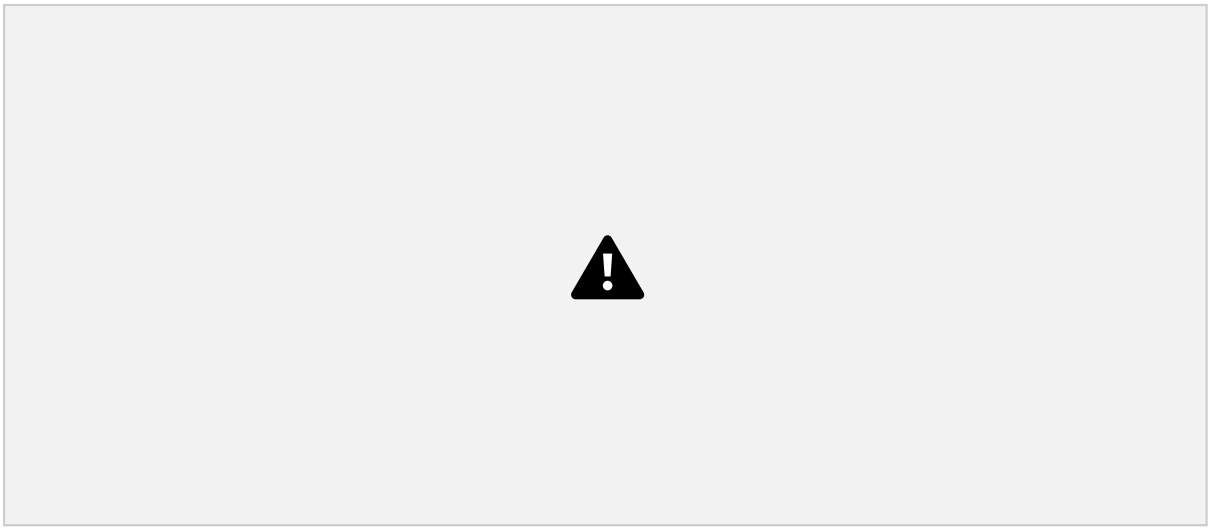


Figure 11.22 Left: A dataset in the plane that is not linearly separable. Right: We have used the radial basis functions to raise each of the triangles and lower each of the squares. Notice that now we can separate the dataset by a plane, which means our modified dataset is linearly separable. (Source: Image created with the assistance of Grapher™ from Golden Software, LLC; https://www.goldensoftware.com/products/grapher).

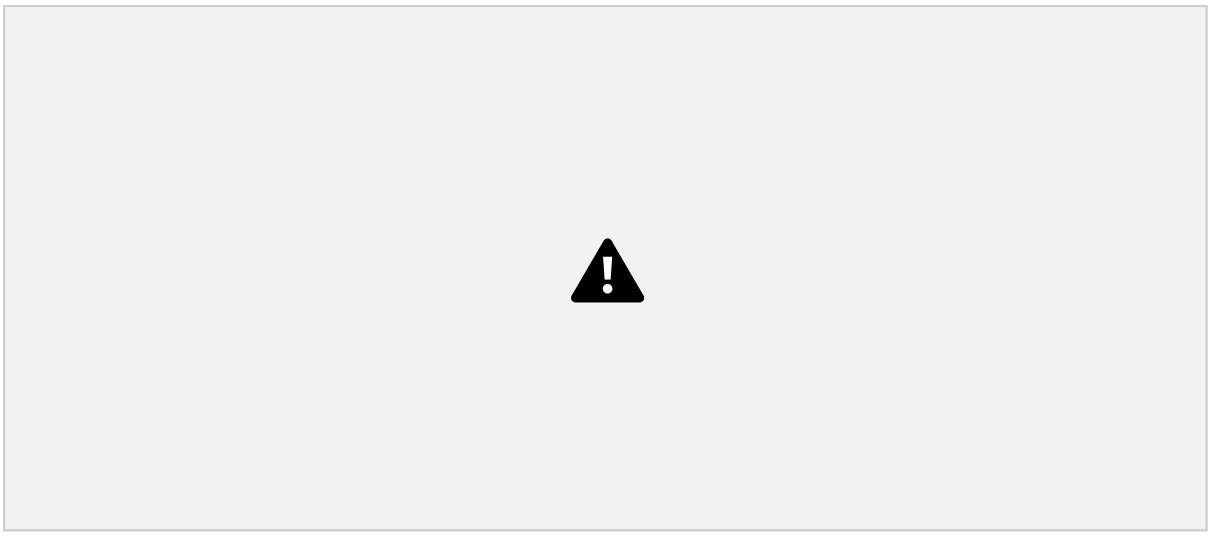


Figure 11.23 Left: If we look at the points at height 0, they form a curve. If we think of the high points as land and the low points as the sea, this curve is the coastline. Right: When we project (flatten) the points back to the plane, the coastline is now our classifier that separates the triangles from the squares. (Source: Image created with the assistance of Grapher™ from Golden Software, LLC; https://www.goldensoftware.com/products/grapher).

A more in-depth look at radial basis functions

Radial basis functions can exist in any number of variables. At the beginning of this section, we saw them in one and two variables. For one variable, the simplest radial basis function has the formula $y = e^{-x^2}$. This looks like a bump over the line (figure 11.24). It looks a lot like a standard normal (Gaussian) distribution. The standard normal distribution is similar, but it has a slightly different formula, so that the area underneath it is 1.

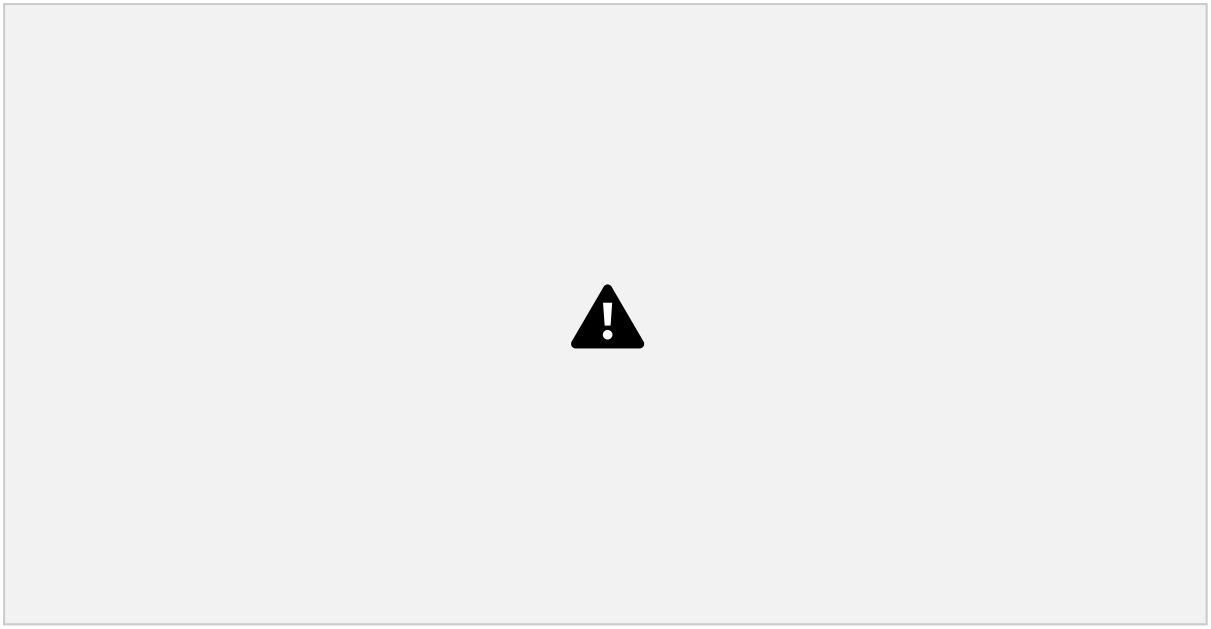


Figure 11.24 An example of a radial basis function. It looks a lot like a normal (Gaussian) distribution.

Notice that this bump happens at 0. If we wanted it to appear at any different point, say p, we can translate the formula and get $y = e^{-(x-p)2}$. For example, the radial basis function centered at the point 5 is precisely $y = e^{-(x-5)2}$.

For two variables, the formula for the most basic radial basis function is $z = e^{-(x2+y2)}$, and it looks like the plot shown in figure 11.25. Again, you may notice that it looks a lot like a multivariate normal distribution. It is, again, a modified version of the multivariate normal distribution.

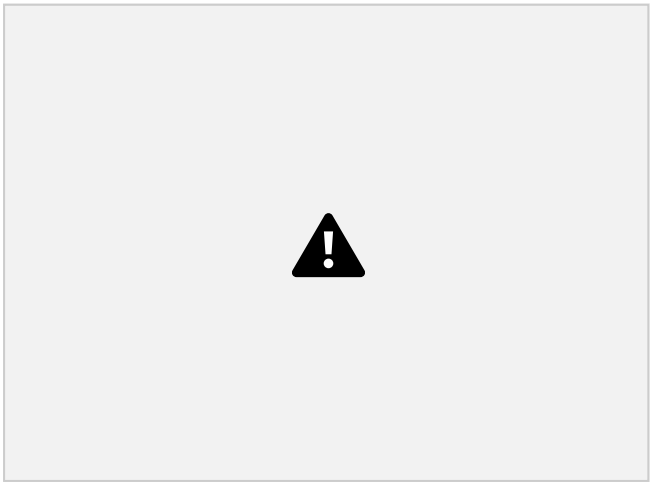


Figure 11.25 A radial basis function on two variables. It again looks a lot like a normal distribution. (Source: Image created with the assistance of Grapher™ from Golden Software, LLC; https://www.goldensoftware.com/products/grapher).

This bump happens exactly at the point (0,0). If we wanted it to appear at any different point, say (*p, q*), we can translate the formula, and get $y = e^{-[(x-p)2+(y-q)2]}$. For example, the radial basis function centered at the point (2, –3) is precisely $y = e^{-[(x-2)2+(y+3)2]}$.

For n variables, the formula for the basic radial basis function is $y = e^{-(x_1 2+\cdots+x_n 2)}$. We can't draw a plot in $n + 1$ dimensions, but if we imagine pinching an n-dimensional blanket and lifting it up with our fingers, that's how it looks. However, because the algorithm that we use is purely mathematical, the computer has no trouble running it in as many variables as we want. As usual, this n-dimensional bump is centered at 0, but if we wanted it centered at the point $(p_1, \ldots, p_n)$, the formula is $y = e^{-[(x_1 - p_1)2+\cdots+(x_n - p_n)2]}$

A measure of how close points are: Similarity

To build an SVM using the RBF kernel, we need one notion: the notion of *similarity*. We say that two points are similar if they are close to each other, and not similar if they are far away (figure 11.26). In other words, the similarity between two points is high if they are close to each other and low if they are far away from each other. If the pair of points are the same point, then the similarity is 1. In theory, the similarity between two points that are an infinite distance apart is 0.

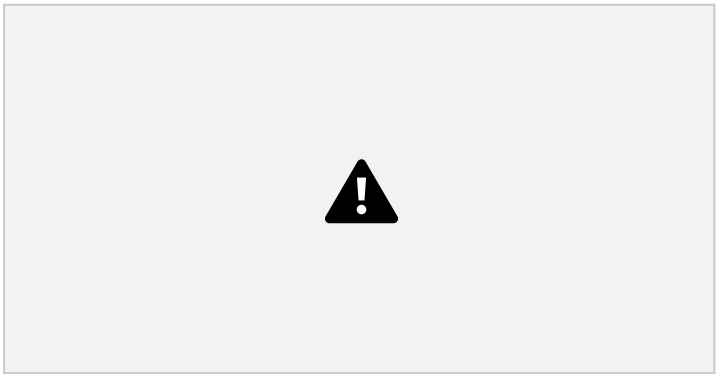


Figure 11.26 Two points that are close to each other are defined to have high similarity. Two points that are far away are defined to have low similarity.

Now we need to find a formula for similarity. As we can see, the similarity between two points decreases as the distance between them increases. Thus, many formulas for similarity would work, as long as they satisfy that condition. Because we are using exponential functions in this section, let's define it as follows. For points p and q, the similarity between p and q is as follows:

$similarity(p,q) = e^{-\text{distance}(p,q)2}$

That looks like a complicated formula for similarity, but there is a very nice way to look at it. If we want to find the similarity between two points, say p and q, this similarity is precisely the height of the radial basis function centered at p and applied at the point q. This is, if we pinch the blanket at point p and lift it, then the height of the blanket at point q is high if the q is close to p and low if q is far from p. In figure 11.27, we can see this for one variable, but imagine it in any number of variables by using the blanket analogy.

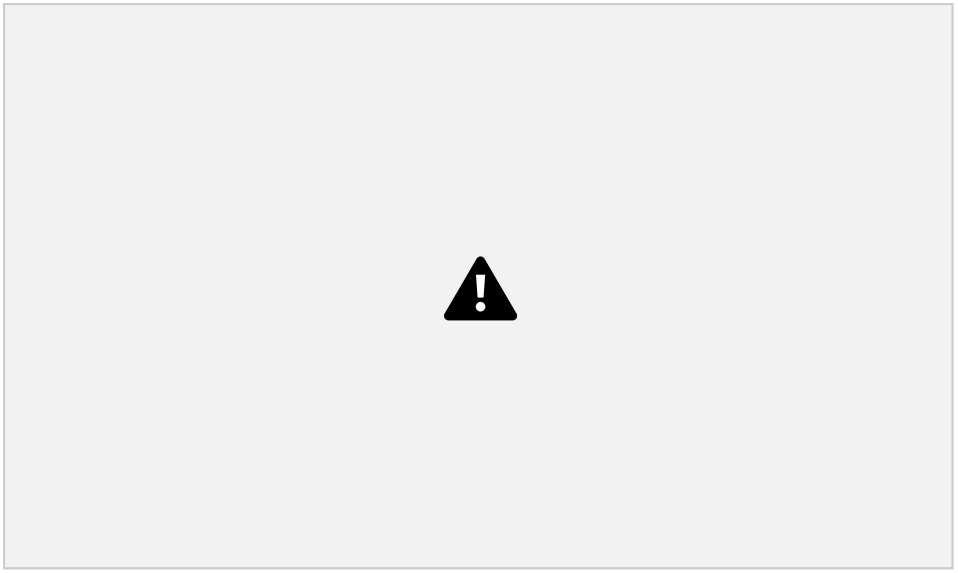


Figure 11.27 The similarity is defined as the height of a point in the radial basis function, where the input is the distance. Note that the higher the distance, the lower the similarity, and vice versa.

Training an SVM with the RBF kernel

Now that we have all the tools to train an SVM using the RBF kernel, let's see how to put it all together. Let's first look at the simple dataset displayed in figure 11.19. The dataset itself appears in table 11.6.

Table 11.6 The one-dimensional dataset shown in figure 11.19. Note that it isn't linearly separable, because the point with label 0 is right between the two points with label 1.

Point	x	y (label)
1	–1	1
2	0	0
3	1	1

As we saw, this dataset is not linearly separable. To make it linearly separable, we'll add a few columns. The three columns we are adding are the similarity columns, and they record the similarity between the points. The similarity between two points with x-coordinates x_1 and x_2 is measured as $e^{(x_1 + x_2)2}$, as indicated in the section "Using bumps in higher dimensions to our benefit." For example, the similarity between points 1 and 2 is $e^{(-1\ -0)2} = 0.368$. In the Sim1 column, we'll record the similarity between point 1 and the other three points, and so on. The extended dataset is shown in table 11.7.

Table 11.7 We extend the dataset in table 11.6 by adding three new columns. Each column corresponds to the similarity of all points with respect to each point. This extended dataset lives in a four-dimensional space, and it is linearly separable.

Point	x	Sim1	Sim2	Sim3	y
1	–1	1	0.368	0.018	1
2	0	0.368	1	0.368	0
3	1	0.018	0.368	1	1

This extended dataset is now linearly separable! Many classifiers will separate this set, but in particular, the one with the following boundary equation will:

$\hat{y} = step(Sim1 - Sim2 + Sim3)$

Let's verify this by predicting the label at every point as shown next:

- **Point 1**: $\hat{y} = step(1 - 0.368 + 0.018) = step(0.65) = 1$
- **Point 2**: $\hat{y} = step(0.368 - 1 + 0.368) = step(-0.264) = 0$
- **Point 3**: $\hat{y} = step(0.018 - 0.368 + 1) = step(0.65) = 1$

Furthermore, because $Sim1=e^{(x+1)2}$, $Sim2=e^{(x-0)2}$, and $Sim3=e^{(x-1)2}$ then our final classifier makes the following predictions:

$\hat{y} = step(e^{(x+1)2} - e^{x2} + e^{(x-1)2})$

Now, let's do this same procedure but in two dimensions. This section does not require code, but the calculations are large, so if you'd like to take a look at them, they are in the following notebook: https://github.com/luisguiserrano/manning/blob/master/Chapter_11_Support_Vector_Machines/Calculating_similarities.ipynb.

Table 11.8 A simple dataset in two dimensions, plotted in figure 11.28. We'll use an SVM with an RBF kernel to classify this dataset.

Point	x_1	x_2	y
1	0	0	0
2	–1	0	0
3	0	–1	0
4	0	1	1
5	1	0	1
6	–1	1	1
7	1	–1	1

Consider the dataset in table 11.8, which we already classified graphically (figures 11.22 and 11.23). For convenience, it is plotted again in figure 11.28. In this plot, the points with label 0 appear as squares and those with label 1 as triangles.

Notice that in the first column of table 11.8 and in figure 11.28, we have numbered every point. This is not part of the data; we did it only for convenience. We will now add seven columns to this table. The columns are the similarities with respect to every point. For example, for point 1, we add a similarity column named Sim1. The entry for every point in this column is the amount of similarity between that point and point 1. Let's calculate one of them, for example, the similarity with point 6. The distance between point 1 and point 6, by the Pythagorean theorem follows:

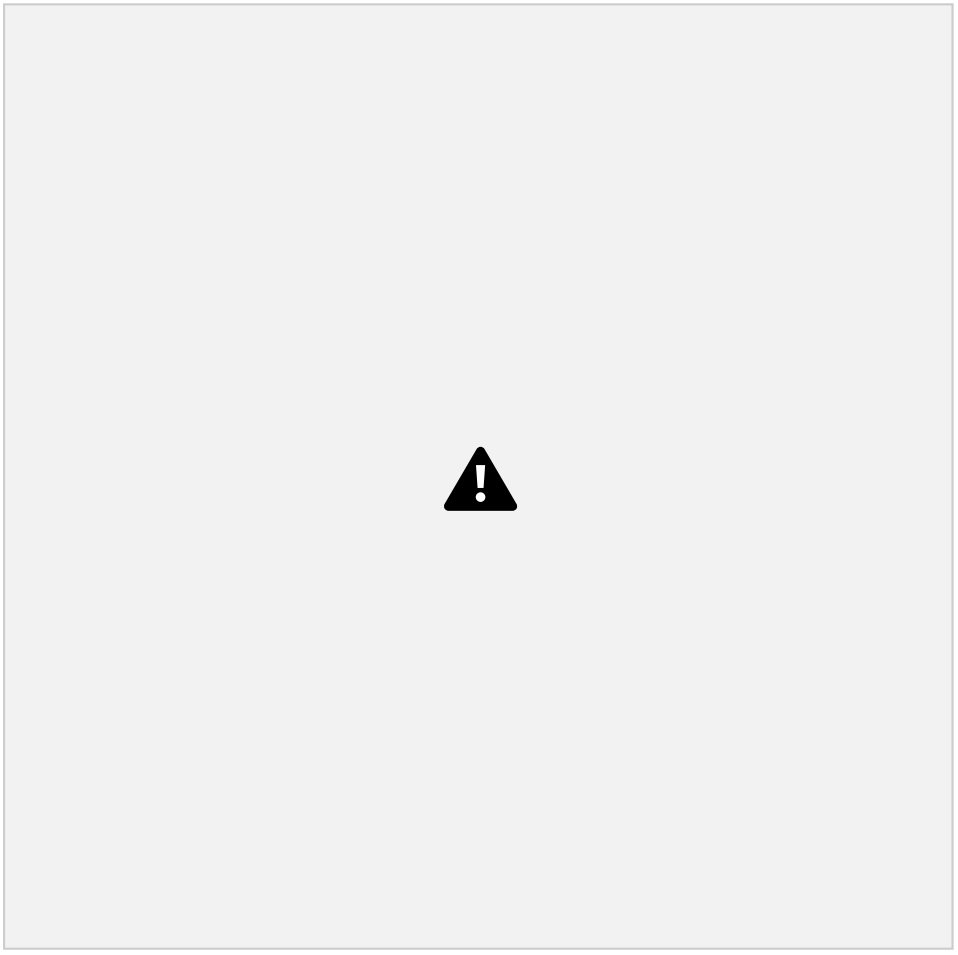


Figure 11.28 The plot of the dataset in table 11.8, where the points with label 0 are squares and those with label 1 are triangles. Notice that the squares and triangles cannot be separated with a line. We'll use an SVM with an RBF kernel to separate them with a curved boundary.

Therefore, the similarity is precisely

$similarity(point\ 1, point\ 6) = e^{-distance(q,p)2} = e^{-2} = 0.135.$

This number goes in row 1 and column Sim6 (and by symmetry, also in row 6 and column Sim1). Fill in a few more values in this table to convince yourself that this is the case, or take a look at the notebook where the whole table is calculated. The result is shown in table 11.9.

Table 11.9 We have added seven similarity columns to the dataset in table 11.8. Each one records the similarities with all the other six points.

Point	x_1	x_2	Sim1	Sim2	Sim3	Sim4	Sim5	Sim6	Sim7	y
1	0	0	1	0.368	0.368	0.368	0.368	0.135	0.135	0
2	–1	0	0.368	1	0.135	0.135	0.018	0.368	0.007	0
3	0	–1	0.368	0.135	1	0.018	0.135	0.007	0.368	0
4	0	1	0.368	0.135	0.018	1	0.135	0.368	0.007	1
5	1	0	0.368	0.018	0.135	0.135	1	0.007	0.368	1
6	–1	1	0.135	0.368	0.007	0.367	0.007	1	0	1
7	1	–1	0.135	0.007	0.368	0.007	0.368	0	1	1

Notice the following things:

1. The similarity between each point and itself is always 1.
2. For each pair of points, the similarity is high when they are close in the plot and low when they are far.
3. The table consisting of the columns Sim1 to Sim7 is symmetric, because the similarity between p and q is the same as the similarity between q and p (as it depends only on the distance between p and q).
4. The similarity between points 6 and 7 appears as 0, but in reality, it is not. The distance between points 6 and 7 is , so their similarity is e^{-8} = 0.00033546262, which rounds to zero because we are using three significant figures.

Now, on to building our classifier! Notice that for the data in the small table 11.8, no linear classifier works (because the points can't be split by a line), but on the much larger table 11.9, which has a lot more features (columns), we can fit such a classifier. We proceed to fit an SVM to this data. Many SVMs can classify this dataset correctly, and in the notebook, I've used Turi Create to build one. However, a simpler one works as well. This classifier has the following weights:

- The weights of x_1 and x_2 are 0.
- The weight of Sim p is 1, for p = 1, 2, and 3.
- The weight of Sim p is –1, for p = 4, 5, 6, and 7.
- The bias is b = 0.

We find the classifier was adding a label –1 to the columns corresponding to the points labeled 0, and a +1 to the columns corresponding to the points labeled 1. This is equivalent to the process of adding a mountain at any point of label 1 and a valley at every point of label 0, like in figure 11.29. To check mathematically that this works, take table 11.7, add the values of the columns Sim4, Sim5, Sim6, and Sim7, then subtract the values of the columns Sim1, Sim2 and Sim3. You'll notice that you get a negative number in the first three rows and a positive one in the last four rows. Therefore, we can use a threshold of 0, and we have a classifier that classifies this dataset correctly, because the points labeled 1 get a positive score, and the points labeled 0 get a negative score. Using a threshold of 0 is equivalent to using the coastline to separate the points in the plot in figure 11.29.

If we plug in the similarity function, the classifier we obtain is the following:

$$\hat{y} = step(-e^{x_1^2 + x_2^2} - e^{(x_1 + 1)^2 + x_2^2} - e^{x_1^2 + (x_2 + 1)^2} + e^{x_1^2 + (x_2 - 1)^2} + e^{(x_1 - 1)^2 + x_2^2} + e^{(x_1 + 1)^2 + (x_2 - 1)^2} + e^{(x_1 - 1)^2 + (x_2 + 1)^2})$$

In summary, we found a dataset that was not linearly separable. We used radial basis functions and similarity between points to add several columns to the dataset. This helped us build a linear classifier (in a much higher-dimensional space). We then projected the higher-dimensional linear classifier into the plane to get the classifier we wanted. We can see the resulting curved classifier in figure 11.29.

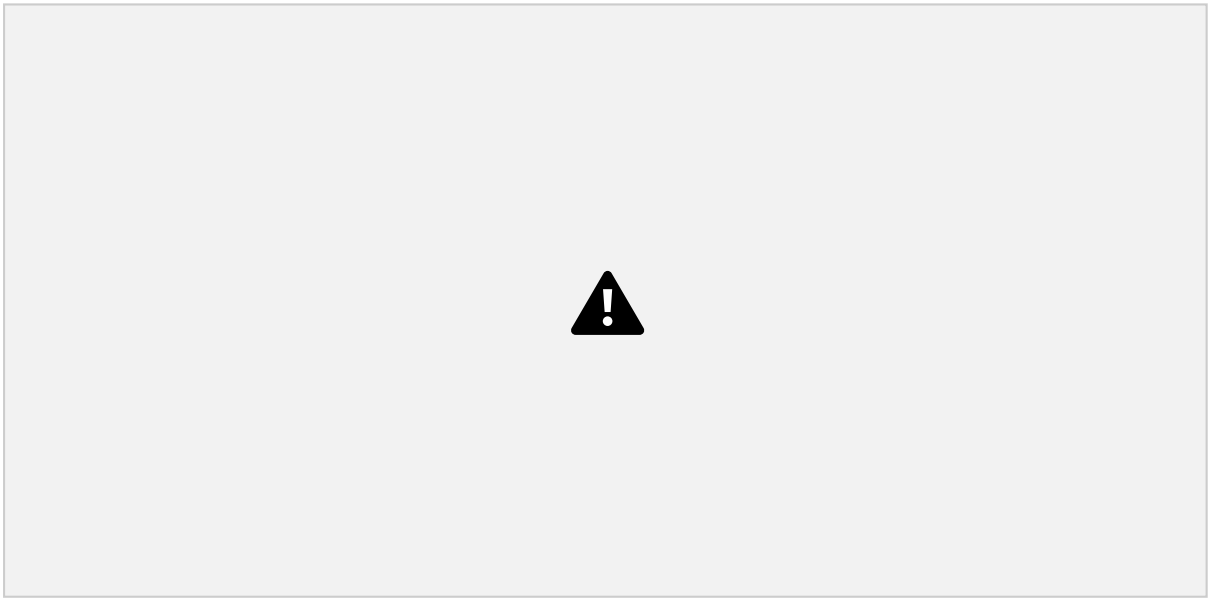


Figure 11.29 In this dataset, we raised each triangle and lowered each square. Then we drew a plane at height 0, which separates the squares and the triangles. The plane intersects the surface in a curved boundary. We then projected everything back down to two dimensions, and this curved boundary is the one that separates our triangles from our squares. The boundary is drawn at the right. (Source: Image created with the assistance of Grapher™ from Golden Software, LLC; https://www.goldensoftware.com/products/grapher).

Overfitting and underfitting with the RBF kernel: The gamma parameter

At the beginning of this section, we mentioned that many different radial basis functions exist, namely one per point in the plane. There are actually many more. Some of them lift the plane at a point and form a narrow surface, and others form a wide surface. Some examples can be seen in figure 11.30. In practice, the wideness of our radial basis functions is something we want to tune. For this, we use a parameter called the *gamma parameter*. When gamma is small, the surface formed is very wide, and when it is large, the surface is very narrow.

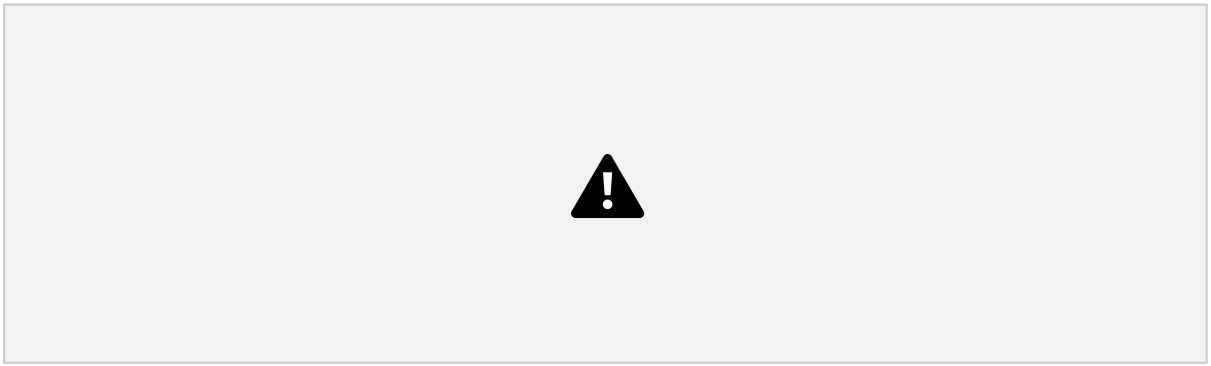


Figure 11.30 The gamma parameter determines how wide the surface is. Notice that for small values of gamma, the surface is very wide, and for large values of gamma, the surface is very narrow. (Source: Image created with the assistance of Grapher™ from Golden Software, LLC; https://www.goldensoftware.com/products/grapher).

Gamma is a hyperparameter. Recall that hyperparameters are the specifications that we use to train our model. The way we tune this hyperparameter is using methods that we've seen before, such as the model complexity graph (the section "A numerical way to decide how complex our model should be" in chapter 4). Different values of gamma tend to overfit and underfit. Let's look back at the example at the beginning of this section, with three different values of gamma. The three models are plotted in figure 11.31.

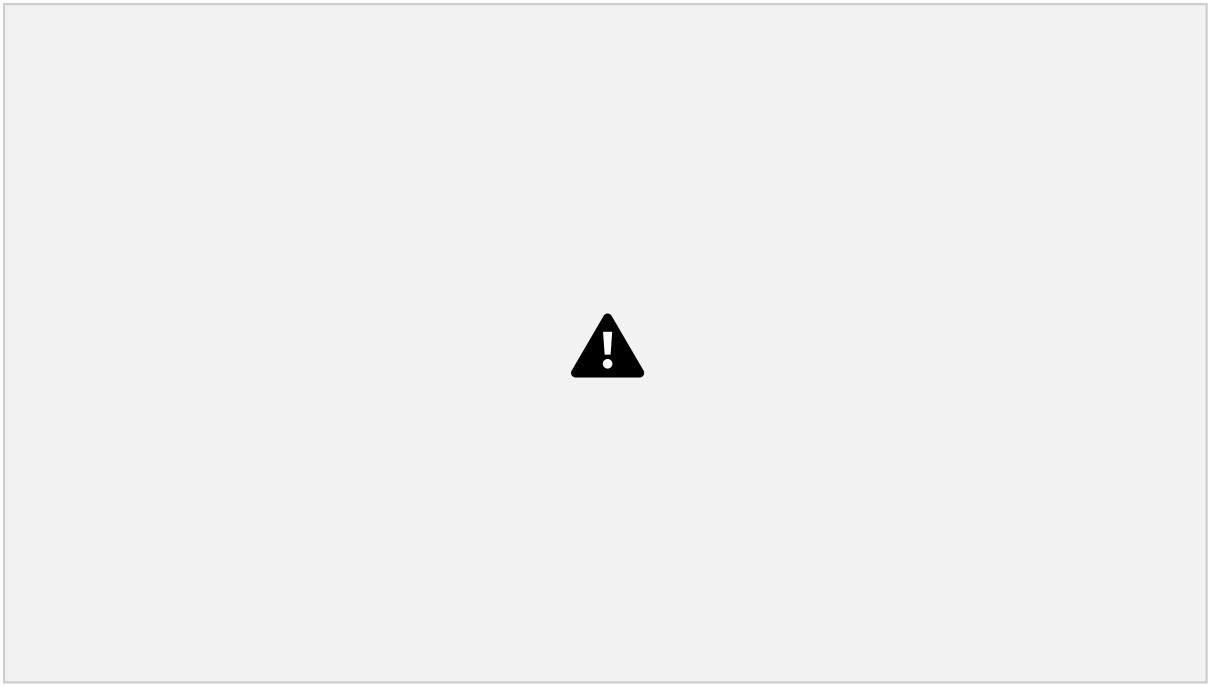


Figure 11.31 Three SVM classifiers shown with an RBF kernel and different values of gamma. (Source: Image created with the assistance of Grapher™ from Golden Software, LLC; https://www.goldensoftware.com/products/grapher).

Notice that for a very small value of gamma, the model overfits, because the curve is too simple, and it doesn't classify our data well. For a large value of gamma, the model vastly overfits, because it builds a tiny mountain for each triangle and a tiny valley for each square. This makes it classify almost everything as a square, except for the areas just around the triangles. A medium value of gamma seems to work well, because it builds a boundary that is simple enough, yet classifies the points correctly.

The equation for the radial basis function doesn't change much when we add the gamma parameter—all we have to do is multiply the exponent by gamma. In the general case, the equation of the radial basis function follows:

$$y = e^{-\gamma[(x_1 - p_1)2 + \cdots + (x_n + p_n)2]}$$

Don't worry very much about learning this formula—just remember that even in higher dimensions, the bumps we make can be wide or narrow. As usual, there is a way to code this and make it work, which is what we do in the next section.

Coding the kernel method

Now that we've learned the kernel method for SVMs, we learn code them in Scikit-Learn and train a model in a more complex dataset using the polynomial and RBF kernels. To train an SVM in Scikit-Learn with a particular kernel, all we do is add the kernel as a parameter when we define the SVM. The code for this section follows:

- **Notebook**: SVM_graphical_example.ipynb
    - https://github.com/luisguiserrano/manning/blob/master/Chapter_11_Support_Vector_Machines/SVM_graphical_example.ipynb
- Datasets:
    - one_circle.csv
    - two_circles.csv

Coding the polynomial kernel to classify a circular dataset

In this subsection, we see how to code the polynomial kernel in Scikit-Learn. For this, we use the dataset called one_circle.csv, shown in figure 11.32.

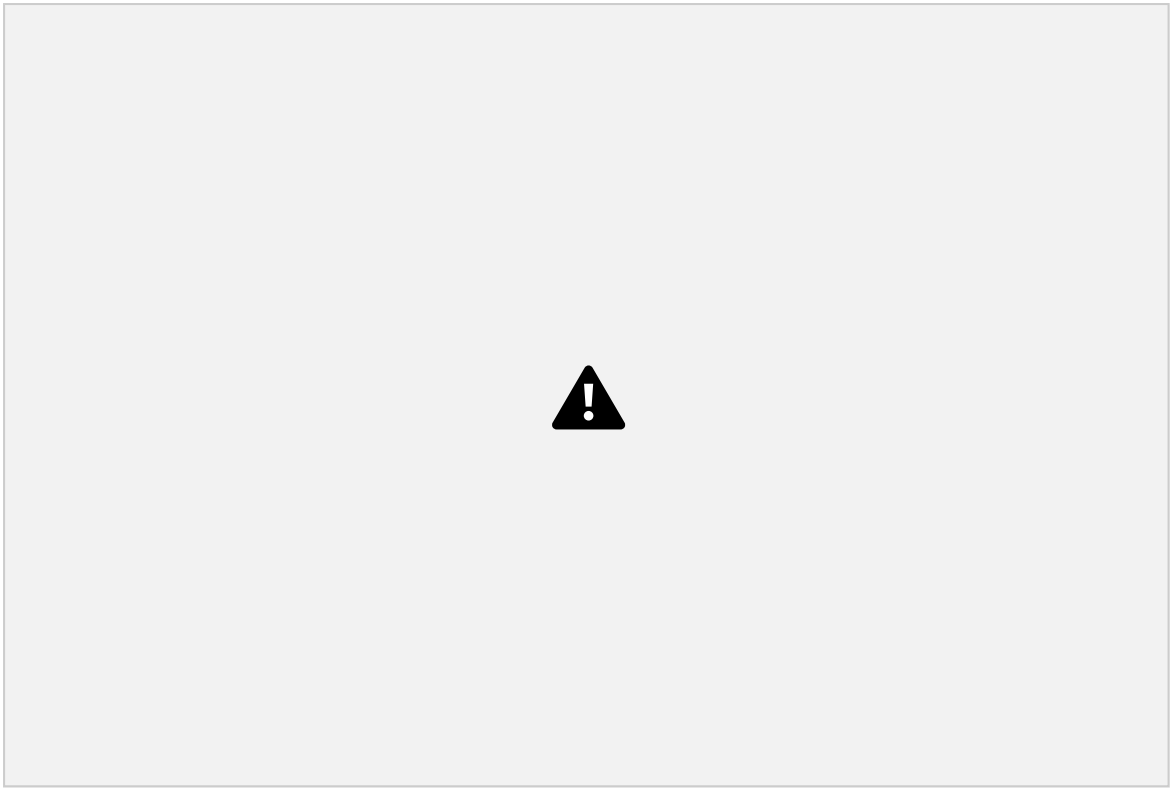


Figure 11.32 A circular dataset, with some noise. We will use an SVM with the polynomial kernel to classify this dataset.

Notice that aside from some outliers, this dataset is mostly circular. We train an SVM classifier where we specify the `kernel` parameter to be `poly`, and the `degree` parameter to be 2, as shown in the next code snippet. The reason we want the degree to be 2 is because the equation of a circle is a polynomial of degree 2. The result is shown in figure 11.33.

```
svm_degree_2 = SVC(kernel='poly', degree=2)

svm_degree_2.fit(features, labels)
```

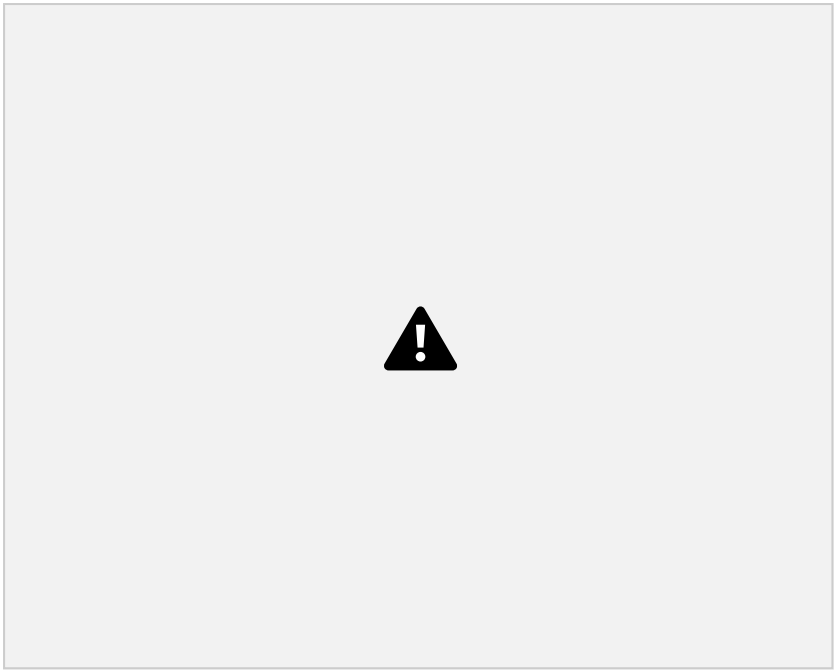


Figure 11.33 An SVM classifier with a polynomial kernel of degree 2

Notice that this SVM with a polynomial kernel of degree 2 manages to build a mostly circular region to bound the dataset, as desired.

Coding the RBF kernel to classify a dataset formed by two intersecting circles and playing with the gamma parameter

We've drawn a circle, but let's get more complicated. In this subsection, we learn how to code several SVMs with the RBF kernel to classify a dataset that has the shape of two intersecting circles. This dataset, called two_circles.csv, is illustrated in figure 11.34.

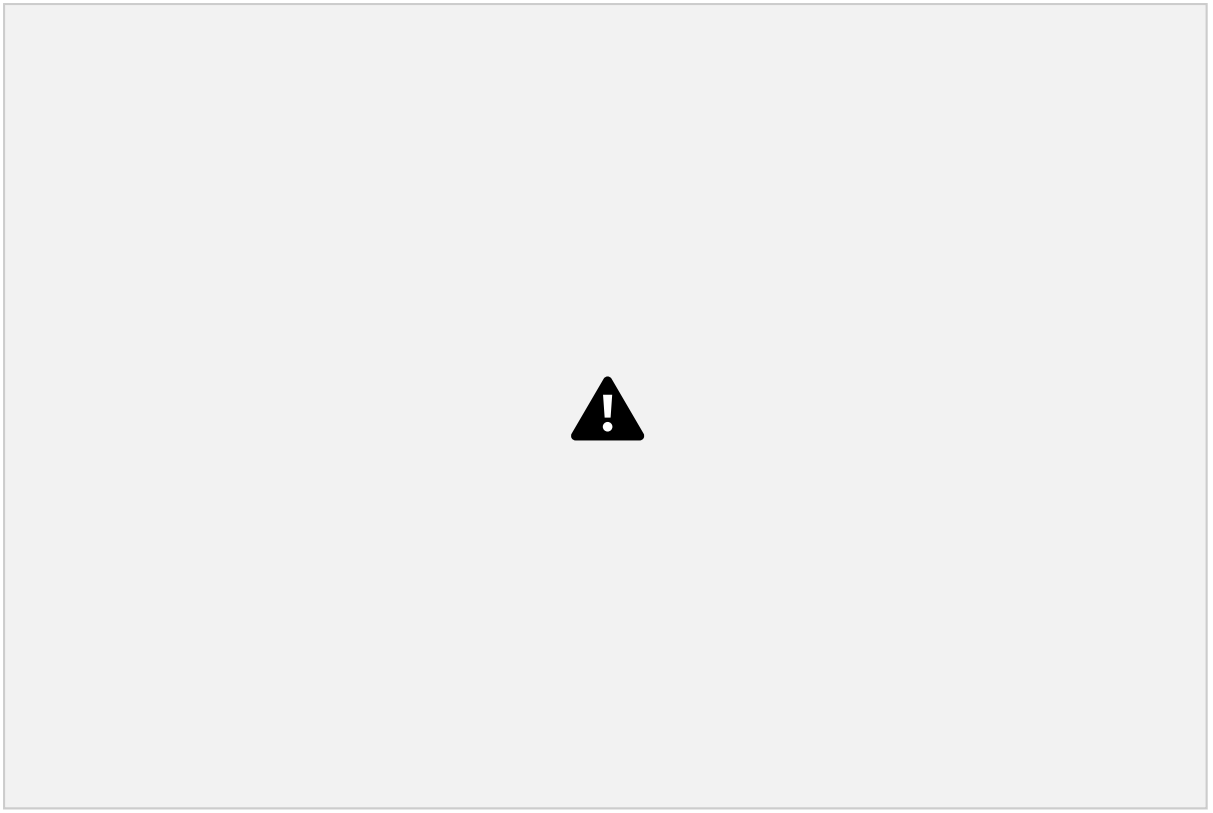


Figure 11.34 A dataset consisting of two intersecting circles, with some outliers. We will use an SVM with the RBF kernel to classify this dataset.

To use the RBF kernel, we specify `kernel = 'rbf'`. We can also specify a value for gamma. We'll train four different SVM classifiers, for the following values of gamma: 0.1, 1, 10, and 100, as shown next:

```
svm_gamma_01 = SVC(kernel='rbf', gamma=0.1)  ❶

svm_gamma_01.fit(features, labels)


svm_gamma_1 = SVC(kernel='rbf', gamma=1)     ❷

svm_gamma_1.fit(features, labels)


svm_gamma_10 = SVC(kernel='rbf', gamma=10)   ❸

svm_gamma_10.fit(features, labels)


svm_gamma_100 = SVC(kernel='rbf', gamma=100) ❹

svm_gamma_100.fit(features, labels)
```

❶ Gamma = 0.1

❷ Gamma = 1

❸ Gamma = 10

❹ Gamma = 100

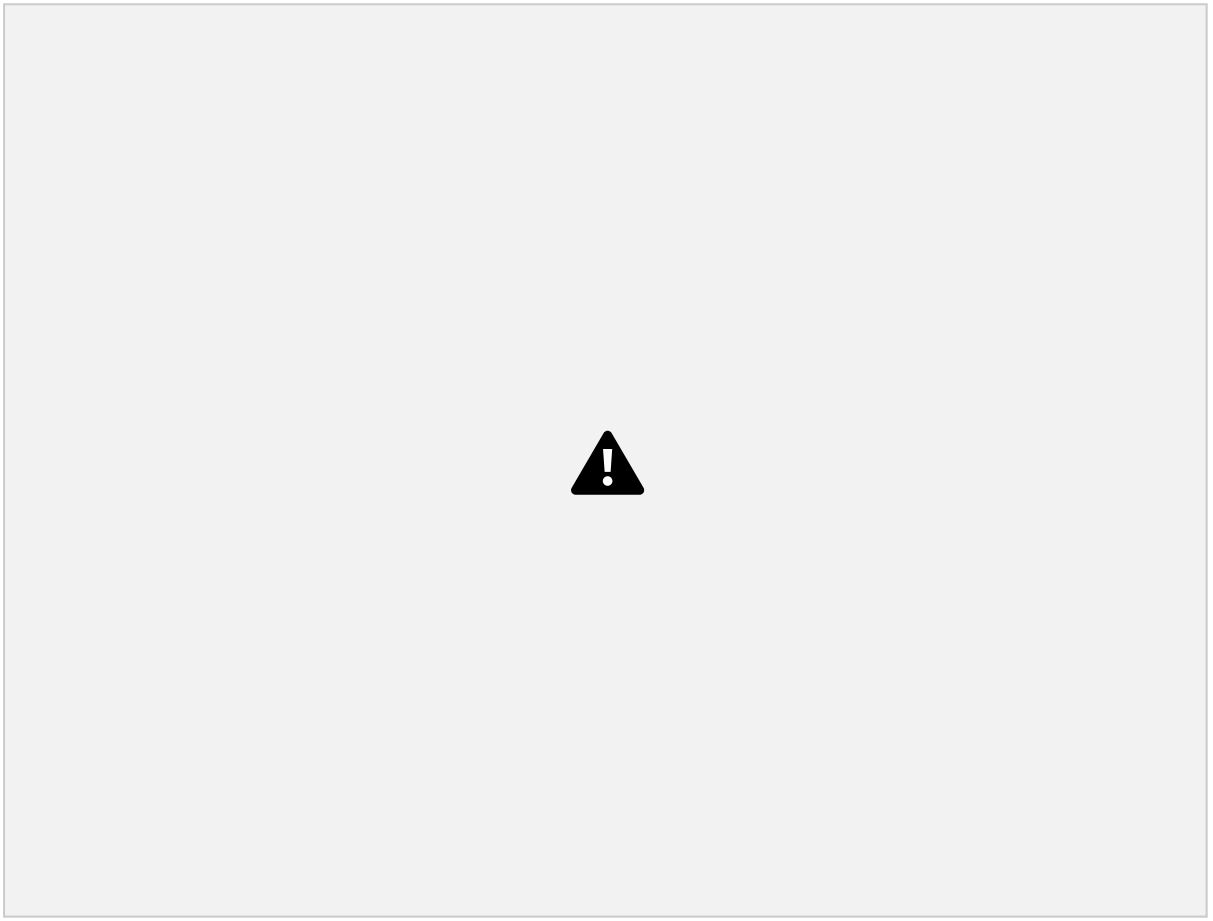


Figure 11.35 Four SVM classifiers with an RBF kernel and different values of gamma

The four classifiers appear in figure 11.35. Notice that for gamma = 0.1, the model underfits a little, because it thinks the boundary is one oval, and it makes some mistakes. Gamma = 1 gives a good model that captures the data well. By the time we get to gamma = 10, we can see that the model starts to overfit. Notice how it tries to classify every point correctly, including the outliers, which it encircles individually. By the time we get to gamma=100, we can see some serious overfitting. This classifier only surrounds each triangle with a small circular region and classifies everything else as a square. Thus, for this model, gamma = 1 seems to be the best value among the ones we tried.

Summary

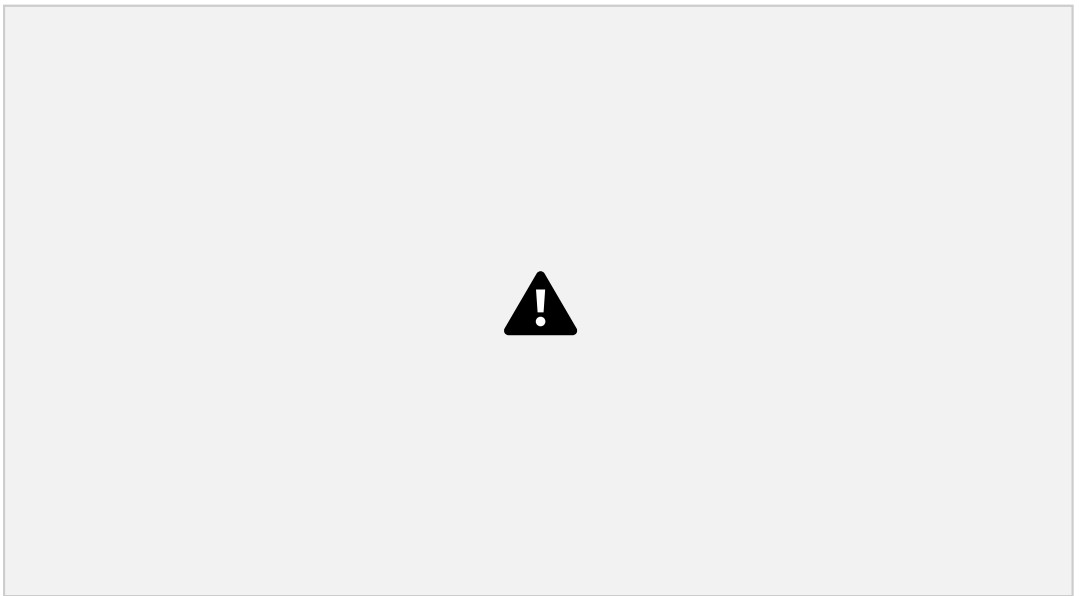
- A support vector machine (SVM) is a classifier that consists of fitting two parallel lines (or hyperplanes), and trying to space them as far apart as possible, while still trying to classify the data correctly.
- The way to build support vector machines is with an error function that comprises two terms: the sum of two perceptron errors, one per parallel line, and the distance error, which is high when the two parallel lines are far apart and low when they are close together.
- We use the C parameter to regulate between trying to classify the points correctly and trying to space out the lines. This is useful while training because it gives us control over our preferences, namely, if we want to build a classifier that classifies the data very well, or a classifier with a well-spaced boundary.
- The kernel method is a useful and very powerful tool for building nonlinear classifiers.
- The kernel method consists of using functions to help us embed our dataset inside a higher-dimensional space, in which the points may be easier to classify with a linear classifier. This is equivalent to adding columns to our dataset in a clever way to make the enhanced dataset linearly separable.
- Several different kernels, such as the polynomial kernel and the RBF kernel, are available. The polynomial kernel allows us to build polynomial regions such as circles, parabolas, and hyperbolas. The RBF kernel allows us to build more complex curved regions.

Exercises

Exercise 11.1

(This exercise completes the calculation needed in the section “Distance error function.”)

Show that the distance between the lines with equations $w_1x_1 + w_2x_1 + b = 1$ and $w_1x_1 + w_2x_1 + b = -1$ is precisely  .



Exercise 11.2

As we learned in exercise 5.3, it is impossible to build a perceptron model that mimics the XOR gate. In other words, it is impossible to fit the following dataset (with 100% accuracy) with a perceptron model:

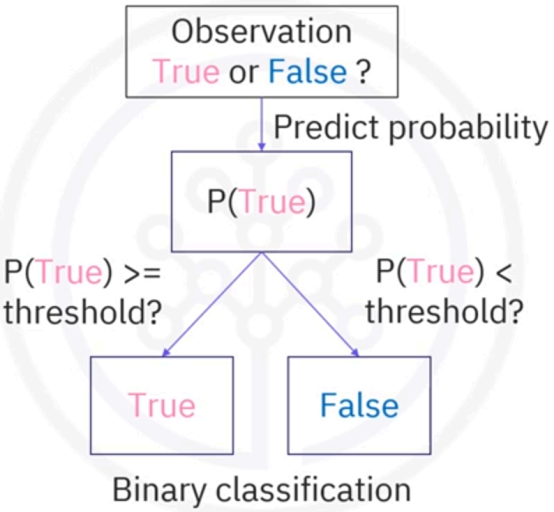
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

This is because the dataset is not linearly separable. An SVM has the same problem, because an SVM is also a linear model. However, we can use a kernel to help us out. What kernel should we use to turn this dataset into a linearly separable one? What would the resulting SVM look like?

hint Look at example 2 in the section “Using polynomial equations to your benefit,” which solves a very similar problem.

Logistic Regression.

What is logistic regression?



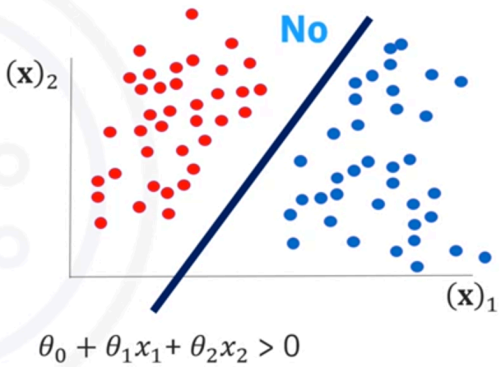
When is logistic regression a good choice?

- If the data is linearly separable, the decision boundary of logistic regression is a line, a plane, or a hyperplane

Example: $\theta_0 + \theta_1 x_1 + \theta_2 x_2 > 0$

- To understand the impact of an independent feature

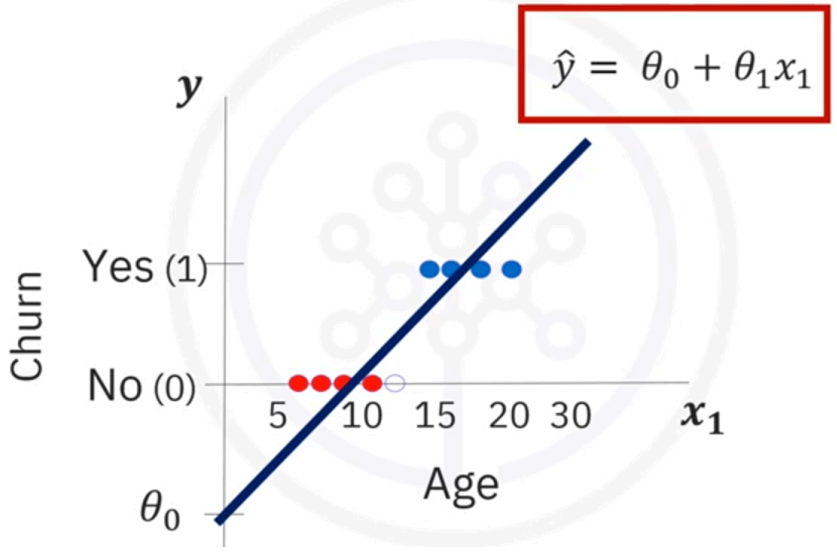
Example: Select features based on model coefficient size or weights



Logistic regression applications

- Predicting heart attack risk
- Diagnosing patients based on a set of characteristics
- Predicting whether a customer will purchase a product or halt a subscription
- Predicting product failure probability
- Predicting mortgage default likelihood

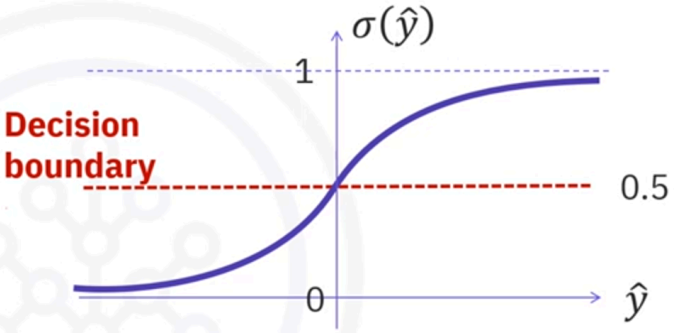
Predicting churn using linear regression



Probabilities to class predictions

$$\widehat{p} = \sigma(\hat{y}) = \frac{1}{1 + e^{-\hat{y}}}$$

Probability that class is 1



$$\sigma(\hat{y}) \rightarrow \begin{cases} 0 & \text{if } \sigma(\hat{y}) < 0.5 \\ 1 & \text{if } \sigma(\hat{y}) \geq 0.5 \end{cases}$$

In the previous chapter, we built a classifier that determined if a sentence was happy or sad. But as we can imagine, some sentences are happier than others. For example, the sentence "I'm good" and the sentence "Today was the most wonderful day in my life!" are both happy, yet the second is much happier than the first. Wouldn't it be nice to have a classifier that not only predicts if sentences are happy or sad but that gives a rating for how happy sentences are—say, a classifier that tells us that the first sentence is 60% happy and the second one is 95% happy? In this chapter, we define the *logistic classifier*, which does precisely that. This

classifier assigns a score from 0 to 1 to each sentence, in a way that the happier a sentence is, the higher the score it receives.

In a nutshell, a logistic classifier is a type of model that works just like a perceptron classifier, except instead of returning a yes or no answer, it returns a number between 0 and 1. In this case, the goal is to assign scores close to 0 to the saddest sentences, scores close to 1 to the happiest sentences, and scores close to 0.5 to neutral sentences. This threshold of 0.5 is common in practice, though arbitrary. In chapter 7, we'll see how to adjust it to optimize our model, but for this chapter we use 0.5.

This chapter relies on chapter 5, because the algorithms we develop here are similar, aside from some technical differences. Making sure you understand chapter 5 well will help you understand the material in this chapter. In chapter 5, we described the perceptron algorithm using an error function that tells us how good a perceptron classifier is and an iterative step that turns a classifier into a slightly better classifier. In this chapter, we learn the logistic regression algorithm, which works in a similar way. The main differences follow:

- The step function is replaced by a new activation function, which returns values between 0 and 1.
- The perceptron error function is replaced by a new error function, which is based on a probability calculation.
- The perceptron trick is replaced by a new trick, which improves the classifier based on this new error function.

aside In this chapter we carry out a lot of numerical computations. If you follow the equations, you might find that your calculations differ from those in the book by a small amount. The book rounds the numbers at the very end of the equation, not in between steps. This, however, should have very little effect on the final results.

At the end of the chapter, we apply our knowledge to a real-life dataset of movie reviews on the popular site IMDB (www.imdb.com). We use a logistic classifier to predict whether movie reviews are positive or negative.

The code for this chapter is available in the following GitHub repository: https://github.com/luisguiserrano/manning/tree/master/Chapter_6_Logistic_Regression.

Logistic classifiers: A continuous version of perceptron classifiers

In chapter 5, we covered the perceptron, which is a type of classifier that uses the features of our data to make a prediction. The prediction can be 1 or 0. This is called a *discrete perceptron*, because it returns an answer from a discrete set (the set containing 0 and 1). In this chapter, we learn *continuous perceptrons*, which return an answer that can be any number in the interval between 0 and 1. A more common name for continuous perceptrons is *logistic classifiers*. The output of a logistic classifier can be interpreted as a score, and the goal of the logistic classifier is to assign scores as close as possible to the label of the points—points with label 0 should get scores close to 0, and points with label 1 should get scores close to 1.

We can visualize continuous perceptrons similar to discrete perceptrons: with a line (or high-dimensional plane) that separates two classes of data. The only difference is that the discrete perceptron predicts that everything to one side of the line has label 1 and to the other side has label 0, whereas the continuous perceptron assigns a value from 0 to 1 to all the points based on their position with respect to the line. Every point on the line gets a value of 0.5. This value means the model can't decide if the sentence is happy or sad. For example, in the ongoing sentiment analysis example, the sentence "Today is Tuesday" is neither happy nor sad, so the model would assign it a score close to 0.5. Points in the positive zone get scores larger than 0.5, where the points even farther away from the 0.5 line in the positive direction get values closer to 1. Points in the negative zone get scores smaller than 0.5, where, again, the points farther from the line get values closer to 0. No point gets a value of 1 or 0 (unless we consider points at infinity), as shown in figure 6.1.

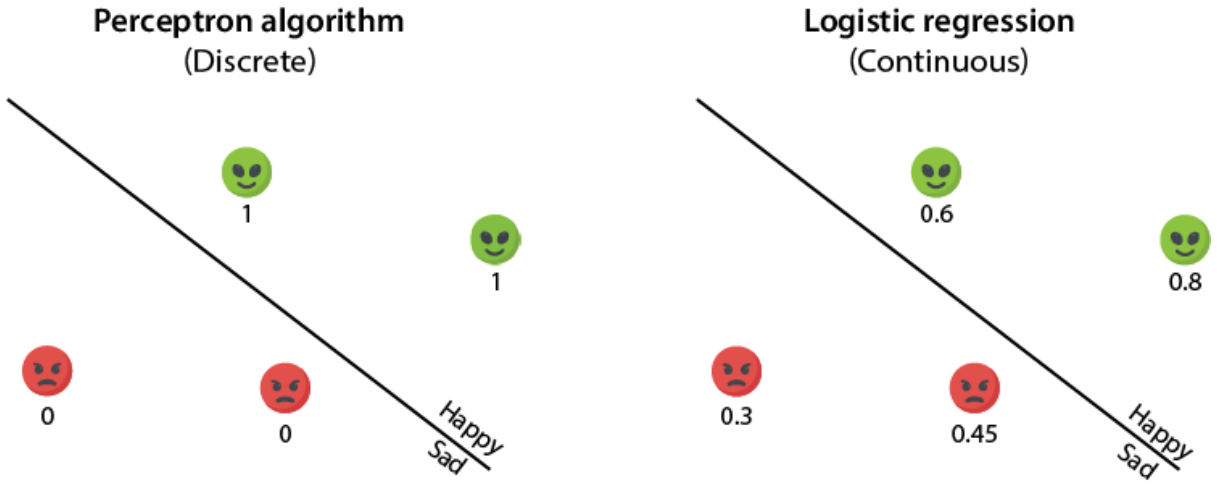


Figure 6.1 Left: The perceptron algorithm trains a discrete perceptron, where the predictions are 0 (happy) and 1 (sad). Right: The logistic regression algorithm trains a continuous perceptron, where the predictions are numbers between 0 and 1 which indicate the predicted level of happiness.

Why do we call this *classification* instead of *regression*, given that the logistic classifier is not outputting a state per se but a number? The reason is, after scoring the points, we can classify them into two classes, namely, those points with a score of 0.5 or higher and those with a score lower than 0.5. Graphically, the two classes are separated by the boundary line, just like with the perceptron classifier. However, the algorithm we use to train logistic classifiers is called the *logistic regression algorithm*. This notation is a bit peculiar, but we'll keep it as it is to match the literature.

A probability approach to classification: The sigmoid function

How do we slightly modify the perceptron models from the previous section to get a score for each sentence, as opposed to a simple "happy" or "sad"? Recall how we made the predictions in the perceptron models. We scored each sentence by separately scoring each word and adding the scores, plus the bias. If the score was positive, we predicted that the sentence was happy, and if it was negative, we predicted that the sentence was sad. In other words, what we did was apply the step function to the score. The step function returns a 1 if the score was nonnegative and a 0 if it was negative.

Now we do something similar. We take a function that receives the score as the input and outputs a number between 0 and 1. The number is close to 1 if the score is positive and close to zero if the score is negative. If the score is zero, then the output is 0.5. Imagine if you could take the entire number line and crunch it into the interval between 0 and 1. It would look like the function in figure 6.2.

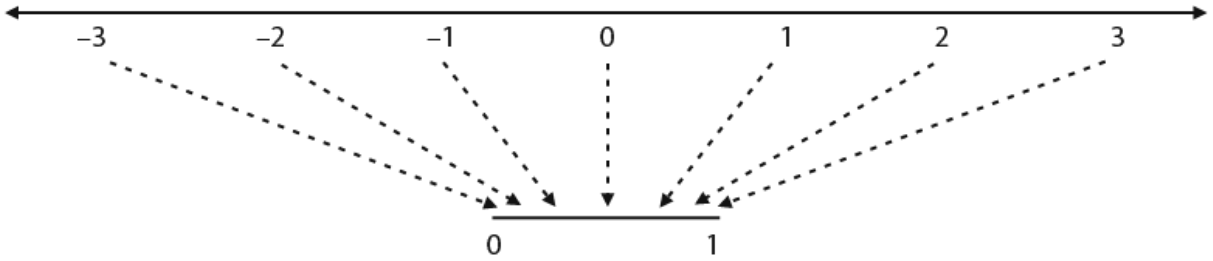


Figure 6.2 The sigmoid function sends the entire number line to the interval (0,1).

Many functions can help us here, and in this case, we use one called the *sigmoid*, denoted with the Greek letter *sigma* (σ). The formula for the sigmoid follows:

$$\sigma(x)=\frac{1}{1+e^{-x}}$$

What really matters here is not the formula but what the function does, which is crunching the real number line into the interval (0,1). In figure 6.3, we can see a comparison of the graphs of the step and the sigmoid functions.

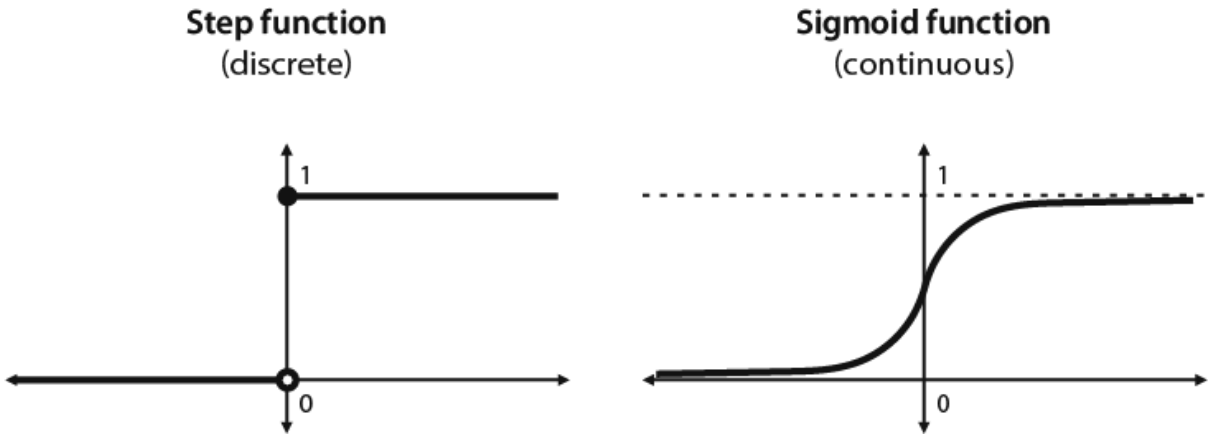


Figure 6.3 Left: The step function used to build discrete perceptrons. It outputs a value of 0 for any negative input and a value of 1 for any input that is positive or zero. It has a discontinuity at zero. Right: The sigmoid function used to build continuous perceptrons. It outputs values less than 0.5 for negative inputs and values greater than 0.5 for positive inputs. At zero, it outputs 0.5. It is continuous and differentiable everywhere.

The sigmoid function is, in general, better than the step function for several reasons. Having continuous predictions gives us more information than discrete predictions. In addition, when we get into the calculus, the sigmoid function has a much nicer derivative than the step

function. The step function has a derivative of zero, with the exception of the origin, where it is undefined. In table 6.1, we calculate some values of the sigmoid function to make sure the function does what we want it to.

Table 6.1 Some inputs and their outputs under the sigmoid function. Notice that for large negative inputs, the output is close to 0, whereas for large positive inputs, the output is close to 1. For the input 0, the output is 0.5.

x	$\sigma(x)$
–5	0.007
–1	0.269
0	0.5
1	0.731
5	0.993

The prediction of a logistic classifier is obtained by applying the sigmoid function to the score, and it returns a number between 0 and 1, which, as was mentioned earlier, can be interpreted in our example as the probability that the sentence is happy.

In chapter 5, we defined an error function for a perceptron, called the perceptron error. We used this perceptron error to iteratively build a perceptron classifier. In this chapter, we follow the same procedure. The error of a continuous perceptron is slightly different from the one of a discrete predictor, but they still have similarities.

The dataset and the predictions

In this chapter, we use the same use case as in chapter 5, in which we have a dataset of sentences in alien language with the labels “happy” and “sad,” denoted by 1 and 0, respectively. The dataset for this chapter is slightly different than that in chapter 5, and it is shown in table 6.2.

Table 6.2 The dataset of sentences with their happy/sad labels. The coordinates are the number of appearances of the words aack and beep in the sentence.

	Words	Coordinates (#aack, #beep)	Label
Sentence 1	Aack beep beep aack aack.	(3,2)	Sad (0)
Sentence 2	Beep aack beep.	(1,2)	Happy (1)
Sentence 3	Beep!	(0,1)	Happy (1)

Sentence 4	Aack aack.	(2,0)	Sad (0)

The model we use has the following weights and bias:

Logistic Classifier 1

- Weight of *Aack*: $a = 1$
- Weight of *Beep*: $b = 2$
- Bias: $c = -4$

We use the same notation as in chapter 5, where the variables x_{aack} and x_{beep} keep track of the appearances of *aack* and *beep*, respectively. A perceptron classifier would predict according to the formula $\hat{y} = step(ax_{aack} + bx_{beep} + c)$, but because this is a logistic classifier, it uses the sigmoid function instead of the step function. Thus, its prediction is $\hat{y} = \sigma(ax_{aack} + bx_{beep} + c)$. In this case, the prediction follows:

Prediction: $\hat{y} = \sigma(1 \cdot x_{aack} + 2 \cdot x_{beep} - 4)$

Therefore, the classifier makes the following predictions on our dataset:

- **Sentence 1**: $\hat{y} = \sigma(3 + 2 \cdot 2 - 4) = \sigma(3) = 0.953$.
- **Sentence 2**: $\hat{y} = \sigma(1 + 2 \cdot 2 - 4) = \sigma(1) = 0.731$.
- **Sentence 3**: $\hat{y} = \sigma(0 + 2 \cdot 1 - 4) = \sigma(-2) = 0.119$.
- **Sentence 4**: $\hat{y} = \sigma(2 + 2 \cdot 0 - 4) = \sigma(-2) = 0.119$.

The boundary between the “happy” and “sad” classes is the line with equation $x_{aack} + 2x_{beep} - 4 = 0$, depicted in figure 6.4.

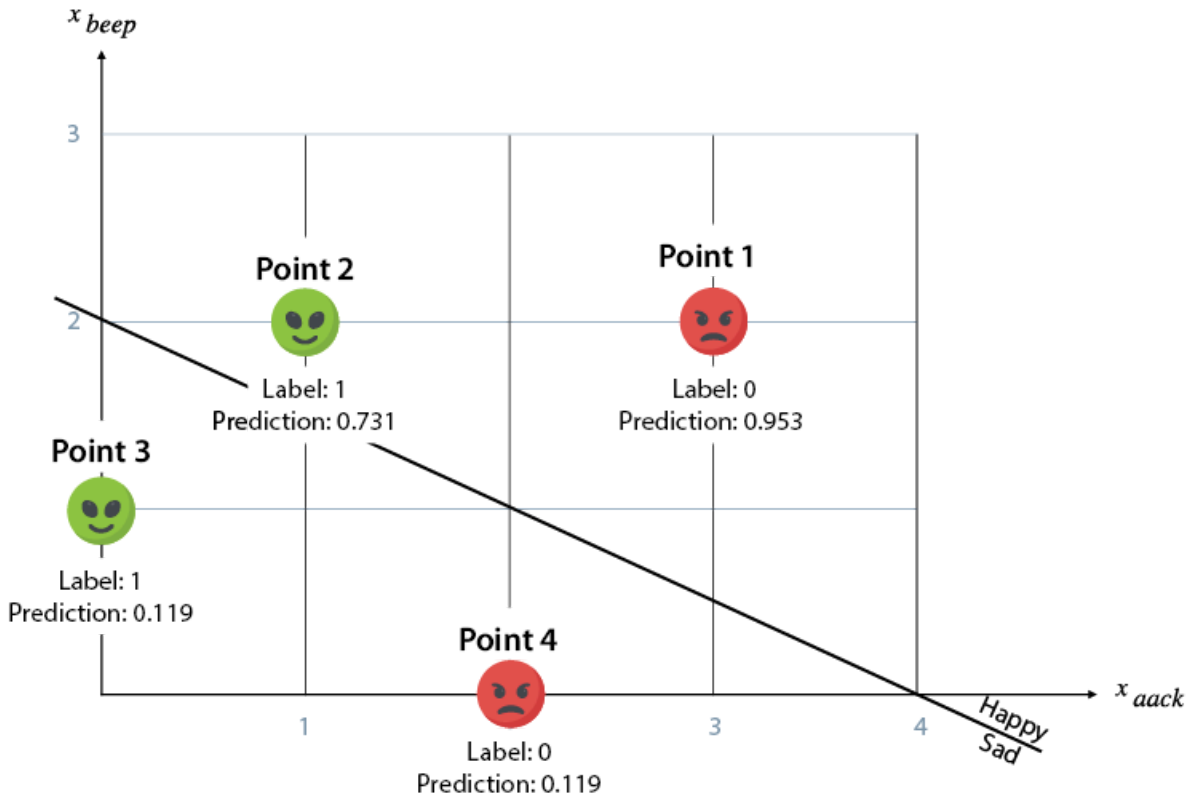


Figure 6.4 The plot of the dataset in table 6.2 with predictions. Notice that points 2 and 4 are correctly classified, but points 1 and 3 are misclassified.

This line splits the plane into positive (happy) and negative (sad) zones. The positive zone is formed by the points with prediction higher than or equal to 0.5, and the negative zone is formed by those with prediction less than 0.5.

The error functions: Absolute, square, and log loss

In this section, we build three error functions for a logistic classifier. What properties would you like a good error function to have? Some examples follow:

- If a point is correctly classified, the error is a small number.
- If a point is incorrectly classified, the error is a large number.
- The error of a classifier for a set of points is the sum (or average) of the errors at all the points.

Many functions satisfy these properties, and we will see three of them; the absolute error, the square error, and the log loss. In table 6.3, we have the labels and predictions for the four points corresponding to the sentences in our dataset with the following characteristics:

- The points on the line are given a prediction of 0.5.
- Points that are in the positive zone are given predictions higher than 0.5, and the farther a point is from the line in that direction, the closer its prediction is to 1.
- Points that are in the negative zone are given predictions lower than 0.5, and the farther a point is from the line in that direction, the closer its prediction is to 0.

Table 6.3 Four points—two happy and two sad with their predictions—as illustrated in figure 6.4. Notice that points 1 and 4 are correctly classified, but points 2 and 3 are not. A good error function should assign small errors to the correctly classified points and large errors to the poorly classified points.

Point	Label	Prediction	Error?
1	0 (Sad)	0.953	Should be large
2	1 (Happy)	0.731	Should be small
3	1 (Happy)	0.119	Should be large
4	0 (Sad)	0.119	Should be small

Notice that in table 6.3, points 2 and 4 get a prediction that is close to the label, so they should have small errors. In contrast, points 1 and 3 get a prediction that is far from the label, so they should have large errors. Three error functions that have this particular property follow:

Error function 1: Absolute error

The absolute error is similar to the absolute error we defined for linear regression in chapter 3. It is the absolute value of the difference between the prediction and the label. As we can see, it is large when the prediction is far from the label and small when they are close.

Error function 2: Square error

Again, just like in linear regression, we also have the square error. This is the square of the difference between the prediction and the label, and it works for the same reason that the absolute error works.

Before we proceed, let's calculate the absolute and square error for the points in table 6.4. Notice that points 2 and 4 (correctly classified) have small errors, and points 1 and 3 (incorrectly classified) have larger errors.

Table 6.4 We have attached the absolute error and the square error for the points in table 6.3. Notice that as we desired, points 2 and 4 have small errors, and points 1 and 3 have larger errors.

Point	Label	Predicted label	Absolute Error	Square Error
1	0 (Sad)	0.953	0.953	0.908
2	1 (Happy)	0.731	0.269	0.072
3	1 (Happy)	0.119	0.881	0.776

4	0 (Sad)	0.119	0.119	0.014

The absolute and the square errors may remind you of the error functions used in regression. However, in classification, they are not so widely used. The most popular is the next one we see. Why is it more popular? The math (derivatives) works much nicer with the next function. Also, these errors are all pretty small. In fact, they are all smaller than 1, no matter how poorly classified the point is. The reason is that the difference (or the square of the difference) between two numbers that are between 0 and 1 is at most 1. To properly train models, we need error functions that take larger values than that. Thankfully, a third error function can do that for us.

Error function 3: log loss

The *log loss* is the most widely used error function for continuous perceptrons. Most of the error functions in this book have the word *error* in their name, but this one instead has the word *loss* in its name. The *log* part in the name comes from a natural logarithm that we use in the formula. However, the real soul of the log loss is probability.

The outputs of a continuous perceptron are numbers between 0 and 1, so they can be considered probabilities. The model assigns a probability to every data point, and that is the probability that the point is happy. From this, we can infer the probability that the point is sad, which is 1 minus the probability of being happy. For example, if the prediction is 0.75, that means the model believes the point is happy with a probability of 0.75 and sad with a probability of 0.25.

Now, here is the main observation. The goal of the model is to assign high probabilities to the happy points (those with label 1) and low probabilities to the sad points (those with label 0). Notice that the probability that a point is sad is 1 minus the probability that the point is happy. Thus, for each point, let's calculate the probability that the model gives to its label. For the points in our dataset, the corresponding probabilities follow:

- **Point 1**:
  - Label = 0 (sad)
  - Prediction (probability of being happy) = 0.953
  - Probability of being its label: 1 – 0.953 = **0.047**
- **Point 2**:
  - Label = 1 (happy)
  - Prediction (probability of being happy) = 0.731
  - Probability of being its label: **0.731**
- **Point 3**:
  - Label = 1 (happy)
  - Prediction (probability of being happy) = 0.119
  - Probability of being its label: **0.119**
- **Point 4**:
  - Label = 0 (sad)
  - Prediction (probability of being happy) = 0.119
  - Probability of being its label: 1 – 0.119 = **0.881**

Notice that points 2 and 4 are the points that are well classified, and the model assigns a high probability that they are their own label. In contrast, points 1 and 3 are poorly classified, and the model assigns a low probability that they are their own label.

The logistic classifier, in contrast with the perceptron classifier, doesn't give definite answers. The perceptron classifier would say, "I am 100% sure that this point is happy," whereas the logistic classifier says, "Your point has a 73% probability of being happy and 27% of being sad." Although the goal of the perceptron classifier is to be correct as many times as possible, the goal of the logistic classifier is to assign to each point the highest possible probability of having the correct label. This classifier assigns the probabilities 0.047, 0.731, 0.119, and 0.881 to the four labels. Ideally, we'd like these numbers to be higher. How do we measure these four numbers? One way would be to add them or average them. But because they are probabilities, the natural approach is to multiply them. When events are independent, the probability of them occurring simultaneously is the product of their probabilities. If we assume that the four predictions are independent, then the probability that this model assigns to the labels "sad, happy, happy, sad" is the product of the four numbers, which is $0.047 \cdot 0.731 \cdot 0.119 \cdot 0.881 = 0.004$. This is a very small probability. Our hope would be that a model that fits this dataset better would result in a higher probability.

That probability we just calculated seems like a good measure for our model, but it has some problems. For instance, it is a product of many small numbers. Products of many small numbers tend to be tiny. Imagine if our dataset had one million points. The probability would be a product of one million numbers, all between 0 and 1. This number may be so small that a computer may not be able to represent it. Also, manipulating a product of one million numbers is extremely difficult. Is there any way that we could perhaps turn it into something easier to manipulate, like a sum?

Luckily for us, we have a convenient way to turn products into sums—using the logarithms. For this entire book, all we need to know about the logarithm is that it turns products into sums. More specifically, the logarithm of a product of two numbers is the sum of the logarithms of the numbers, as shown next:

$$ln(a \cdot b) = ln(a) + ln(b)$$

We can use logarithms in base 2, 10, or e. In this chapter, we use the natural logarithm, which is on base e. However, the same results can be obtained if we were to use the logarithm in any other base.

If we apply the natural logarithm to our product of probabilities, we obtain

$$ln(0.047 \cdot 0.731 \cdot 0.119 \cdot 0.881) = ln(0.047) + ln(0.731) + ln(0.119) + ln(0.881) = -5.616.$$

One small detail. Notice that the result is a negative number. In fact, this will always be the case, because the logarithm of a number between 0 and 1 is always negative. Thus, if we take the negative logarithm of the product of probabilities, it is always a positive number.

The log loss is defined as the negative logarithm of the product of probabilities, which is also the sum of the negative logarithms of the probabilities. Furthermore, each of the summands

is the log loss at that point. In table 6.5, you can see the calculation of the log loss for each of the points. By adding the log losses of all the points, we obtain a total log loss of 5.616.

Table 6.5 Calculation of the log loss for the points in our dataset. Notice that points that are well classified (2 and 4) have a small log loss, whereas points that are poorly classified (1 and 3) have a large log loss.

Point	Label	Predicted label	Probability of being its label	Log loss
1	0 (Sad)	0.953	0.047	–ln(0.047) = 3.049
2	1 (Happy)	0.731	0.731	–ln(0.731) = 0.313
3	1 (Happy)	0.119	0.119	–ln(0.119) = 2.127
4	0 (Sad)	0.119	0.881	–ln(0.881) = 0.127

Notice that, indeed, the well-classified points (2 and 4) have a small log loss, and the poorly classified points have a large log loss. The reason is that if a number x is close to 0, $-ln(x)$ is a large number, but if x is close to 1, then $-ln(x)$ is a small number.

To summarize, the steps for calculating the log loss follow:

- For each point, we calculate the probability that the classifier gives its label.
    - For the happy points, this probability is the score.
    - For the sad points, this probability is 1 minus the score.
- We multiply all these probabilities to obtain the total probability that the classifier has given to these labels.
- We apply the natural logarithm to that total probability.
- The logarithm of a product is the sum of the logarithms of the factors, so we obtain a sum of logarithms, one for each point.
- We notice that all the terms are negative, because the logarithm of a number less than 1 is a negative number. Thus, we multiply everything by –1 to get a sum of positive numbers.
- This sum is our log loss.

The log loss is closely related to the concept of *cross-entropy*, which is a way to measure similarity between two probability distributions. More details about cross-entropy are available in the references in appendix C.

Formula for the log loss

The log loss for a point can be condensed into a nice formula. Recall that the log loss is the negative logarithm of the probability that the point is its label (happy or sad). The prediction

the model gives to each point is $\hat{y}$, and that is the probability that the point is happy. Thus, the probability that the point is sad, according to the model, is $1 - \hat{y}$. Therefore, we can write the log loss as follows:

- If the label is 0: $\textit{log loss} = -\ln(1 - \hat{y})$
- If the label is 1: $\textit{log loss} = -\ln(\hat{y})$

Because the label is y, the previous `if` statement can be condensed into the following formula:

$\textit{log loss} = -y \ln(\hat{y}) - (1 - y) \ln(1 - \hat{y})$

The previous formula works because if the label is 0, the first summand is 0, and if the label is 1, the second summand is 0. We use the term *log loss* when we refer to the log loss of a point or of a whole dataset. The log loss of a dataset is the sum of the log losses at every point.

Comparing classifiers using the log loss

Now that we have settled on an error function for logistic classifiers, the log loss, we can use it to compare two classifiers. Recall that the classifier we've been using in this chapter is defined by the following weights and bias:

Logistic Classifier 1

- Weight of *Aack*: $a = 1$
- Weight of *Beep*: $b = 2$
- Bias: $c = -4$

In this section, we compare it with the following logistic classifier:

Logistic Classifier 2

- Weight of *Aack*: $a = -1$
- Weight of *Beep*: $b = 1$
- Bias: $c = 0$

The predictions that each classifier makes follow:

- **Classifier 1**: $\hat{y} = \sigma(x_{\text{aack}} + 2x_{\text{beep}} - 4)$
- **Classifier 2**: $\hat{y} = \sigma(-x_{\text{aack}} + x_{\text{beep}})$

The predictions of both classifiers are recorded in table 6.6, and the plot of the dataset and the two boundary lines are shown in figure 6.5.

Table 6.6 Calculation of the log loss for the points in our dataset. Notice that the predictions made by classifier 2 are much closer to the labels of the points than the predictions made by classifier 1. Thus, classifier 2 is a better classifier.

Point	Label	Classifier 1 prediction	Classifier 2 prediction
1	0 (Sad)	0.953	0.269
2	1 (Happy)	0.731	0.731
3	1 (Happy)	0.119	0.731
4	0 (Sad)	0.881	0.119

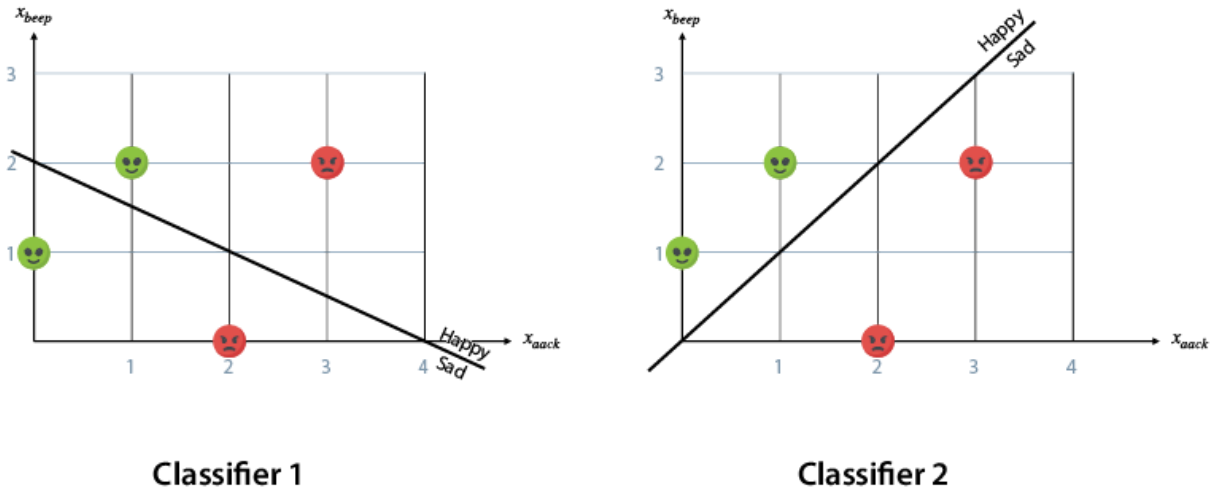


Figure 6.5 Left: A bad classifier that makes two mistakes. Right: A good classifier that classifies all four points correctly.

From the results in table 6.6 and figure 6.5, it is clear that classifier 2 is much better than classifier 1. For instance, in figure 6.5, we can see that classifier 2 correctly located the two happy sentences in the positive zone and the two sad sentences in the negative zone. Next, we compare the log losses. Recall that the log loss for classifier 1 was 5.616. We should obtain a smaller log loss for classifier 2, because this is the better classifier.

According to the formula $log\ loss = -y\ ln(\hat{y}) - (1 - y)\ ln(1 - \hat{y})$, the log loss for classifier 2 at each of the points in our dataset follows:

- **Point 1**: $y = 0$, $\hat{y} = 0.269$:
  - $log\ loss = ln(1 - 0.269) = 0.313$
- **Point 2**: $y = 1$, $\hat{y} = 0.731$:
  - $log\ loss = ln(0.731) = 0.313$
- **Point 3**: $y = 1$, $\hat{y} = 0.731$:
  - $log\ loss = ln(0.731) = 0.313$
- **Point 4**: $y = 0$, $\hat{y} = 0.119$:
  - $log\ loss = ln(1 - 0.119) = 0.127$

The total log loss for the dataset is the sum of these four, which is 1.067. Notice that this is much smaller than 5.616, confirming that classifier 2 is indeed much better than classifier 1.

How to find a good logistic classifier? The logistic regression algorithm

In this section, we learn how to train a logistic classifier. The process is similar to the process of training a linear regression model or a perceptron classifier and consists of the following steps:

- Start with a random logistic classifier.
- Repeat many times:
    - Slightly improve the classifier.
- Measure the log loss to decide when to stop running the loop.

The key to the algorithm is the step inside the loop, which consists of slightly improving a logistic classifier. This step uses a trick called the *logistic trick*. The logistic trick is similar to the perceptron trick, as we see in the next section.

The logistic trick: A way to slightly improve the continuous perceptron

Recall from chapter 5 that the perceptron trick consists of starting with a random classifier, successively picking a random point, and applying the perceptron trick. It had the following two cases:

- **Case 1**: If the point is correctly classified, leave the line as it is.
- **Case 2**: If the point is incorrectly classified, move the line a little closer to the point.

The logistic trick (illustrated in figure 6.6) is similar to the perceptron trick. The only thing that changes is that when the point is well classified, we move the line *away* from the point. It has the following two cases:

- **Case 1**: If the point is correctly classified, slightly move the line away from the point.
- **Case 2**: If the point is incorrectly classified, slightly move the line closer to the point.

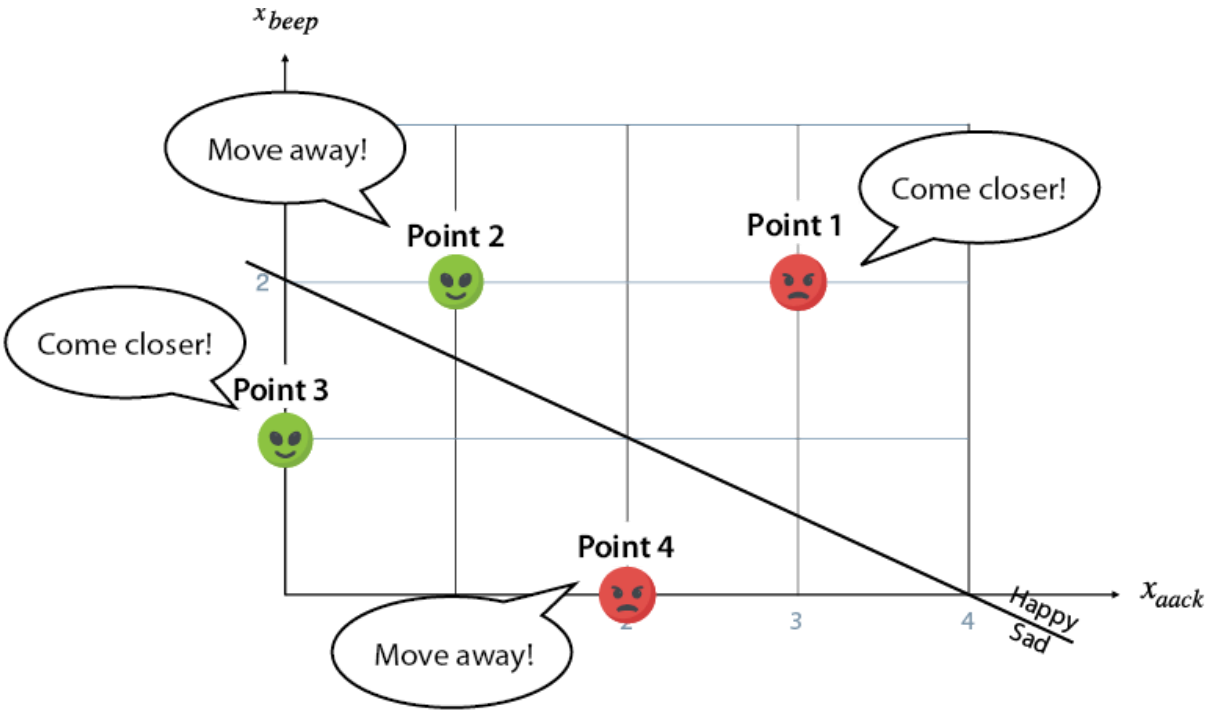


Figure 6.6 In the logistic regression algorithm, every point has a say. Points that are correctly classified tell the line to move farther away, to be deeper in the correct zone. Points that are incorrectly classified tell the line to come closer, in hopes of one day being on the correct side of the line.

Why do we move the line away from a correctly classified point? If the point is well classified, it means it is in the correct zone with respect to the line. If we move the line farther away, we move the point even deeper into the correct zone. Because the prediction is based on how far the point is from the boundary line, for points in the positive (happy) zone, the prediction increases if the point is farther from the line. Similarly, for points in the negative (sad) zone, the prediction decreases if the point is farther from the line. Thus, if the label of the point is 1, we are increasing the prediction (making it even closer to 1), and if the label of the point is 0, we are decreasing the prediction (making it even closer to 0).

For example, look at classifier 1 and the first sentence in our dataset. Recall that the classifier has weights $a = 1$, $b = 2$, and bias $c = -4$. The sentence corresponds to a point of coordinates $(x_{aack}, x_{beep}) = (3,2)$, and label $y = 0$. The prediction we obtained for this point was $\hat{y} = \sigma(3 + 2 \cdot 2 - 4) = \sigma(3) = 0.953$. The prediction is quite far from the label, so the error is high: in fact, in table 6.5, we calculated it to be 3.049. The error that this classifier made was to think that this sentence is happier than it is. Thus, to tune the weights to ensure that the classifier reduces the prediction for this sentence, we should drastically decrease the weights a, b, and the bias c.

Using the same logic, we can analyze how to tune the weights to improve the classification for the other points. For the second sentence in the dataset, the label is $y = 1$ and the prediction is 0.731. This is a good prediction, but if we want to improve it, we should slightly increase the weights and the bias. For the third sentence, because the label is $y = 1$ and the prediction is $\hat{y} = 0.119$, we should drastically increase the weights and the bias. Finally, for

the fourth sentence, the label is $y = 0$ and the prediction is $\hat{y} = 0.119$, so we should slightly decrease the weights and the bias. These are summarized in table 6.7.

Table 6.7 Calculation of the log loss for the points in our dataset. Notice that points that are well classified (2 and 4) have a small log loss, whereas points that are poorly classified (1 and 3) have a large log loss.

Point	Label y	Classifier 1 prediction $\hat{y}$	How to tune the weights a, b, and the bias c	$y - \hat{y}$
1	0	0.953	Decrease by a large amount	–0.953
2	1	0.731	Increase by a small amount	0.269
3	1	0.119	Increase by a large amount	0.881
4	0	0.119	Decrease by a small amount	–0.119

The following observations can help us figure out the perfect amount that we want to add to the weights and bias to improve the predictions:

- **Observation 1**: the last column of table 6.7 has the value of the label minus the prediction. Notice the similarities between the two rightmost columns in this table. This hints that the amount we should update the weights and the bias should be a multiple of $y - \hat{y}$.
- **Observation 2**: imagine a sentence in which the word *aack* appears 10 times and *beep* appears once. If we are to add (or subtract) a value to the weights of these two words, it makes sense to think that the weight of *aack* should be updated by a larger amount, because this word is more crucial to the overall score of the sentence. Thus, the amount we should update the weight of *aack* should be multiplied by x_{aack} , and the amount we should update the weight of *beep* should be multiplied by x_{beep} .
- **Observation 3**: the amount that we update the weights and biases should also be multiplied by the learning rate η because we want to make sure that this number is small.

Putting the three observations together, we conclude that the following is a good set of updated weights:

- $a' = a + \eta(y - \hat{y})x_1$
- $b' = b + \eta(y - \hat{y})x_2$
- $c' = c + \eta(y - \hat{y})$

Thus, the pseudocode for the logistic trick follows. Notice how similar it is to the pseudocode for the perceptron trick we learned at the end of the section “The perceptron trick” in chapter 5.

Pseudocode for the logistic trick

Inputs:

- A logistic classifier with weights *a, b,* and bias *c*
- A point with coordinates (x_1, x_2) and label *y*
- A small value η (the learning rate)

Output:

- A perceptron with new weights *a′, b′,* and bias *c′* which is at least as good as the input

Procedure:

- The prediction the perceptron makes at the point is $\hat{y} = \sigma(ax_1 + bx_2 + c)$.

Return:

- The perceptron with the following weights and bias:
  - $a' = a + \eta(y - \hat{y})x_1$
  - $b' = b + \eta(y - \hat{y})x_2$
  - $c' = c + \eta(y - \hat{y})$

The way we updated the weights and bias in the logistic trick is no coincidence. It comes from applying the gradient descent algorithm to reduce the log loss. The mathematical details are described in appendix B, section “Using gradient descent to train classification models.”

To verify that the logistic trick works in our case, let’s apply it to the current dataset. In fact, we’ll apply the trick to each of the four points separately, to see how much each one of them would modify the weights and bias of the model. Finally, we’ll compare the log loss at that point before and after the update and verify that it has indeed been reduced. For the following calculations, we use a learning rate of $\eta = 0.05$.

Updating the classifier using each of the sentences

Using the first sentence:

- Initial weights and bias: $a = 1$, $b = 2$, $c = –4$
- Label: $y = 0$
- Prediction: 0.953
- Initial log loss: $–0 \cdot ln(0.953) – 1\ ln(1 – 0.953) = 3.049$
- Coordinates of the point: $x_{aack} = 3$, $x_{beep} = 2$
- Learning rate: $\eta = 0.01$
- Updated weights and bias:
  - $a' = 1 + 0.05 \cdot (0 – 0.953) \cdot 3 = 0.857$
  - $b' = 2 + 0.05 \cdot (0 – 0.953) \cdot 2 = 1.905$
  - $c' = –4 + 0.05 \cdot (0 – 0.953) = –4.048$

- Updated prediction: $\hat{y} = \sigma(0.857 \cdot 3 + 1.905 \cdot 2 - 4.048 = 0.912$. (Notice that the prediction decreased, so it is now closer to the label 0).
- Final log loss: $-0 \cdot ln(0.912) - 1\ ln(1 - 0.912) = 2.426$. (Note that the error decreased from 3.049 to 2.426).

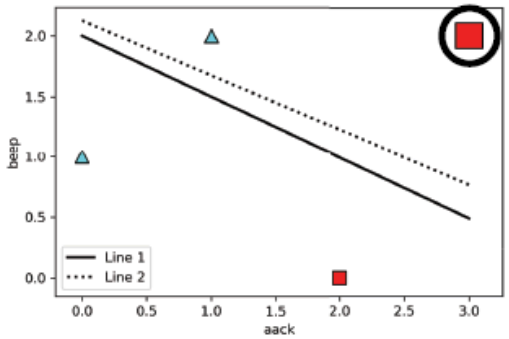
The calculations for the other three points are shown in table 6.8. Notice that in the table, the updated prediction is always closer to the label than the initial prediction, and the final log loss is always smaller than the initial log loss. This means that no matter which point we use for the logistic trick, we'll be improving the model for that point and decreasing the final log loss.

Table 6.8 Calculations of the predictions, log loss, updated weights, and updated predictions for all the points.

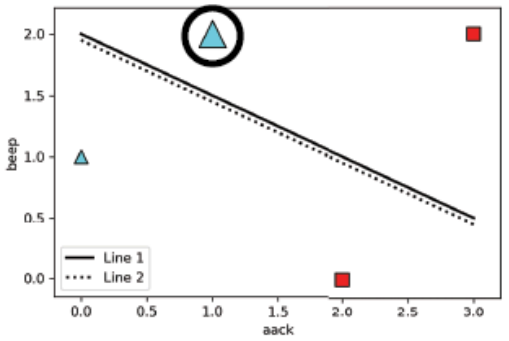
Point	Coordinates	Label	Initial prediction	Initial log loss	Updated weights:	Updated prediction	Final log loss
1	(3,2)	0	0.953	3.049	$a' = 0.857$ $b' = 1.905$ $c' = -4.048$	0.912	2.426
2	(1,2)	1	0.731	0.313	$a' = 1.013$ $b' = 2.027$ $c' = -3.987$	0.747	0.292
3	(0,1)	1	0.119	2.127	$a' = 1$ $b' = 2.044$ $c' = -3.956$	0.129	2.050

4	(2,0)	0	0.119	0.127	a' = 0.988 b' = 2 c' = –4.006	0.127	0.123

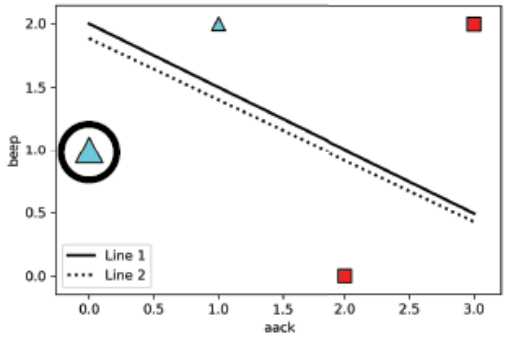
At the beginning of this section, we discussed that the logistic trick can also be visualized geometrically as moving the boundary line with respect to the point. More specifically, the line is moved closer to the point if the point is misclassified and farther from the point if the point is correctly classified. We can verify this by plotting the original classifier and the modified classifier in the four cases in table 6.8. In figure 6.7, you can see the four plots. In each of them, the solid line is the original classifier, and the dotted line is the classifier obtained by applying the logistic trick, using the highlighted point. Notice that points 2 and 4, which are correctly classified, push the line away, whereas points 1 and 3, which are misclassified, move the line closer to them.



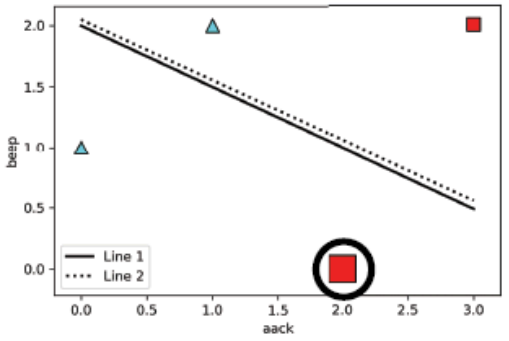
Point: (3,2) (misclassified)
Line 1: $1x_{aack} + 2x_{beep} - 4 = 0$
Line 2: $0.857x_{aack} + 1.905x_{beep} - 4.048 = 0$



Point: (1,2) (correctly classified)
Line 1: $1x_{aack} + 2x_{beep} - 4 = 0$
Line 2: $1.013x_{aack} + 2.027x_{beep} - 3.987 = 0$



Point: (0,1) (misclassified)
Line 1: $1x_{aack} + 2x_{beep} - 4 = 0$
Line 2: $1x_{aack} + 2.044x_{beep} - 3.956 = 0$



Point: (2,0) (correctly classified)
Line 1: $1x_{aack} + 2x_{beep} - 4 = 0$
Line 2: $0.988x_{aack} + 2x_{beep} - 4.006 = 0$

Figure 6.7 The logistic trick applied to each of the four data points. Notice that for correctly classified points, the line moves away from the point, whereas for misclassified points, the line moves closer to the point.

Repeating the logistic trick many times: The logistic regression algorithm

The logistic regression algorithm is what we use to train a logistic classifier. In the same way that the perceptron algorithm consists of repeating the perceptron trick many times, the logistic regression algorithm consists of repeating the logistic trick many times. The pseudocode follows:

Pseudocode for the logistic regression algorithm

Inputs:

- A dataset of points, labeled 1 and 0
- A number of epochs, n
- A learning rate η

Output:

- A logistic classifier consisting of a set of weights and a bias, which fits the dataset

Procedure:

- Start with random values for the weights and bias of the logistic classifier.
- Repeat many times:
    - Pick a random data point.
    - Update the weights and the bias using the logistic trick.

Return:

- The perceptron classifier with the updated weights and bias

As we saw previously, each iteration of the logistic trick either moves the line closer to a misclassified point or farther away from a correctly classified point.

Stochastic, mini-batch, and batch gradient descent

The logistic regression algorithm, together with linear regression and the perceptron, is another algorithm that is based on gradient descent. If we use gradient descent to reduce the log loss, the gradient descent step becomes the logistic trick.

The general logistic regression algorithm works not only for datasets with two features but for datasets with as many features as we want. In this case, just like the perceptron algorithm, the boundary won't look like a line, but it would look like a higher-dimensional hyperplane splitting points in a higher dimensional space. However, we don't need to visualize this higher-dimensional space; we only need to build a logistic regression classifier with as many weights as features in our data. The logistic trick and the logistic algorithm update the weights in a similar way to what we did in the previous sections.

Just like with the previous algorithms we learned, in practice, we don't update the model by picking one point at a time. Instead, we use mini-batch gradient descent—we take a batch of points and update the model to fit those points better. For the fully general logistic regression algorithm and a thorough mathematical derivation of the logistic trick using gradient descent, please refer to appendix B, section "Using gradient descent to train classification models."

Coding the logistic regression algorithm

In this section, we see how to code the logistic regression algorithm by hand. The code for this section follows:

- **Notebook**: Coding_logistic_regression.ipynb
    - https://github.com/luisguiserrano/manning/blob/master/Chapter_6_Logistic_Regression/Coding_logistic_regression.ipynb

We'll test our code in the same dataset that we used in chapter 5. The dataset is shown in table 6.9.

Table 6.9 The dataset that we will fit with a logistic classifier

Aack x_1	*Beep* x_2	Label y
1	0	0
0	2	0
1	1	0
1	2	0
1	3	1
2	2	1
2	3	1
3	2	1

The code for loading our small dataset follows, and the plot of the dataset is shown in figure 6.8:

```
import numpy as np
features = np.array([[1,0],[0,2],[1,1],[1,2],[1,3],[2,2],[2,3],[3,2]])
labels = np.array([0,0,0,0,1,1,1,1])
```

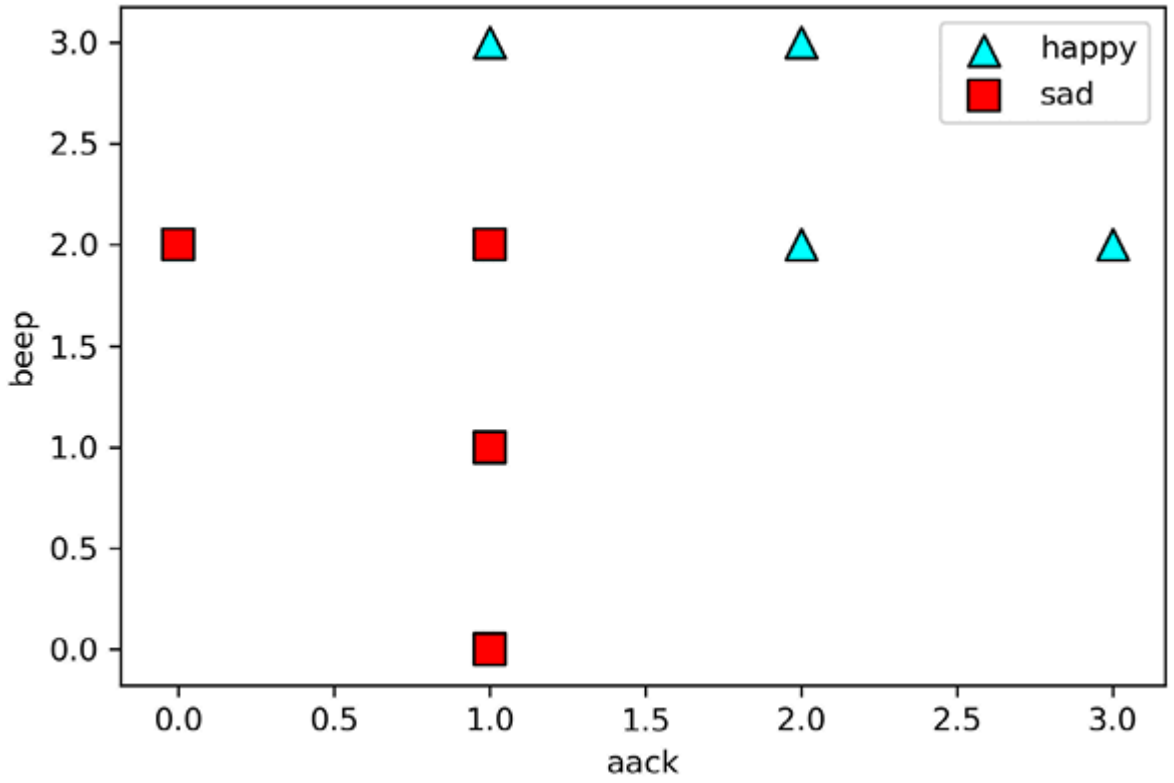


Figure 6.8 The plot of our dataset, where the happy sentences are represented by triangles and the sad sentences by squares.

Coding the logistic regression algorithm by hand

In this section, we see how to code the logistic trick and the logistic regression algorithm by hand. More generally, we'll code the logistic regression algorithm for a dataset with *n* weights. The notation we use follows:

- Features: $x_1, x_2, \ldots, x_n$
- Label: y
- Weights: $w_1, w_2, \ldots, w_n$
- Bias: b

The score for a particular sentence is the sigmoid of the sum of the weight of each word (w_i) times the number of times that appears (x_i), plus the bias (b). Notice that we use the summation notation for

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + \cdots + a_n .$$

- Prediction: $\hat{y} = \sigma(w_1x_1 + w_2x_2 + \ldots + w_nx_n + b) = \sigma(\Sigma_{i=1}^{n} w_i x_i + b)$.

For our current problem, we'll refer to x_{aack} and x_{beep} as x_1 and x_2, respectively. Their corresponding weights are w_1 and w_1, and the bias is b.

We start by coding the sigmoid function, the score, and the prediction. Recall that the formula for the sigmoid function is

$$\sigma(x)=\frac{1}{1+e^{-x}}$$

```
def sigmoid(x):
    return np.exp(x)/(1+np.exp(x))
```

For the score function, we use the dot product between the features and the weights. Recall that the dot product between vectors $(x_1, x_2, \dots, x_n)$ and $(w_1, w_2, \dots, w_n)$ is $w_1 x_1 + w_2 x_2 + \dots + w_n x_n$.

```
def score(weights, bias, features):
    return np.dot(weights, features) + bias
```

Finally, recall that the prediction is the sigmoid activation function applied to the score.

```
def prediction(weights, bias, features):
    return sigmoid(score(weights, bias, features))
```

Now that we have the prediction, we can proceed to the log loss. Recall that the formula for the log loss is

$log\ loss = -y\ ln(\hat{y}) - (1 - y)\ ln(1 - y)$.

Let's code that formula as follows:

```
def log_loss(weights, bias, features, label):
    pred = prediction(weights, bias, features)
    return -label*np.log(pred) - (1-label)*np.log(1-pred)
```

We need the log loss over the whole dataset, so we can add over all the data points as shown here:

```
def total_log_loss(weights, bias, features, labels):
    total_error = 0
    for i in range(len(features)):
        total_error += log_loss(weights, bias, features[i], labels[i])
    return total_error
```

Now we are ready to code the logistic regression trick, and the logistic regression algorithm. In more than two variables, recall that the logistic regression step for the *i*-th weight is the following formula, where η is the learning rate:

- $w_i \to w_i + \eta(y - \hat{y})x_i$ for $i = 1, 2, \dots, n$
- $b \to b + \eta(y - \hat{y})$ for $i = 1, 2, \dots, n$.

```
def logistic_trick(weights, bias, features, label, learning_rate = 0.01):
```

```
    pred = prediction(weights, bias, features)
    for i in range(len(weights)):
        weights[i] += (label-pred)*features[i]*learning_rate
        bias += (label-pred)*learning_rate
    return weights, bias

def logistic_regression_algorithm(features, labels, learning_rate = 0.01, epochs = 1000):
    utils.plot_points(features, labels)
    weights = [1.0 for i in range(len(features[0]))]
    bias = 0.0
    errors = []
    for i in range(epochs):
        errors.append(total_log_loss(weights, bias, features, labels))
        j = random.randint(0, len(features)-1)
        weights, bias = logistic_trick(weights, bias, features[j], labels[j])
    return weights, bias
```

Now we can run the logistic regression algorithm to build a logistic classifier that fits our dataset as follows:

```
logistic_regression_algorithm(features, labels)
([0.46999999999999953, 0.09999999999999937], -0.6800000000000004)
```

The classifier we obtain has the following weights and biases:

- $w_1 = 0.47$
- $w_2 = 0.10$
- $b = -0.68$

The plot of the classifier (together with a plot of the previous classifiers at each of the epochs) is depicted in figure 6.9.

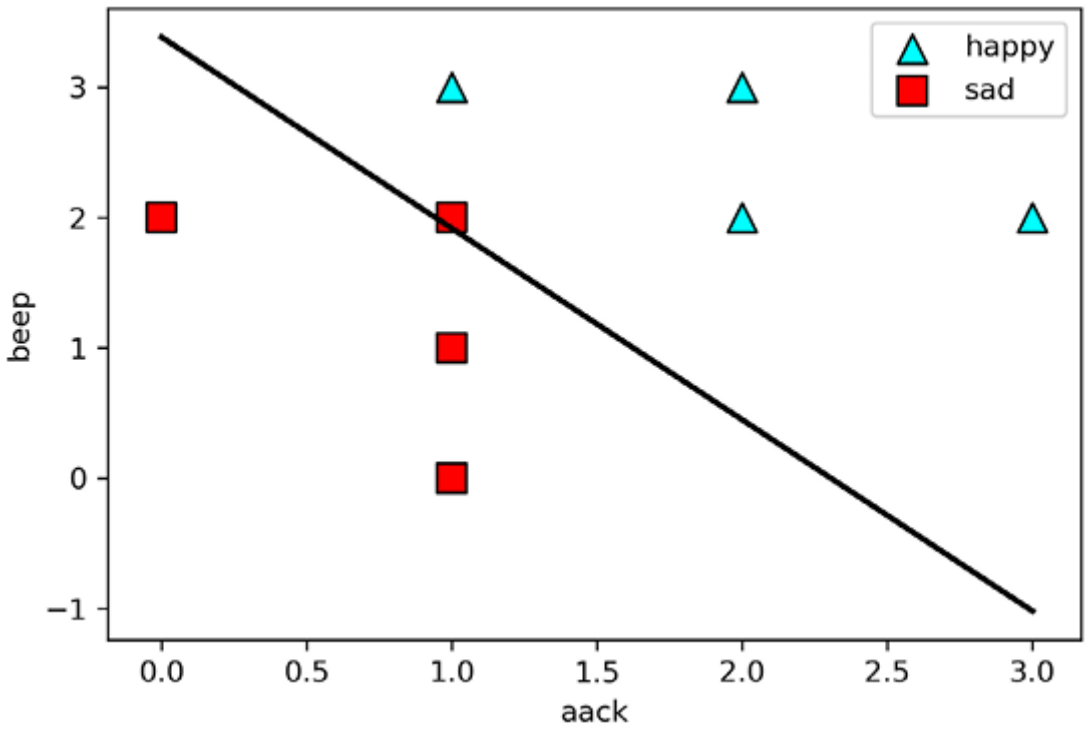


Figure 6.9 The boundary of the resulting logistic classifier

In figure 6.10, we can see the plot of the classifiers corresponding to all the epochs (left) and the plot of the log loss (right). On the plot of the intermediate classifiers, the final one corresponds to the dark line. Notice from the log loss plot that, as we run the algorithm for more epochs, the log loss decreases drastically, which is exactly what we want. Furthermore, the log loss is never zero, even though all the points are correctly classified. This is because for any point, no matter how well classified, the log loss is never zero. Contrast this to figure 5.26 in chapter 5, where the perceptron loss indeed reaches a value of zero when every point is correctly classified.

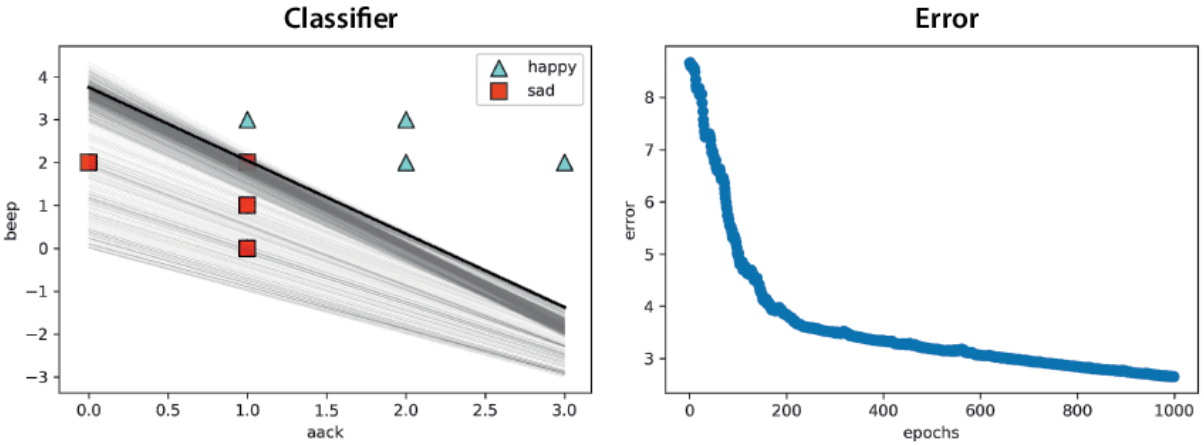


Figure 6.10 Left: A plot of all the intermediate steps of the logistic regression algorithm. Notice that we start with a bad classifier and slowly move toward a good one (the thick line). Right: The error plot. Notice that the more epochs we run the logistic regression algorithm, the lower the error gets.

Real-life application: Classifying IMDB reviews with Turi Create

In this section, we see a real-life application of the logistic classifier in sentiment analysis. We use Turi Create to build a model that analyzes movie reviews on the popular IMDB site. The code for this section follows:

- **Notebook**: Sentiment_analysis_IMDB.ipynb
    - https://github.com/luisguiserrano/manning/blob/master/Chapter_6_Logistic_Regression/Sentiment_analysis_IMDB.ipynb
- **Dataset**: IMDB_Dataset.csv

First, we import Turi Create, download the dataset, and convert it into an SFrame, which we call `movies`, as follows:

```
import turicreate as tc
movies = tc.SFrame('IMDB Dataset.csv')
```

The first five rows of the dataset appear in table 6.10.

Table 6.10 The first five rows of the IMDB dataset. The Review column has the text of the review, and the Sentiment column has the sentiment of the review, which can be positive or negative.

Review	Sentiment
One of the other reviewers has mentioned...	Positive
A wonderful little production...	Positive
I thought this was a wonderful day to spend...	Positive
Basically, there's a family where a little...	Negative
Petter Mattei's "Love in the time of money" is a...	Positive

The dataset has two columns, one with the review, as a string, and one with the sentiment, as positive or negative. First, we need to process the string, because each of the words needs to be a different feature. The Turi Create built-in function `count_words` in the `text_analytics` package is useful for this task, because it turns a sentence into a dictionary with the word counts. For example, the sentence "to be or not to be" is turned into the dictionary {'to':2, 'be':2, 'or':1, 'not':1}. We add a new column called `words` containing this dictionary as follows:

```
movies['words'] = tc.text_analytics.count_words(movies['review'])
```

The first few rows of our dataset with the new column are shown in table 6.11.

Table 6.11 The Words column is a dictionary where each word in the review is recorded together with its number of appearances. This is the column of features for our logistic classifier.

Review	Sentiment	Words
One of the other reviewers has mentioned...	Positive	{'if': 1.0, 'viewing': 1.0, 'comfortable': 1.0, ...
A wonderful little production...	Positive	{'done': 1.0, 'every': 1.0, 'decorating': 1.0, ...
I thought this was a wonderful day to spend...	Positive	{'see': 1.0, 'go': 1.0, 'great': 1.0, 'superm ...
Basically, there's a family where a little...	Negative	{'them': 1.0, 'ignore': 1.0, 'dialogs': 1.0, ...
Peter Mattei's *Love in the Time of Money* is a...	Positive	{'work': 1.0, 'his': 1.0, 'for': 1.0, 'anxiously': ...

We are ready to train our model! For this, we use the function `create` in the `logistic_classifier` package, in which we specify the target (label) to be the `sentiment` column and the features to be the `words` column. Note that the target is expressed as a string with the name of the column containing the label, but the features are expressed as an array of strings with the names of the columns containing each of the features (in case we need to specify several columns), as shown here:

```
model = tc.logistic_classifier.create(movies, features=['words'], target='sentiment')
```

Now that we've trained our model, we can look at the weights of the words, with the `coefficients` command. The table we obtain has several columns, but the ones we care about are `index` and `value`, which show the words and their weights. The top five follow:

- (intercept): 0.065
- if: –0.018
- viewing: 0.089
- comfortable: 0.517
- become: 0.106

The first one, called intercept, is the bias. Because the bias of the model is positive, the empty review is positive, as we learned in chapter 5, in the section "The bias, the *y*-intercept, and the inherent mood of a quiet alien." This makes sense, because users who rate movies negatively tend to leave a review, whereas many users who rate movies positively don't leave any review. The other words are neutral, so their weights don't mean very much, but let's explore the weights of some words, such as *wonderful*, *horrible*, and *the*, as shown next:

- wonderful: 1.043
- horrible: –1.075
- the: 0.0005

As we see, the weight of the word *wonderful* is positive, the weight of the word *horrible* is negative, and the weight of the word *the* is small. This makes sense: *wonderful* is a positive word, *horrible* is a negative word, and *the* is a neutral word.

As a last step, let's find the most positive and negative reviews. For this, we use the model to make predictions for all the movies. These predictions will be stored in a new column called `predictions`, using the following command:

```
movies['prediction'] = model.predict(movies, output_type='probability')
```

Let's find the most positive and most negative movies, according to the model. We do this by sorting the array, as follows:

Most positive review:

```
movies.sort('predictions')[-1]
```

Output: "It seems to me that a lot of people don't know that *Blade* is actually a superhero movie on par with *X-Men*…"

Most negative review:

```
movies.sort('predictions')[0]
```

Output: "Even duller, if possible, than the original…"

We could do a lot more to improve this model. For example, some text manipulation techniques, such as removing punctuation and capitalization, or removing stop words (such as *the*, *and*, *of*, *it*), tend to give us better results. But it's great to see that with a few lines of code, we can build our own sentiment analysis classifier!

Classifying into multiple classes: The softmax function

So far we have seen continuous perceptrons classify two classes, happy and sad. But what if we have more classes? At the end of chapter 5, we discussed that classifying between more than two classes is hard for a discrete perceptron. However, this is easy to do with a logistic classifier.

Imagine an image dataset with three labels: "dog", "cat", and "bird". The way to build a classifier that predicts one of these three labels for every image is to build three classifiers, one for each one of the labels. When a new image comes in, we evaluate it with each of the three classifiers. The classifier corresponding to each animal returns a probability that the image is the corresponding animal. We then classify the image as the animal from the classifier that returned the highest probability.

This, however, is not the ideal way to do it, because this classifier returns a discrete answer, such as “dog,” “cat,” or “bird.” What if we wanted a classifier that returns probabilities for the three animals? Say, an answer could be of the form “10% dog, 85% cat, and 5% bird.” The way we do this is using the softmax function.

The softmax function works as follows: recall that a logistic classifier makes a prediction using a two-step process—first it calculates a score, and then it applies the sigmoid function to this score. Let’s forget about the sigmoid function and output the score instead. Now imagine that the three classifiers returned the following scores:

- Dog classifier: 3
- Cat classifier: 2
- Bird classifier: –1

How do we turn these scores into probabilities? Well, here’s an idea: we can normalize. This means dividing all these numbers by their sum, which is five, to get them to add to one. When we do this, we get the probabilities 3/5 for dog, 2/5 for cat, and –1/5 for bird. This works, but it’s not ideal, because the probability of the image being a bird is a negative number. Probabilities must always be positive, so we need to try something different.

What we need is a function that is always positive and that is also increasing. Exponential functions work great for this. Any exponential function, such as 2^x, 3^x, or 10^x, would do the job. By default, we use the function e^x, which has wonderful mathematical properties (e.g., the derivative of e^x is also e^x). We apply this function to the scores, to get the following values:

- Dog classifier: $e^3 = 20.085$
- Cat classifier: $e^2 = 7.389$
- Bird classifier: $e^{-1} = 0.368$

Now, we do what we did before—we normalize, or divide by the sum of these numbers for them to add to one. The sum is 20.085 + 7.389 + 0.368 = 27.842, so we get the following:

- Probability of dog: 20.085/27.842 = 0.721
- Probability of cat: 7.389/27.842 = 0.265
- Probability of bird: 0.368/27.842 = 0.013

These are the three probabilities given by our three classifiers. The function we used was the softmax, and the general version follows: if we have n classifiers that output the n scores $a_1, a_2, \dots, a_n$, the probabilities obtained are $p_1, p_2, \dots, p_n$, where

$$p_i = \frac{e^{a_i}}{e^{a_1} + e^{a_2} + \cdots + e^{a_n}}.$$

This formula is known as the softmax function.

What would happen if we use the softmax function for only two classes? We obtain the sigmoid function. Why not convince yourself of this as an exercise?

Summary

- Continuous perceptrons, or logistic classifiers, are similar to perceptron classifiers, except instead of making a discrete prediction such as 0 or 1, they predict any number between 0 and 1.
- Logistic classifiers are more useful than discrete perceptrons, because they give us more information. Aside from telling us which class the classifier predicts, they also give us a probability. A good logistic classifier would assign low probabilities to points with label 0 and high probabilities to points with label 1.
- The log loss is an error function for logistic classifiers. It is calculated separately for every point as the negative of the natural logarithm of the probability that the classifier assigns to its label.
- The total log loss of a classifier on a dataset is the sum of the log loss at every point.
- The logistic trick takes a labeled data point and a boundary line. If the point is incorrectly classified, the line is moved closer to the point, and if it is correctly classified, the line is moved farther from the point. This is more useful than the perceptron trick, because the perceptron trick doesn't move the line if the point is correctly classified.
- The logistic regression algorithm is used to fit a logistic classifier to a labeled dataset. It consists of starting with a logistic classifier with random weights and continuously picking a random point and applying the logistic trick to obtain a slightly better classifier.
- When we have several classes to predict, we can build several linear classifiers and combine them using the softmax function.

Exercises

Exercise 6.1

A dentist has trained a logistic classifier on a dataset of patients to predict if they have a decayed tooth. The model has determined that the probability that a patient has a decayed tooth is

$\sigma(d + 0.5c - 0.8)$,

where

- d is a variable that indicates whether the patient has had another decayed tooth in the past, and
- c is a variable that indicates whether the patient eats candy.

For example, if a patient eats candy, then $c = 1$, and if they don't, then $c = 0$. What is the probability that a patient that eats candy and was treated for a decayed tooth last year has a decayed tooth today?

Exercise 6.2

Consider the logistic classifier that assigns to the point (x_1, x_2) the prediction $\hat{y} = \sigma(2x_1 + 3x_2 - 4)$, and the point $p = (1, 1)$ with label 0.

1. Calculate the prediction $\hat{y}$ that the model gives to the point p.
2. Calculate the log loss that the model produces at the point p.
3. Use the logistic trick to obtain a new model that produces a smaller log loss. You can use $\eta = 0.1$ as the learning rate.
4. Find the prediction given by the new model at the point p, and verify that the log loss obtained is smaller than the original.

Exercise 6.3

Using the model in the statement of exercise 6.2, find a point for which the prediction is 0.8.

hint First find the score that will give a prediction of 0.8, and recall that the prediction is $\hat{y} = \sigma(\text{score})$.

Cheat Sheet: Linear and Logistic Regression

Comparing different regression types

Model Name	Description	Code Syntax

Simple linear regression	**Purpose:** To predict a dependent variable based on one independent variable. **Pros:** Easy to implement, interpret, and efficient for small datasets. **Cons:** Not suitable for complex relationships; prone to underfitting. **Modeling equation:** $y = b_0 + b_1x$	1. 1 2. 2 3. 3 1. from sklearn.linear_model import LinearRegression 2. model = LinearRegression() 3. model.fit(X, y)
Polynomial regression	**Purpose:** To capture nonlinear relationships between variables. **Pros:** Better at fitting nonlinear data compared to linear regression. **Cons:** Prone to overfitting with high-degree polynomials. **Modeling equation:** $y = b_0 + b_1x + b_2x^2 + ...$	1. 1 2. 2 3. 3 4. 4 5. 5 1. from sklearn.preprocessing import PolynomialFeatures 2. from sklearn.linear_model import LinearRegression 3. poly = PolynomialFeatures(degree=2) 4. X_poly = poly.fit_transform(X) 5. model = LinearRegression().fit(X_poly, y)

Multiple linear regression	**Purpose:** To predict a dependent variable based on multiple independent variables. **Pros:** Accounts for multiple factors influencing the outcome. **Cons:** Assumes a linear relationship between predictors and target. **Modeling equation:** $y = b_0 + b_1x_1 + b_2x_2 + ...$	1. 1 2. 2 3. 3 1. from sklearn.linear_model import LinearRegression 2. model = LinearRegression() 3. model.fit(X, y)

Logistic regression	**Purpose:** To predict probabilities of categorical outcomes. **Pros:** Efficient for binary classification problems. **Cons:** Assumes a linear relationship between independent variables and log-odds. **Modeling equation:** $\log(p/(1-p)) = b_0 + b_1x_1 + ...$	1. 1 2. 2 3. 3 1. from sklearn.linear_model import LogisticRegression 2. model = LogisticRegression() 3. model.fit(X, y)

Associated functions commonly used

Function/Method Name	Brief Description	Code Syntax
train_test_split	Splits the dataset into training and testing subsets to evaluate the model's performance.	1. 1 2. 2 1. from sklearn.model_selection import train_test_split 2. X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

StandardScaler	Standardizes features by removing the mean and scaling to unit variance.	1. 1 2. 2 3. 3 1. from sklearn.preprocessing import StandardScaler 2. scaler = StandardScaler() 3. X_scaled = scaler.fit_transform(X)
log_loss	Calculates the logarithmic loss, a performance metric for classification models.	1. 1 2. 2 1. from sklearn.metrics import log_loss 2. loss = log_loss(y_true, y_pred_proba)
mean_absolute_error	Calculates the mean absolute error between actual and predicted values.	1. 1 2. 2 1. from sklearn.metrics import mean_absolute_error 2. mae = mean_absolute_error(y_true, y_pred)

mean_squared_error	Computes the mean squared error between actual and predicted values.	1. 1 2. 2 1. from sklearn.metrics import mean_squared_error 2. mse = mean_squared_error(y_true, y_pred)
root_mean_squared_error	Calculates the root mean squared error (RMSE), a commonly used metric for regression tasks.	1. 1 2. 2 3. 3 1. from sklearn.metrics import mean_squared_error 2. import numpy as np 3. rmse = np.sqrt(mean_squared_error(y_true, y_pred))
r2_score	Computes the R-squared value, indicating how well the model explains the variability of the target variable.	1. 1 2. 2 1. from sklearn.metrics import r2_score 2. r2 = r2_score(y_true, y_pred)

Classification: k-NN

What is k-NN?

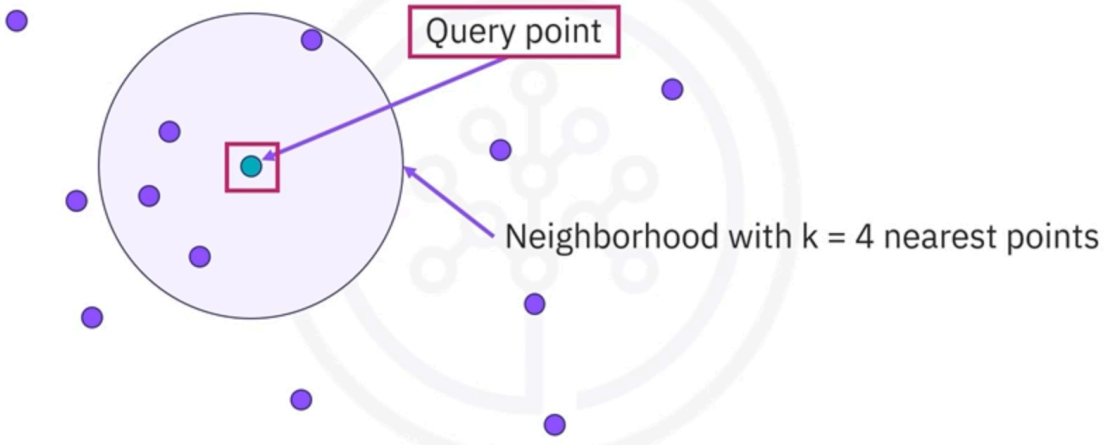
Supervised ML algorithm

Uses labeled points to learn how to label other points

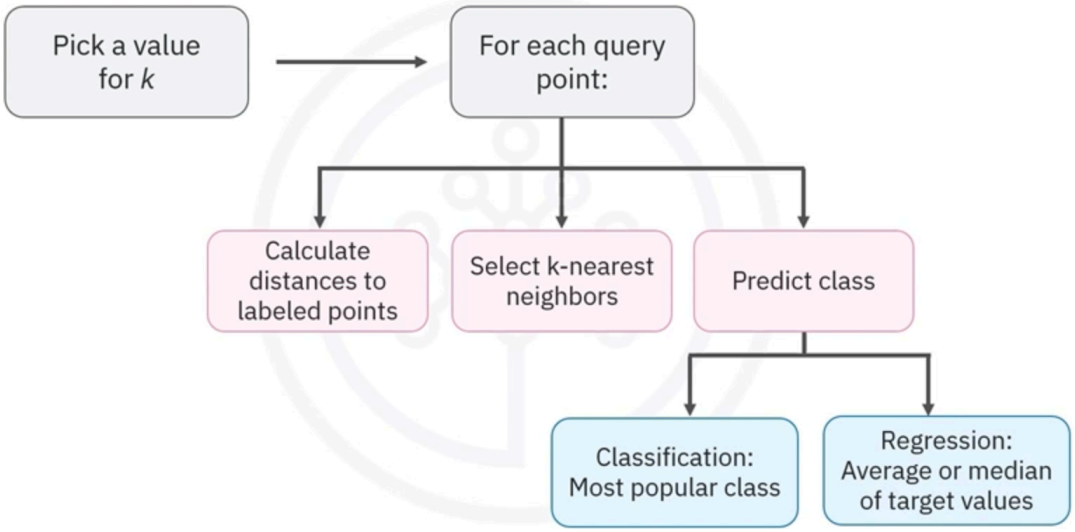
Used for classification and regression

Neighbors: Data points near each other with similar features

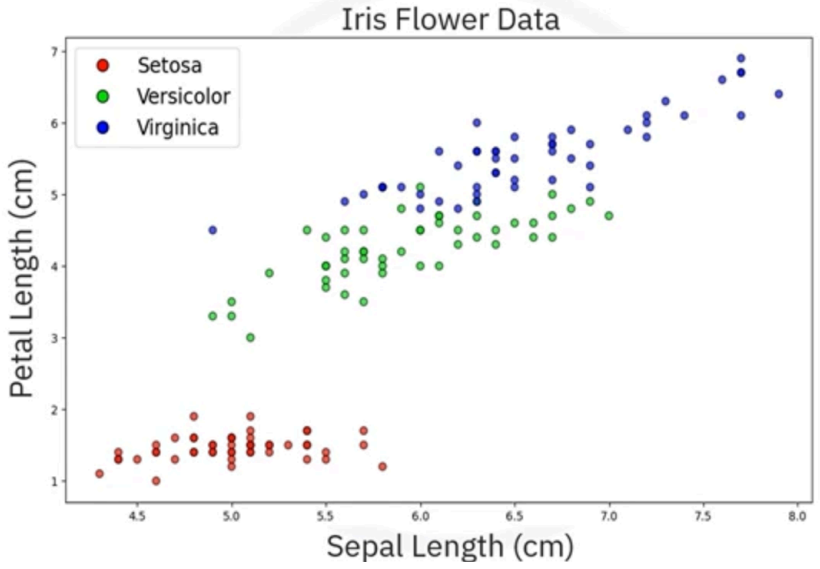
What is k-NN?



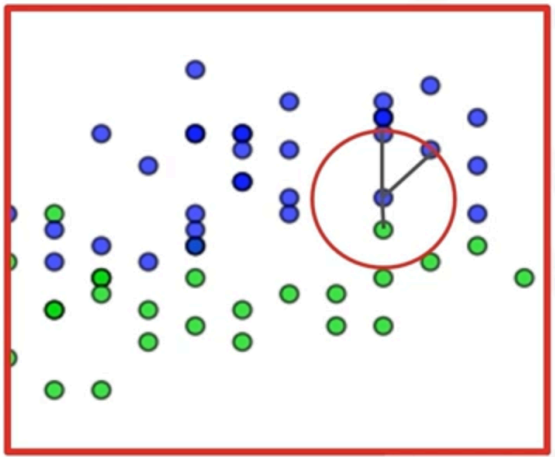
k-NN for classification or regression



Determining classes with k = 3

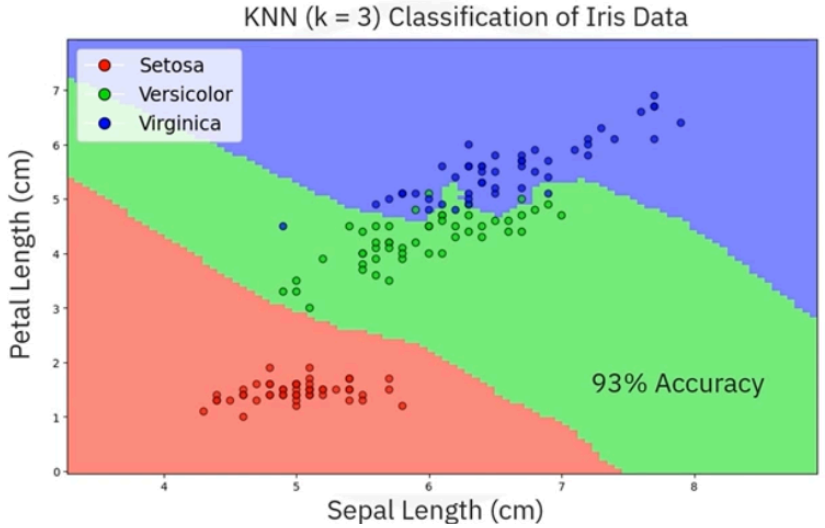


Determining classes with k = 3



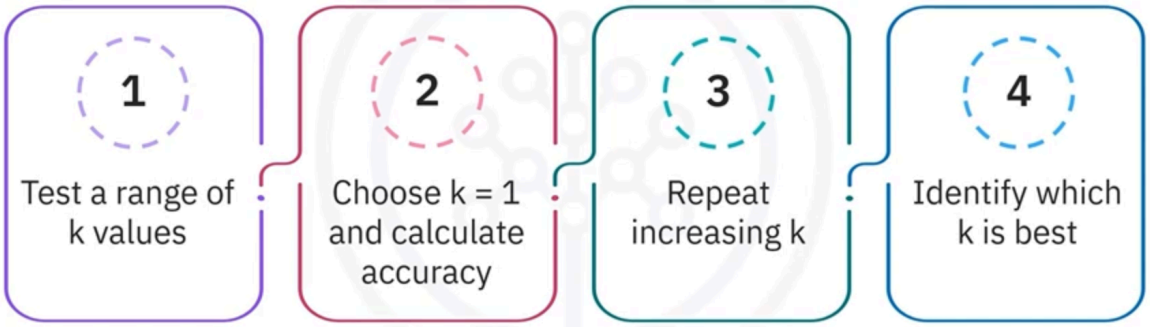
We use majority votes. Knn will classify this point as blue (viginica)

k-NN decision boundary



The idea here to find the optimal value for k and test properly.

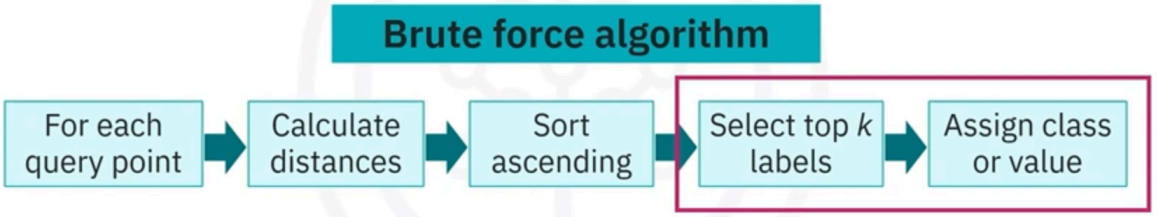
Finding the optimal k



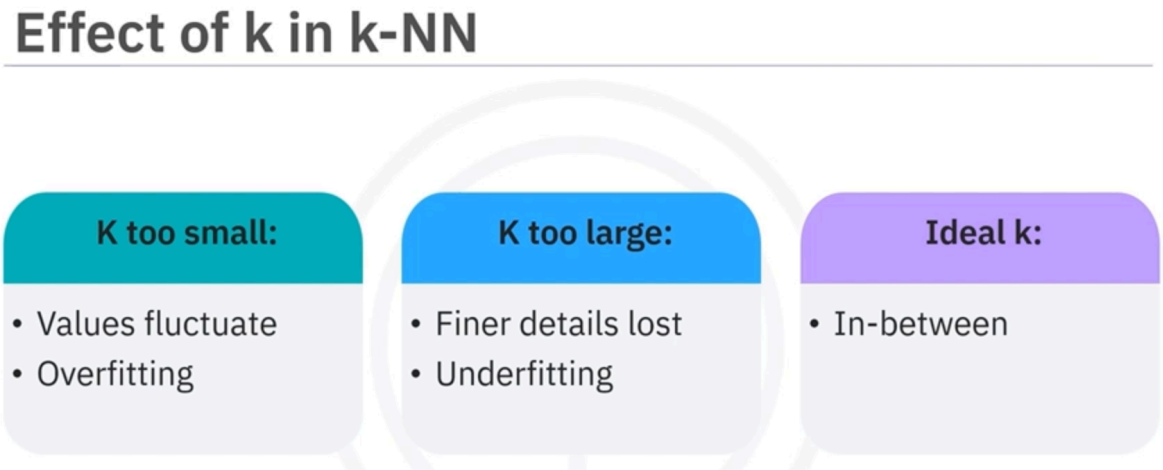
K-NN is a lazy learner.

K-NN is a lazy learner

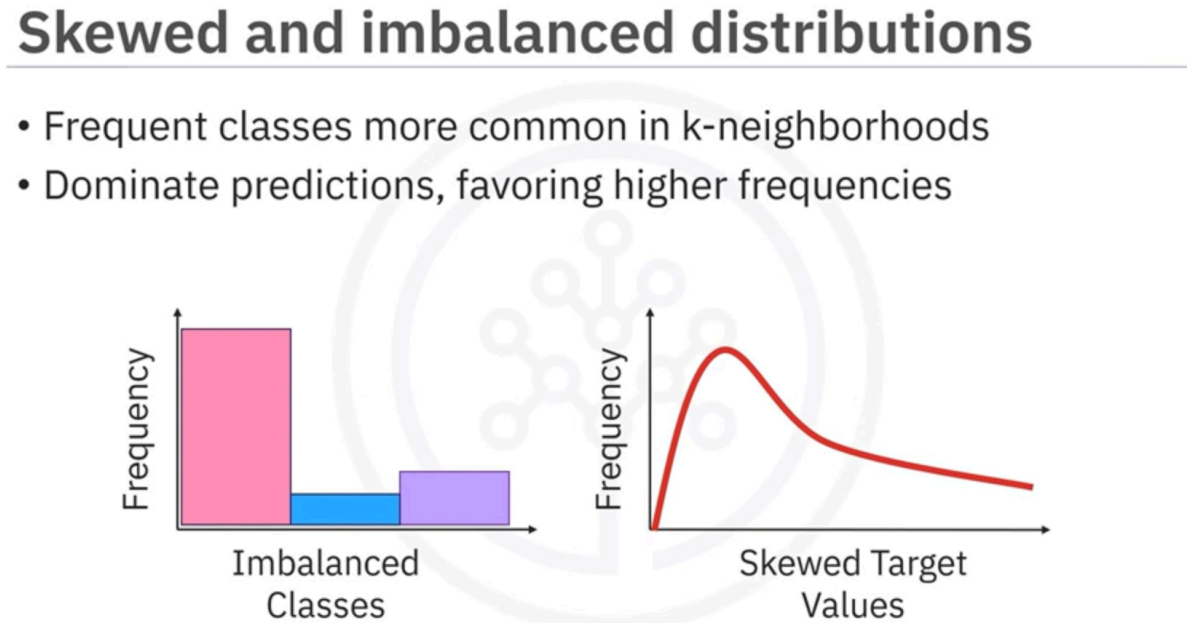
- Memorizes training data
- Makes predictions based on distance to training data points



Kk



Voting algorithms are bad when classes are skewed.



Another problem of knn are inconsistent features. How to fix unbalancing in Knn

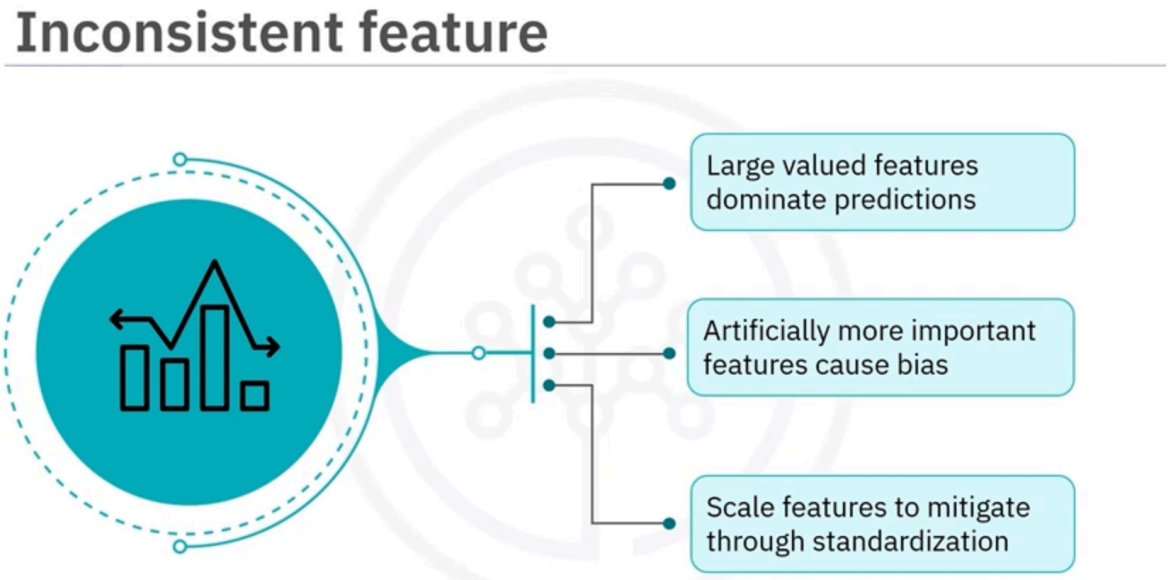
⚖️ 2. Use Weighted KNN (Distance or Class-Based)

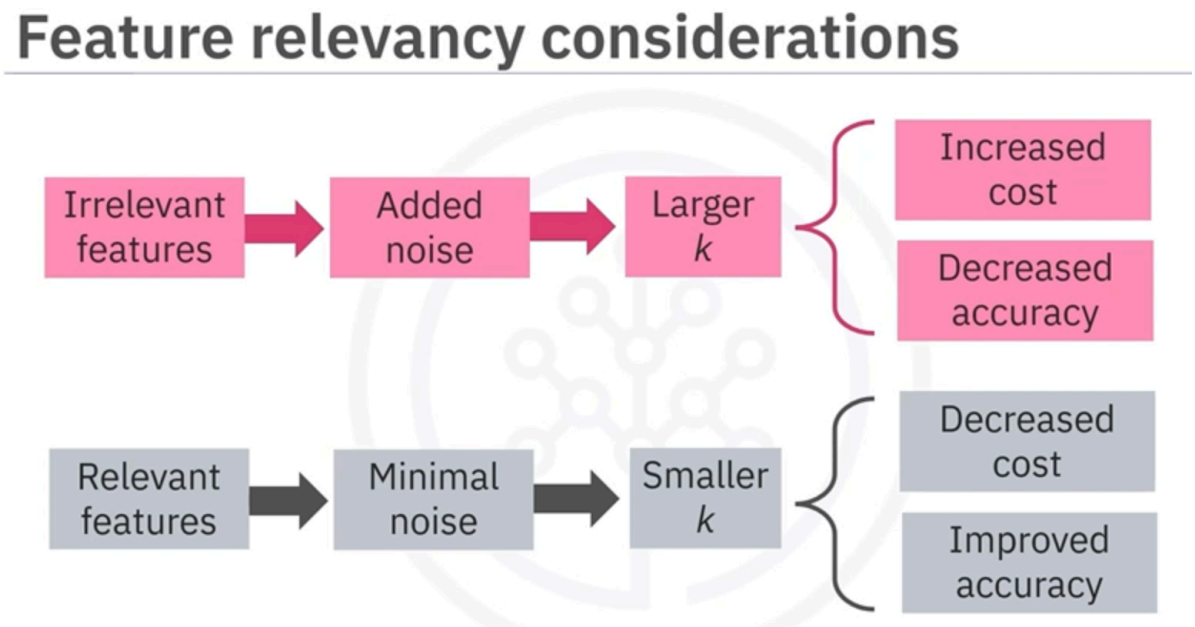
a. Distance-Weighted Voting

- Closer neighbors have more influence than farther ones.
- Common weights: 1/distance or 1/distance².

b. Class-Based Weighting

- Assign weights to neighbors based on **class frequencies**. Downweight majority class neighbors so the minority class has a fairer chance.





Recap

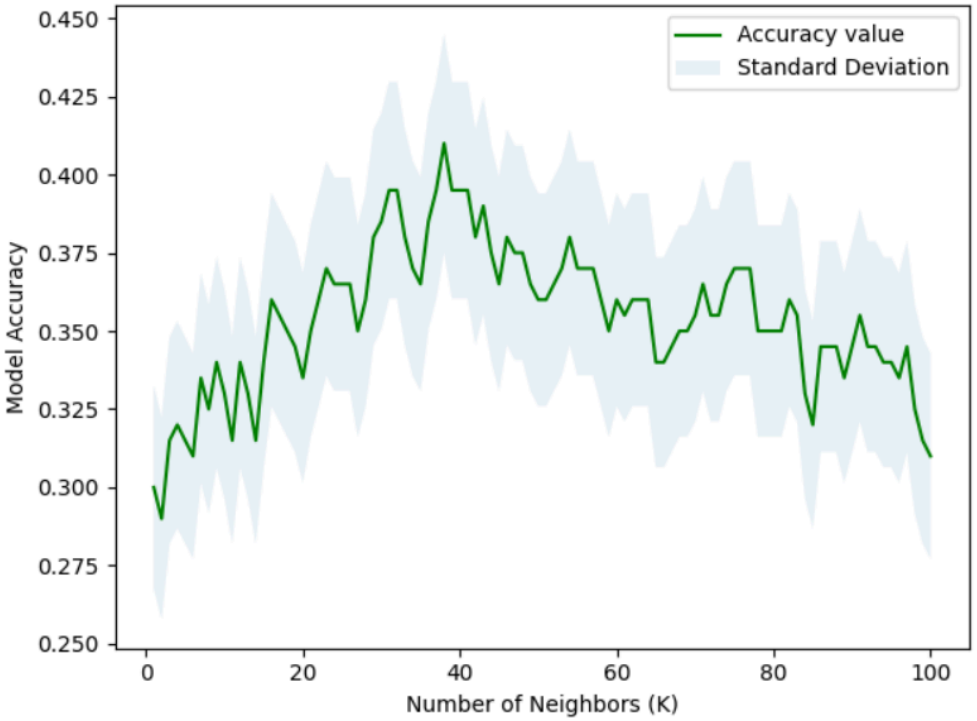
- k-NN: Uses labeled points to label other point
- k-NN is used for classification and regression
- Optimal k: Test values using a labeled data set and measure accuracy
- Skewed distribution causes drawback of majority voting classification
- Resolution:
  - Weight classification
  - Abstraction in data representation
- Selecting relevant features lowers optimal k and improves accuracy
- To select, iteratively test by tuning k

It might happen that

When k is small (e.g., k=1), the model is highly sensitive to the individual points in the dataset. The prediction for each point is based on its closest neighbor, which can lead to highly specific and flexible boundaries. This leads to overfitting on the training data, meaning the model will perform very well on the training set, potentially achieving 100% accuracy. However, it may generalize poorly to unseen data. When k is large, the model starts to take into account more neighbors when making predictions. This has two main consequences:

1. Smoothing of the Decision Boundary: The decision boundary becomes smoother, which means the model is less sensitive to the noise or fluctuations in the training data.
2. Less Specific Predictions: With a larger k, the model considers more neighbors and therefore makes more generalized predictions, which can lead to fewer instances being classified perfectly.

As a result, the model starts to become less flexible, and its ability to memorize the training data (which can lead to perfect accuracy with small k) is reduced.



Even with the optimal hyperparameters, the KNN model does not appear to perform well on this dataset. What could be the possible reasons for this?

Possible Reasons: There are several factors that might contribute to the model's weak performance:

1. **Dependence on Raw Feature Space:** KNN relies heavily on the original features during inference. If the features do not clearly separate the classes, the model struggles to make accurate predictions, as it lacks the ability to transform or optimize the feature space.

2. **Curse of Dimensionality:** When there are many features—especially ones that are weakly correlated with the target variable—the data becomes high-dimensional. In such spaces, distances between data points tend to become less meaningful, diminishing the effectiveness of the distance-based KNN approach.

3. **Equal Treatment of Features:** KNN treats all features equally when calculating distances. If some features are noisy or irrelevant, they can distort distance calculations and lead to poor neighbor selection, ultimately degrading model performance.

Why Use KNN?

1. **Simplicity & Intuition:**

    - KNN is incredibly easy to understand and implement.

- It follows a simple "memorization" approach: look at the nearest neighbors and let them vote.

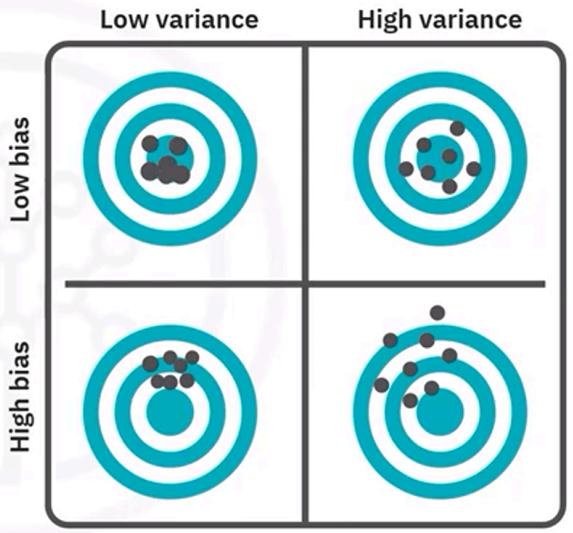
2. **No Training Phase:**
    - KNN is a *lazy learner*—it doesn’t build a model during training.
    - This makes it fast to set up and ideal for small datasets where training time isn’t a concern.
3. **Non-Parametric Nature:**
    - KNN makes no assumptions about the underlying data distribution.
    - This is useful when you’re unsure about the shape of the data or when it doesn't meet the assumptions of models like logistic regression (e.g., linearity).
4. **Adaptability to Multi-Class Problems:**
    - It naturally handles multi-class classification without any major changes.
5. **Works Well with Well-Separated Data:**
    - If your data is low-dimensional and class boundaries are clear, KNN can perform surprisingly well.

So it is one the simplest algorithm and you can use it when simplest conditions happen.

Bias, Variance, and Ensemble Models

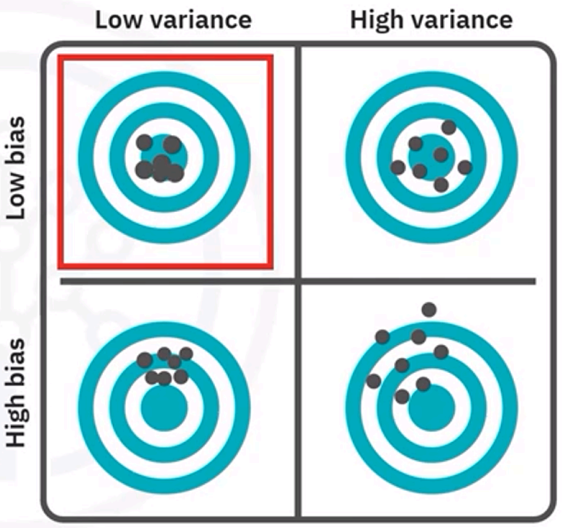
Bias and variance

- Darts near the center show:
  - High accuracy
  - Low bias
- Top boards demonstrate low bias, higher accuracy
- Bottom boards reflect high bias, lower accuracy
- Bias indicates how “on target” predictions are



Bias and variance

- Variance measures how spread out darts are
- Higher variance means darts are spread out
- Lower variance means darts are grouped closely
- High scores need low bias and variance



Prediction bias

Measures the accuracy of predictions

$$\text{Bias} = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i) = \frac{1}{N}\sum_{i=1}^{N}\hat{y}_i - \frac{1}{N}\sum_{i=1}^{N} y_i$$

Reflects differences from target values

Average prediction – average of actuals

Is zero for perfect predictors

$\hat{y}_i = y_i$ for all $i \implies$ Bias = 0

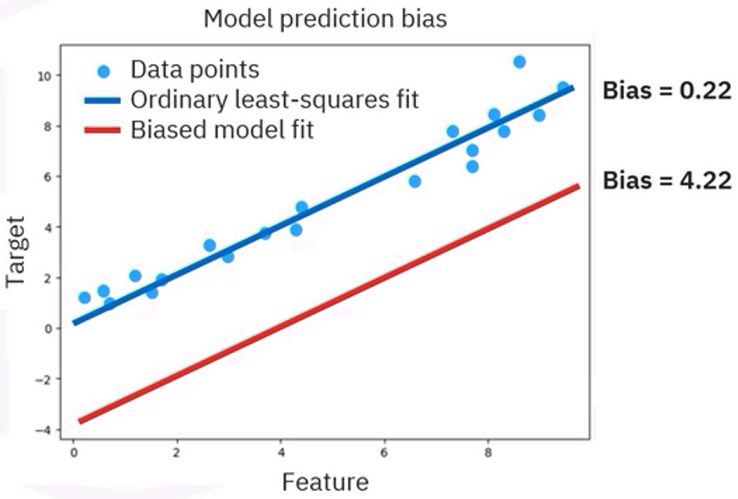
Prediction bias

Blue line:

- Shows least-squares fit
- Bias is 0.22

Red line:

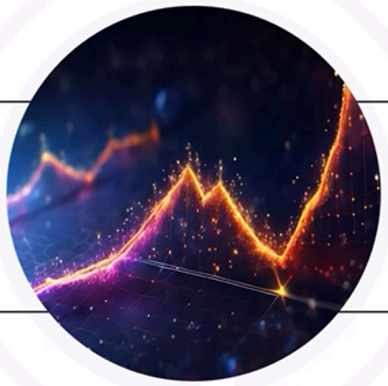
- Shifts model down by 4
- Bias is 4.22



Prediction variance

High variance:

- Leads to overfitting training data
- Tracks noise in training data

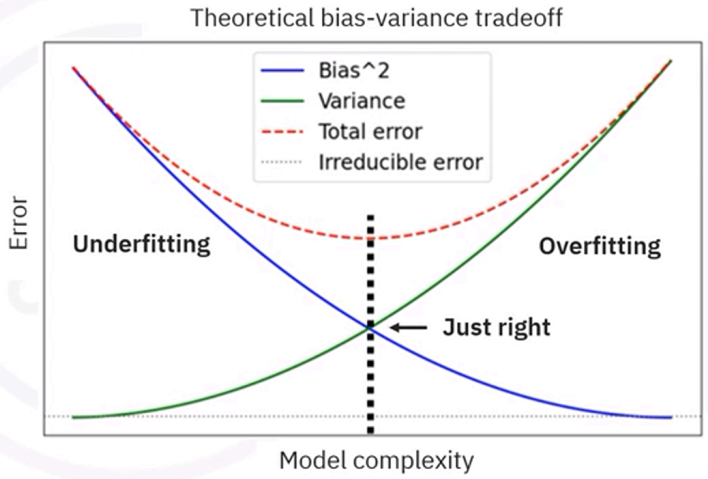


Low variance:

- Generalizes well to unseen data
- Means less sensitivity to noise

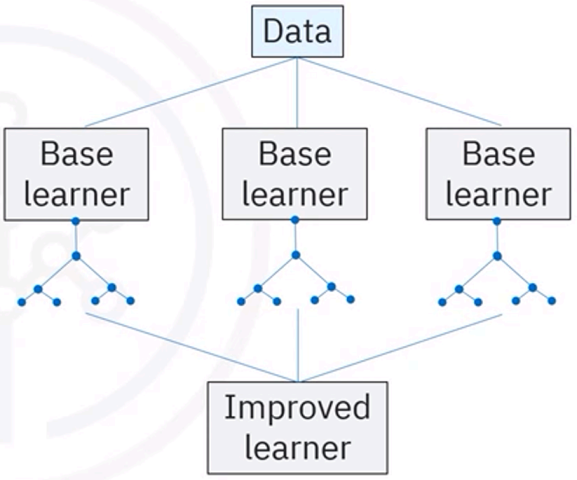
Bias-variance tradeoff

- Plot shows changes in bias and variance



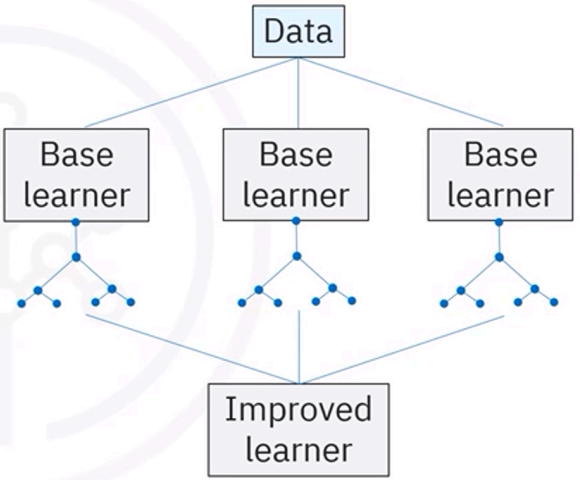
Mitigating bias and variance

- Weak learners perform slightly better than random guessing
- Weak learners have high bias, low variance
- High bias often leads to underfitting



Mitigating bias and variance

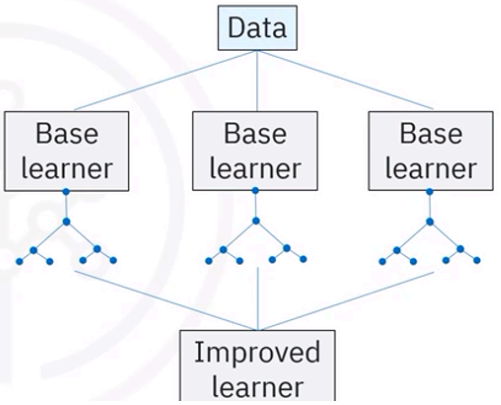
- Strong learners have low bias, high variance
- High variance often causes overfitting



Mitigating bias and variance

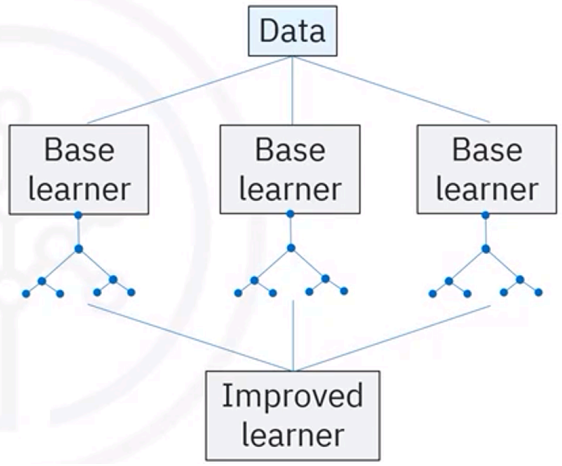
Bagging and boosting:

- Popular methods for ensemble learning
- Effective at balancing bias and variance

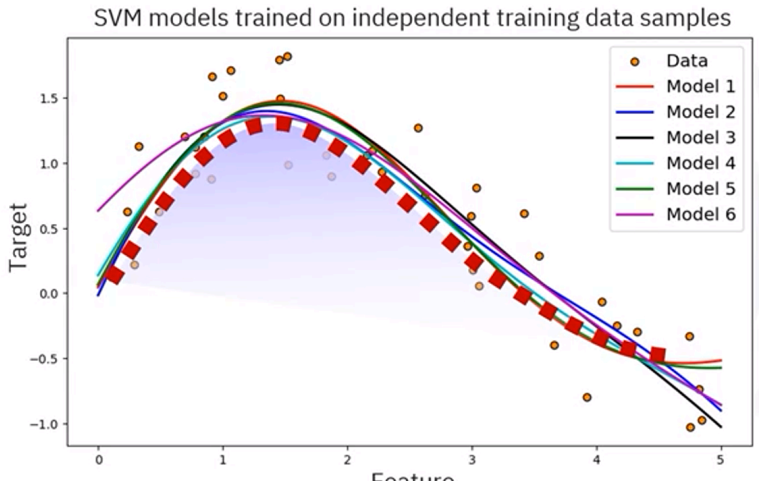


Mitigating bias and variance

- Decision trees serve as base learners
- Tree depth adjusts bias and variance



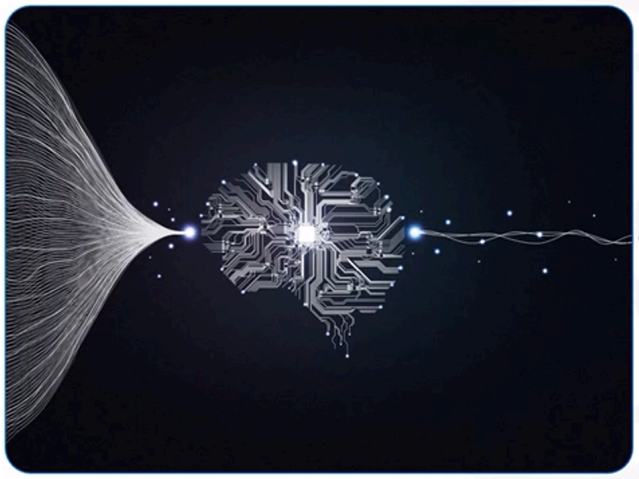
Bootstrap aggregating or bagging



Averaging the models:

- Reduces prediction variance
- Lowers the risk of overfitting

Random forests



Random forests

- Use bagging for training
- Train decision trees on bootstrapped data sets
- Focus on minimizing prediction bias

Boosting

Boosting builds a series of weak learners

Each learner corrects the previous learner's errors

Boosting systematically reduces prediction error

The final model is a weighted sum

Boosting

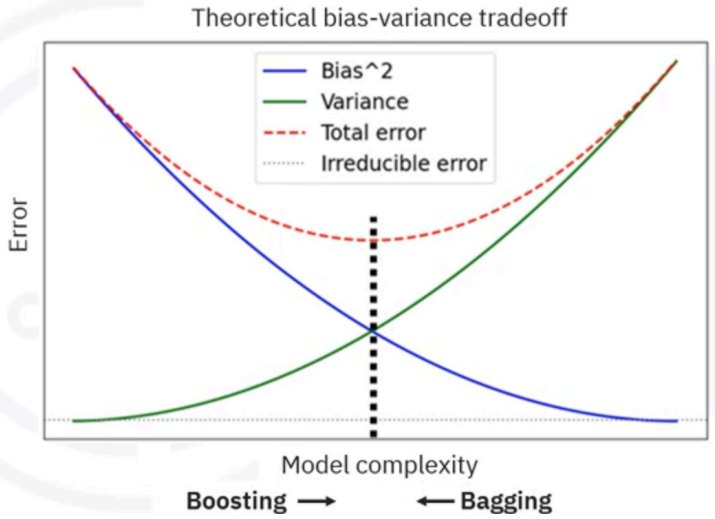
- Increase weights for misclassified data
- Decrease weights for correctly classified data
- Reweighting focuses on correcting mistakes
- Update model weights based on performance

Boosting increase model complexity and decrease bias, in contrast bagging increase variance

Bias-variance tradeoff

Boosting:
- Increases model complexity
- Decreases bias

Bagging:
- Increases variance



Bagging and boosting

Ensemble	Objective	Base Learners	Training	Outcome
Bagging	Mitigate overfitting	High variance Low bias	Parallel on bootstrapped data	Reduced variance
Boosting	Mitigate underfitting	Low variance, High bias	Builds on previous result	Reduced bias

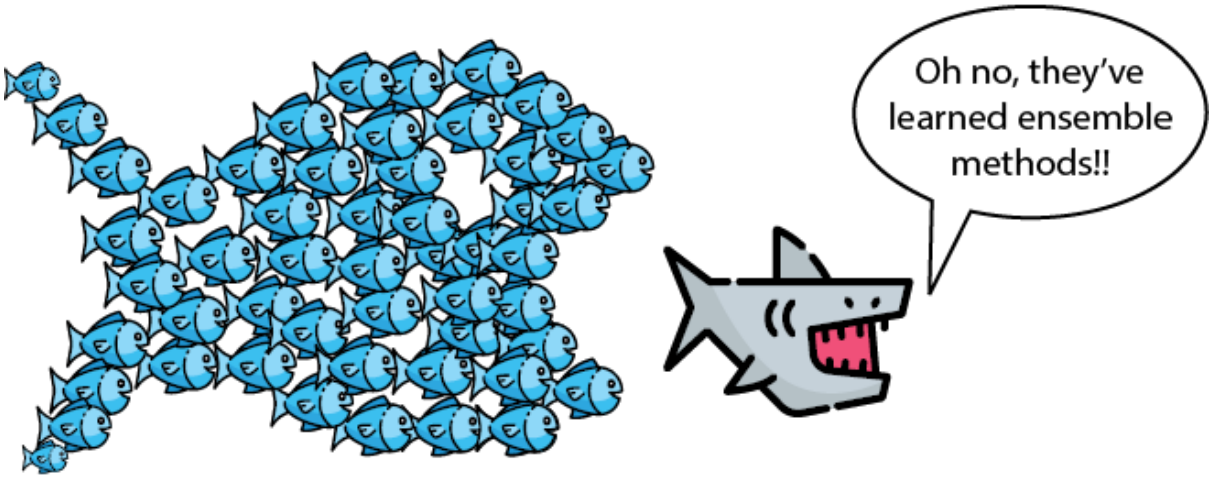
Recap

- Analyze bias and variance for accuracy and precision
- Explain prediction bias to measure prediction accuracy
- Analyze prediction variance to measure prediction fluctuations
- Explain the bias-variance tradeoff
- Explain mitigating bias and variance
- Analyze bagging or bootstrap aggregating to observe variance
- Explain random forests to train multiple decision trees
- Analyze bagging and boosting outcomes

12 Combining models to maximize results: Ensemble learning

In this chapter

- what ensemble learning is, and how it is used to combine weak classifiers into a stronger one
- using bagging to combine classifiers in a random way
- using boosting to combine classifiers in a cleverer way
- some of the most popular ensemble methods: random forests, AdaBoost, gradient boosting, and XGBoost



After learning many interesting and useful machine learning models, it is natural to wonder if it is possible to combine these classifiers. Thankfully, we can, and in this chapter, we learn several ways to build stronger models by combining weaker ones. The two main methods we learn in this chapter are bagging and boosting. In a nutshell, bagging consists of constructing a few models in a random way and joining them together. Boosting, on the other hand, consists of building these models in a smarter way by picking each model strategically to focus on the previous models' mistakes. The results that these ensemble methods have shown in important machine learning problems has been tremendous. For example, the Netflix Prize, which was awarded to the best model that fits a large dataset of Netflix viewership data, was won by a group that used an ensemble of different models.

In this chapter, we learn some of the most powerful and popular bagging and boosting models, including random forests, AdaBoost, gradient boosting, and XGBoost. The majority of these are described for classification, and some are described for regression. However, most of the ensemble methods work in both cases.

A bit of terminology: throughout this book, we have referred to machine learning models as models, or sometimes regressors or classifiers, depending on their task. In this chapter, we introduce the term *learner*, which also refers to a machine learning model. In the literature, it is common to use the terms *weak learner* and *strong learner* when talking about ensemble methods. However, there is no difference between a machine learning model and a learner.

All the code for this chapter is available in this GitHub repository: https://github.com/luisguiserrano/manning/tree/master/Chapter_12_Ensemble_Methods.

With a little help from our friends

Let's visualize ensemble methods using the following analogy: Imagine that we have to take an exam that consists of 100 true/false questions on many different topics, including math, geography, science, history, and music. Luckily, we are allowed to call our five friends—Adriana, Bob, Carlos, Dana, and Emily—to help us. There is a small constraint, which is that all of them work full time, and they don't have time to answer all 100 questions, but they are more than happy to help us with a subset of them. What techniques can we use to get their help? Two possible techniques follow:

Technique 1: For each of the friends, pick several random questions, and ask them to answer them (make sure every question gets an answer from at least one of our friends). After we get the responses, answer the test by selecting the option that was most popular among those who answered that question. For example, if two of our friends answered "True" and one answered "False" on question 1, then we answer question 1 as "True" (if there are ties, we can pick one of the winning responses randomly).

Technique 2: We give the exam to Adriana and ask her to answer only the questions she is the surest about. We assume that those answers are good and remove them from the test. Now we give the remaining questions to Bob, with the same instructions. We continue in this fashion until we pass it to all the five friends.

Technique 1 resembles a bagging algorithm, and technique 2 resembles a boosting algorithm. To be more specific, bagging and boosting use a set of models called *weak learners* and combine them into a *strong learner* (as illustrated in figure 12.1).

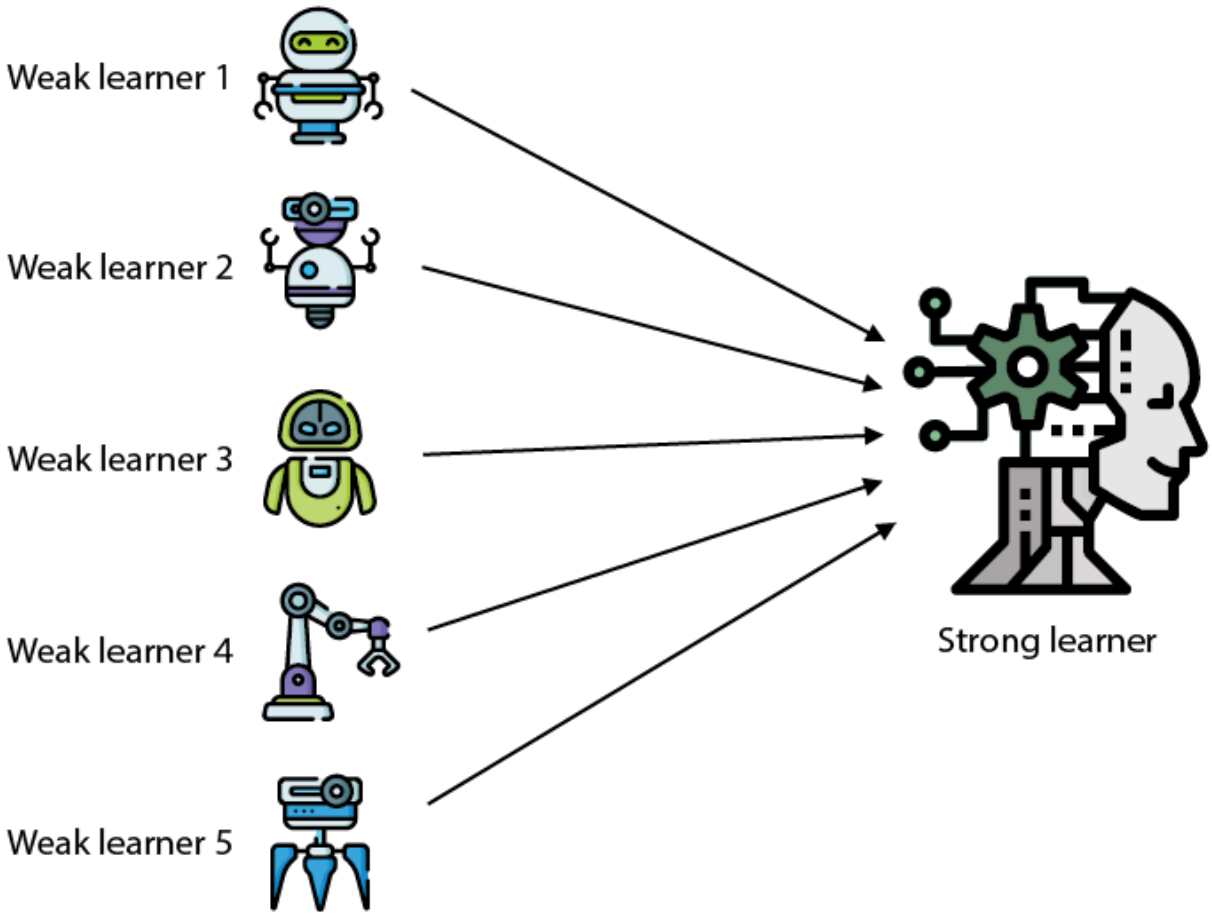


Figure 12.1 Ensemble methods consist of joining several weak learners to build a strong learner.

Bagging: Build random sets by drawing random points from the dataset (with replacement). Train a different model on each of the sets. These models are the weak learners. The strong learner is then formed as a combination of the weak models, and the prediction is done by voting (if it is a classification model) or averaging the predictions (if it is a regression model).

Boosting: Start by training a random model, which is the first weak learner. Evaluate it on the entire dataset. Shrink the points that have good predictions, and enlarge the points that have poor predictions. Train a second weak learner on this modified dataset. We continue in this fashion until we build several models. The way to combine them into a strong learner is the same way as with bagging, namely, by voting or by averaging the predictions of the weak learner. More specifically, if the learners are classifiers, the strong learner predicts the most common class predicted by the weak learners (thus the term *voting*), and if there are ties, by choosing randomly among them. If the learners are regressors, the strong learner predicts the average of the predictions given by the weak learners.

Most of the models in this chapter use decision trees (both for regression and classification) as the weak learners. We do this because decision trees lend themselves very well to this type of approach. However, as you read the chapter, I encourage you to think of how you would combine other types of models, such as perceptrons and SVMs.

We've spent an entire book building very good learners. Why do we want to combine several weak learners instead of simply building a strong learner from the start? One reason is that ensemble methods have been shown to overfit much less than other models. In a nutshell, it is easy for one model to overfit, but if you have several models for the same dataset, the combination of them overfits less. In a sense, it seems that if one learner makes a mistake, the others tend to correct it, and on average, they work better.

We learn the following models in this chapter. The first one is a bagging algorithm, and the last three are boosting:

- Random forests
- AdaBoost
- Gradient boosting
- XGBoost

All these models work for regression and classification. For educational purposes, we learn the first two as classification models and the last two as regression models. The process is similar for both classification and regression. However, read each of them and imagine how it would work in both cases. To learn how all these algorithms work for classification and regression, see the links to videos and reading material in appendix C that explain both cases in detail.

Bagging: Joining some weak learners randomly to build a strong learner

In this section we see one of the most well-known bagging models: a *random forest*. In a random forest, the weak learners are small decision trees trained on random subsets of the dataset. Random forests work well for classification and regression problems, and the process is similar. We will see random forests in a classification example. The code for this section follows:

- **Notebook**: Random_forests_and_AdaBoost.ipynb
    - https://github.com/luisguiserrano/manning/blob/master/Chapter_12_Ensemble_Methods/Random_forests_and_AdaBoost.ipynb

We use a small dataset of spam and ham emails, similar to the one we used in chapter 8 with the naive Bayes model. The dataset is shown in table 12.1 and plotted in figure 12.2. The features of the dataset are the number of times the words “lottery” and “sale” appear in the email, and the “yes/no” label indicates whether the email is spam (yes) or ham (no).

Table 12.1 Table of spam and ham emails, together with the number of appearances of the words “lottery” and “sale” on each email

Lottery	Sale	Spam
7	8	1
3	2	0
8	4	1
2	6	0
6	5	1
9	6	1
8	5	0
7	1	0
1	9	1
4	7	0
1	3	0
3	10	1
2	2	1
9	3	0
5	3	0
10	1	0
5	9	1
10	8	1

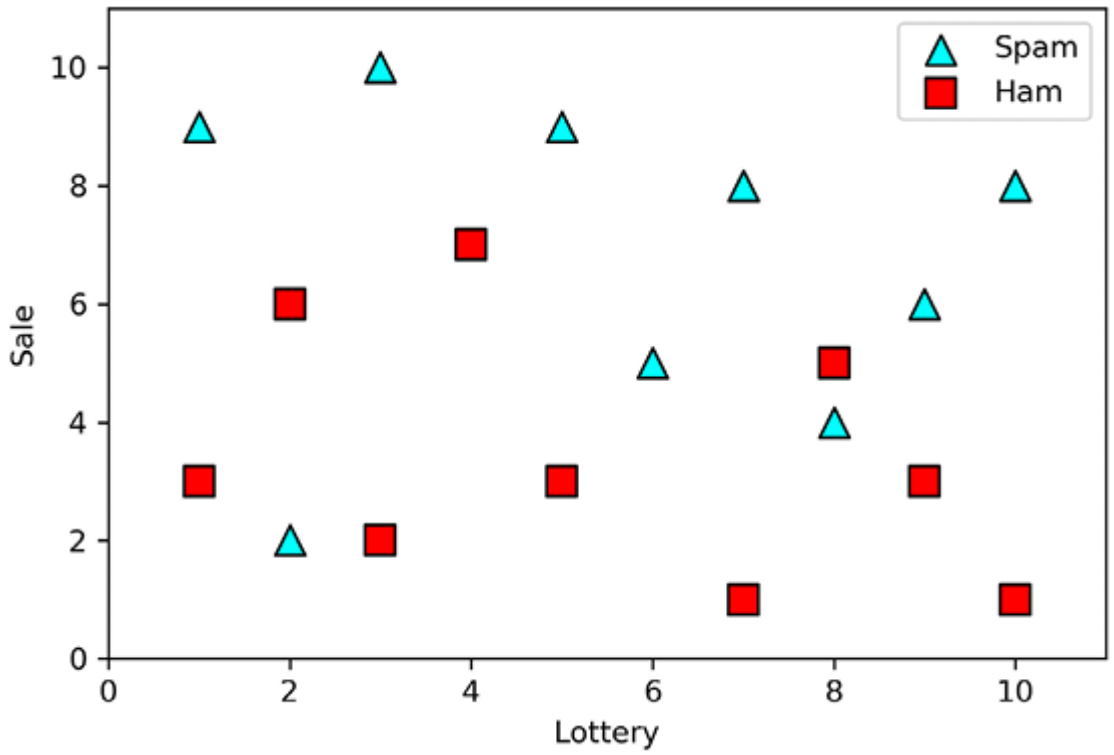


Figure 12.2 Figure 12.2 The plot of the dataset in table 12.1. Spam emails are represented by triangles and ham emails by squares. The horizontal and vertical axes represent the number of appearances of the words “lottery” and “sale,” respectively.

First, (over)fitting a decision tree

Before we get into random forests, let’s fit a decision tree classifier to this data and see how well it performs. Because we’ve learned this in chapter 9, figure 12.3 shows only the final result, but we can see the code in the notebook. On the left of figure 12.3, we can see the actual tree (quite deep!), and on the right, we can see the plot of the boundary. Notice that it fits the dataset very well, with a 100% training accuracy, although it clearly overfits. The overfitting can be noticed on the two outliers that the model tries to classify correctly, without noticing they are outliers.

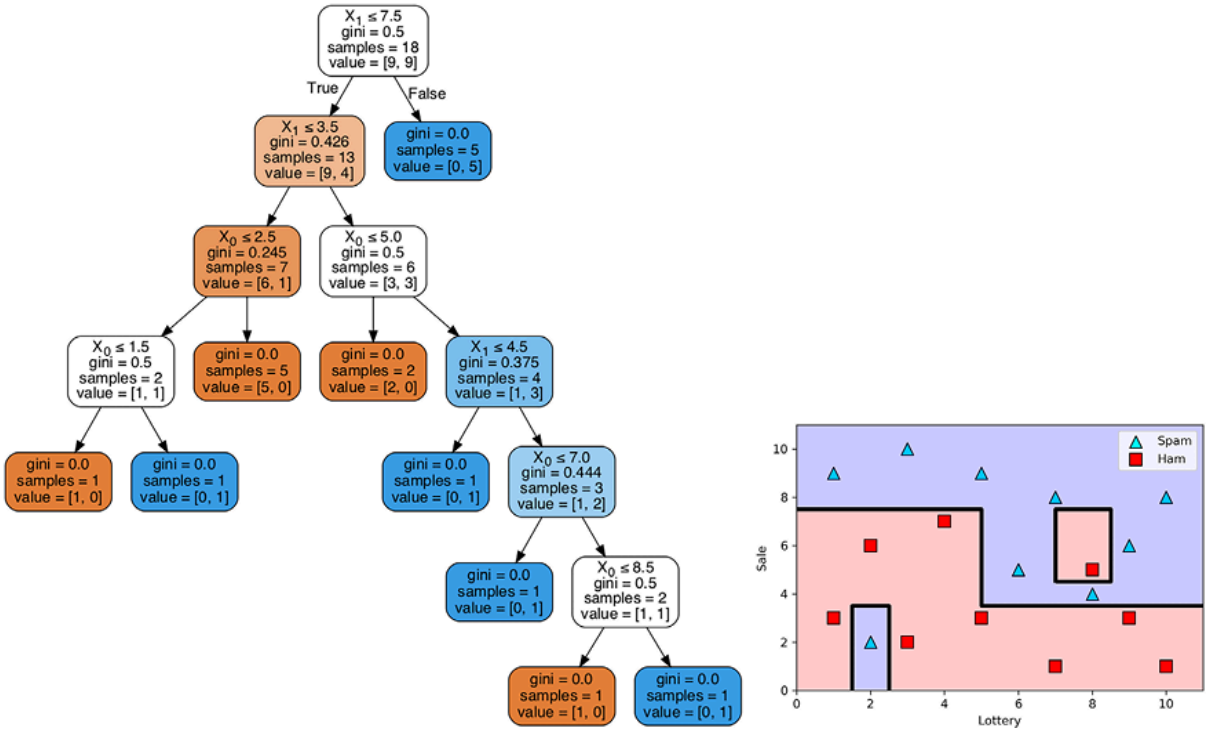


Figure 12.3 Left: A decision tree that classifies our dataset. Right: The boundary defined by this decision tree. Notice that it splits the data very well, although it hints at overfitting, because a good model would treat the two isolated points as outliers, instead of trying to classify them correctly.

In the next sections, we see how to solve this overfitting problem by fitting a random forest.

Fitting a random forest manually

In this section, we learn how to fit a random forest manually, although this is only for educational purposes, because this is not the way to do it in practice. In a nutshell, we pick random subsets from our dataset and train a weak learner (decision tree) on each one of them. Some data points may belong to several subsets, and others may belong to none. The combination of them is our strong learner. The way the strong learner makes predictions is by letting the weak learners vote. For this dataset, we use three weak learners. Because the dataset has 18 points, let's consider three subsets of 6 data points each, as shown in figure 12.4.

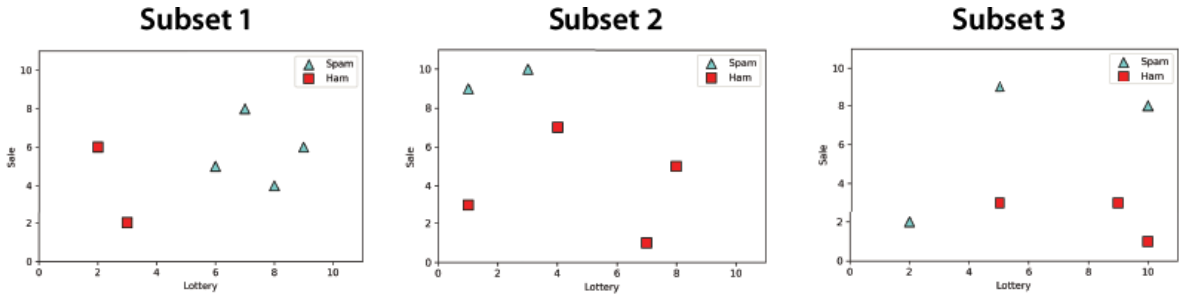


Figure 12.4 The first step to build a random forest is to split our data into three subsets. This is a splitting of the dataset shown in figure 12.2.

Next, we proceed to build our three weak learners. Fit a decision tree of depth 1 on each of these subsets. Recall from chapter 9 that a decision tree of depth 1 contains only one node and two leaves. Its boundary consists of a single horizontal or vertical line that splits the dataset as best as possible. The weak learners are illustrated in figure 12.5.

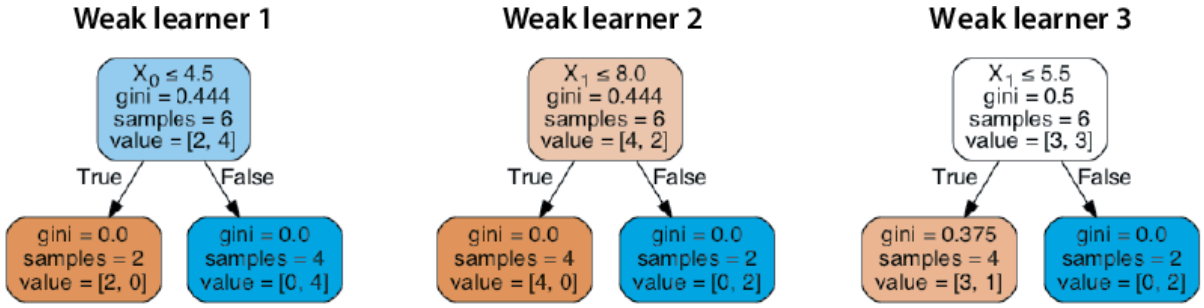


Figure 12.5 The three weak learners that form our random forest are decision trees of depth 1. Each decision tree fits one of the corresponding three subsets from figure 12.4.

We combine these into a strong learner by voting. In other words, for any input, each of the weak learners predicts a value of 0 or 1. The prediction the strong learner makes is the most common output of the three. This combination can be seen in figure 12.6, where the weak learners are on the top and the strong learner on the bottom.

Note that the random forest is a good classifier, because it classifies most of the points correctly, but it allows a few mistakes in order to not overfit the data. However, we don't need to train these random forests manually, because Scikit-Learn has functions for this, which we see in the next section.

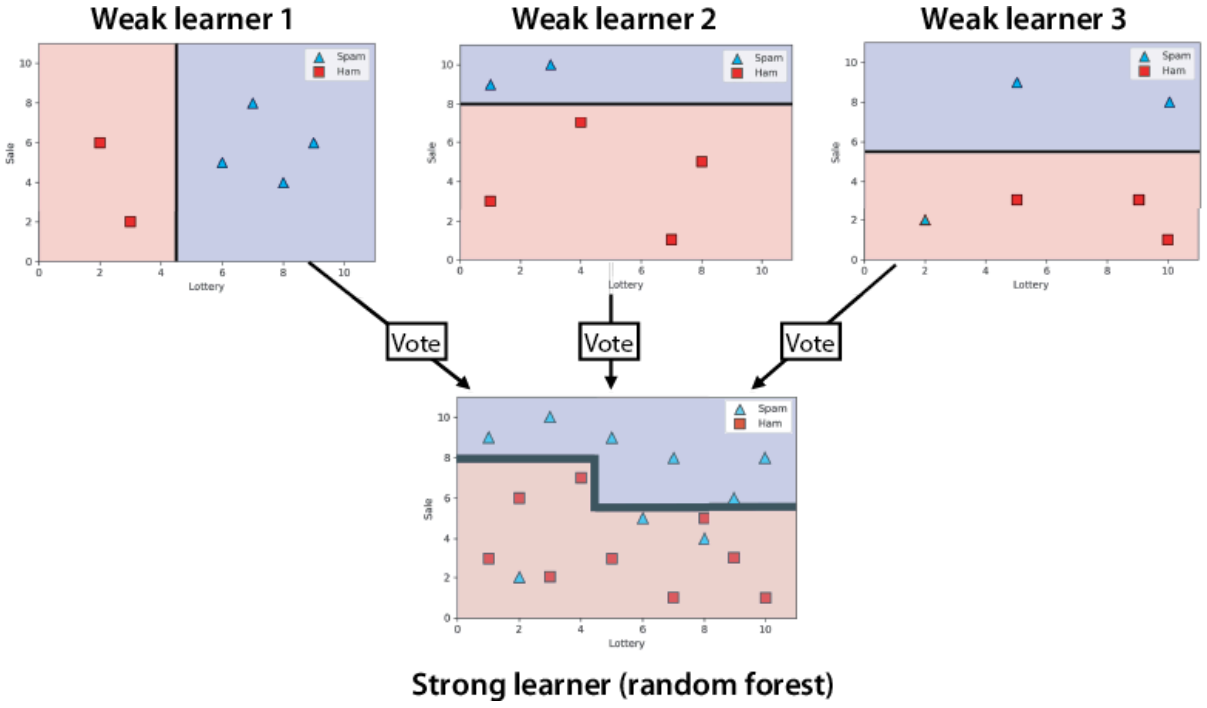


Figure 12.6 The way to obtain the predictions of the random forest is by combining the predictions of the three weak learners. On the top, we can see the three boundaries of the decision trees from figure

12.5. On the bottom, we can see how the three decision trees vote to obtain the boundary of the corresponding random forest.

Training a random forest in Scikit-Learn

In this section, we see how to train a random forest using Scikit-Learn. In the following code, we make use of the `RandomForestClassifier` package. To begin, we have our data in two Pandas DataFrames called `features` and `labels`, as shown next:

```
from sklearn.ensemble import RandomForestClassifier
random_forest_classifier = RandomForestClassifier(random_state=0, n_estimators=5, max_depth=1)
random_forest_classifier.fit(features, labels)
random_forest_classifier.score(features, labels)
```

In the previous code, we specified that we want five weak learners with the `n_estimators` hyperparameter. These weak learners are again decision trees, and we have specified that their depth is 1 with the `max_depth` hyperparameter. The plot of the model is shown in figure 12.7. Note how this model makes some mistakes but manages to find a good boundary, where the spam emails are those with a lot of appearances of the words “lottery” and “sale” (top right of the plot) and the ham emails are those with not many appearances of these words (bottom left of the figure).

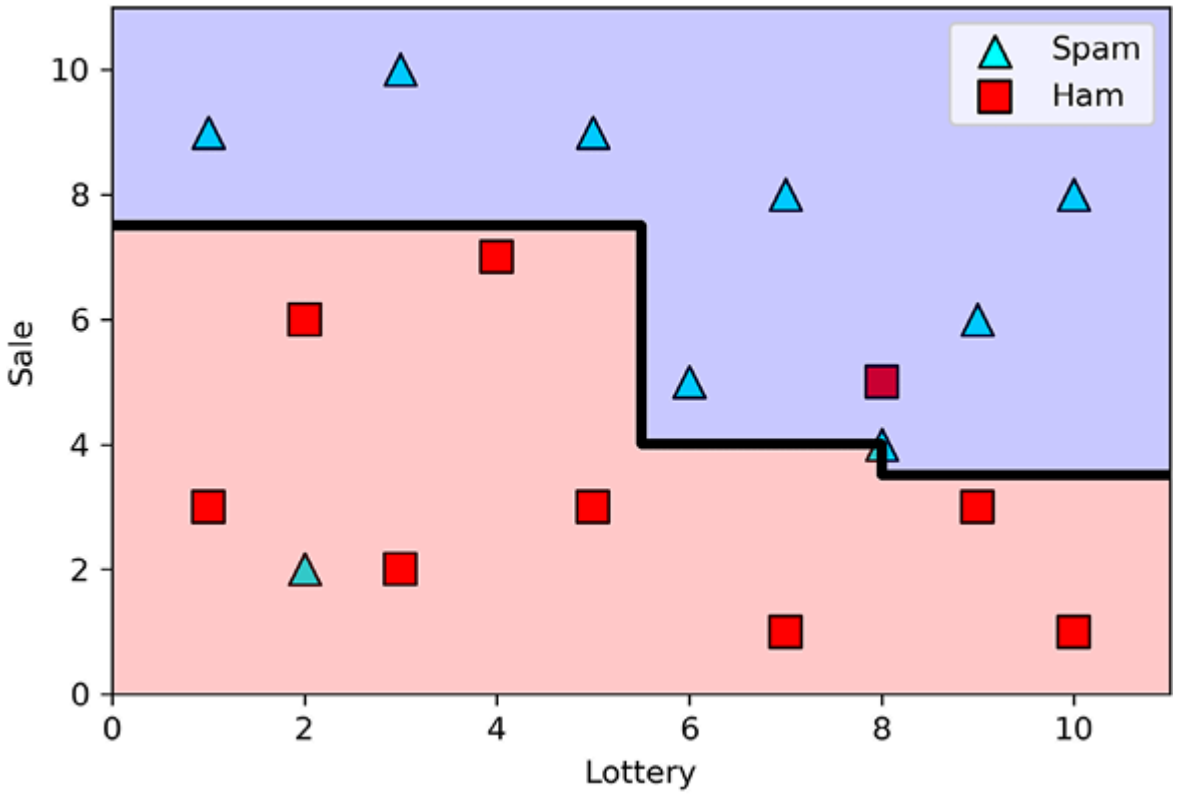


Figure 12.7 The boundary of the random forest obtained with Scikit-Learn. Notice that it classifies the dataset well, and it treats the two misclassified points as outliers, instead of trying to classify them correctly.

Scikit-Learn also allows us to visualize and plot the individual weak learners (see the notebook for the code). The weak learners are shown in figure 12.8. Notice that not all the weak learners are useful. For instance, the first one classifies every point as ham.

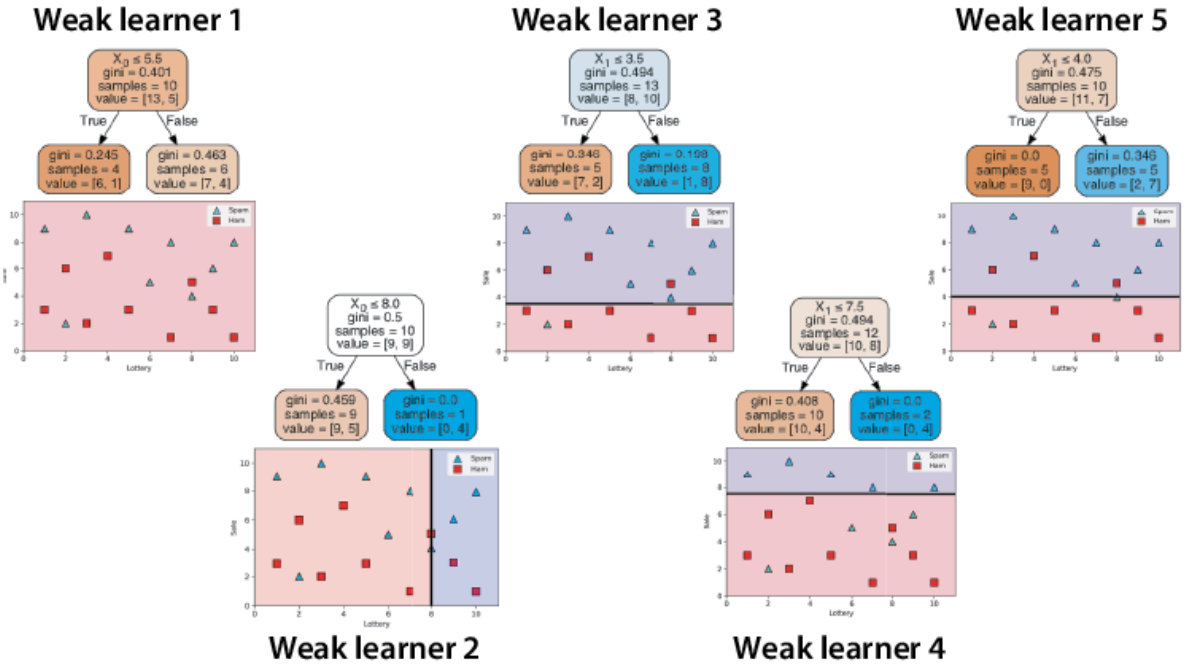


Figure 12.8 The random forest is formed by five weak learners obtained using Scikit-Learn. Each one is a decision tree of depth 1. They combine to form the strong learner shown in figure 12.7.

In this section, we used decision trees of depth 1 as weak learners, but in general, we can use trees of any depth we want. Try retraining this model using decision trees of higher depth by varying the `max_depth` hyperparameter, and see what the random forest looks like!

AdaBoost: Joining weak learners in a clever way to build a strong learner

Boosting is similar to bagging in that we join several weak learners to build a strong learner. The difference is that we don't select the weak learners at random. Instead, each learner is built by focusing on the weaknesses of the previous learners. In this section, we learn a powerful boosting technique called AdaBoost, developed by Freund and Schapire in 1997 (see appendix C for the reference). AdaBoost is short for adaptive boosting, and it works for regression and classification. However, we will use it in a classification example that illustrates the training algorithm very clearly.

In AdaBoost, like in random forests, each weak learner is a decision tree of depth 1. Unlike random forests, each weak learner is trained on the whole dataset, rather than on a portion of it. The only caveat is that after each weak learner is trained, we modify the dataset by enlarging the points that have been incorrectly classified, so that future weak learners pay more attention to these. In a nutshell, AdaBoost works as follows:

Pseudocode for training an AdaBoost model

- Train the first weak learner on the first dataset.
- Repeat the following step for each new weak learner:
    - After a weak learner is trained, the points are modified as follows:
        - The points that are incorrectly classified are enlarged.
    - Train a new weak learner on this modified dataset.

In this section, we develop this pseudocode in more detail over an example. The dataset we use has two classes (triangles and squares) and is plotted in figure 12.9.

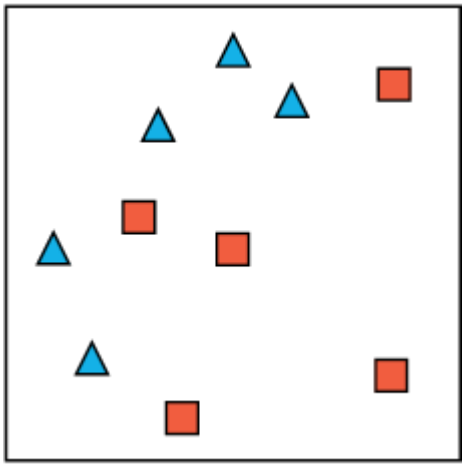


Figure 12.9 The dataset that we will classify using AdaBoost. It has two labels represented by a triangle and a square.

A big picture of AdaBoost: Building the weak learners

Over the next two subsections, we see how to build an AdaBoost model to fit the dataset shown in figure 12.9. First we build the weak learners that we'll then combine into one strong learner.

The first step is to assign to each of the points a weight of 1, as shown on the left of figure 12.10. Next, we build a weak learner on this dataset. Recall that the weak learners are decision trees of depth 1. A decision tree of depth 1 corresponds to the horizontal or vertical line that best splits the points. Several such trees do the job, but we'll pick one—the vertical line illustrated in the middle of figure 12.10—which correctly classifies the two triangles to its left and the five squares to its right, and incorrectly classifies the three triangles to its right. The next step is to enlarge the three incorrectly classified points to give them more importance under the eyes of future weak learners. To enlarge them, recall that each point initially has a weight of 1. We define the *rescaling factor* of this weak learner as the number of correctly classified points divided by the number of incorrectly classified points. In this case, the rescaling factor is 7/3 = 2.33. We proceed to rescale every misclassified point by this rescaling factor, as illustrated on the right of figure 12.10.

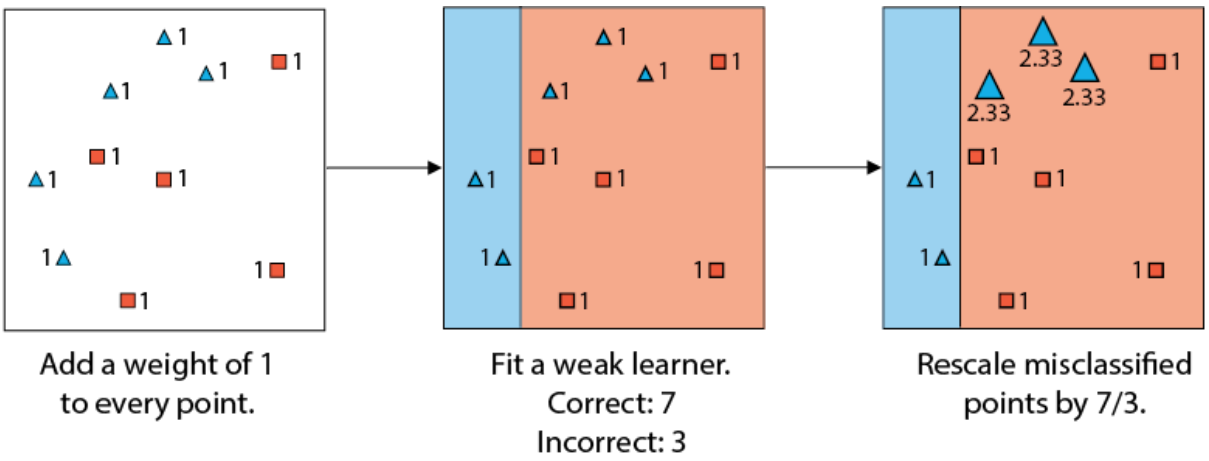


Figure 12.10 Fitting the first weak learner of the AdaBoost model. Left: The dataset, where each point gets assigned a weight of 1. Middle: A weak learner that best fits this dataset. Right: The rescaled dataset, where we have enlarged the misclassified points by a rescaling factor of 7/3.

Now that we've built the first weak learner, we build the next ones in the same manner. The second weak learner is illustrated in figure 12.11. On the left of the figure, we have the rescaled dataset. The second weak learner is one that fits this dataset best. What do we mean by that? Because points have different weights, we want the weak learner for which the sum of the weights of the correctly classified points is the highest. This weak learner is

the horizontal line in the middle of figure 12.11. We now proceed to calculate the rescaling factor. We need to slightly modify its definition, because the points now have weights. The rescaling factor is the ratio between the sum of the weights of the correctly classified points and the sum of the weights of the incorrectly classified points. The first term is 2.33 + 2.33 + 2.33 + 1 + 1 + 1 + 1 = 11, and the second is 1 + 1 + 1 = 3. Thus, the rescaling factor is 11/3 = 3.67. We proceed to multiply the weights of the three misclassified points by this factor of 3.67, as illustrated on the right of figure 12.11.

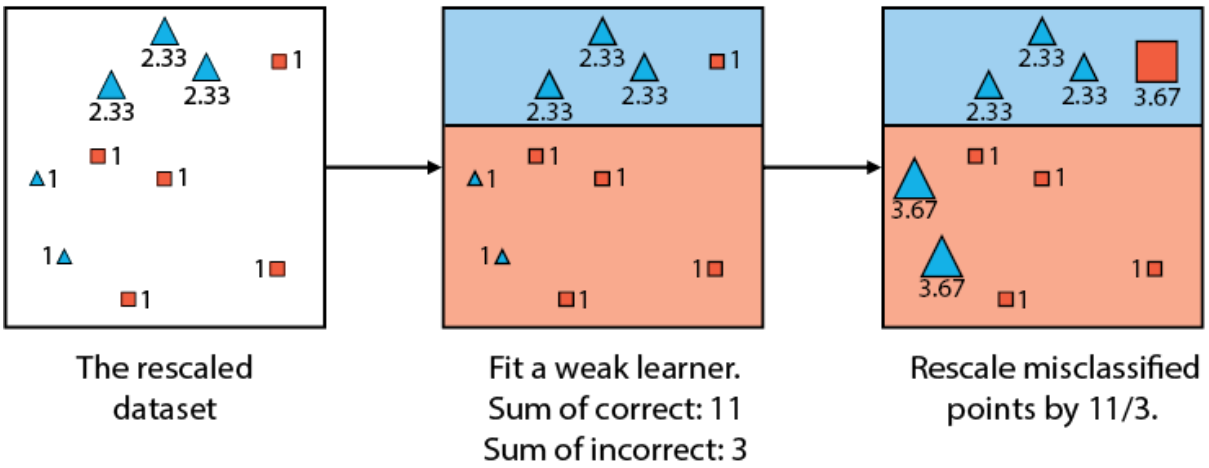


Figure 12.11 Fitting the second weak learner of the AdaBoost model. Left: The rescaled dataset from figure 12.10. Middle: A weak learner that best fits the rescaled dataset—this means, the weak learner for which the sum of weights of the correctly classified points is the largest. Right: The new rescaled dataset, where we have enlarged the misclassified points by a rescaling factor of 11/3.

We continue in this fashion until we've built as many weak learners as we want. For this example, we build only three weak learners. The third weak learner is a vertical line, illustrated in figure 12.12.

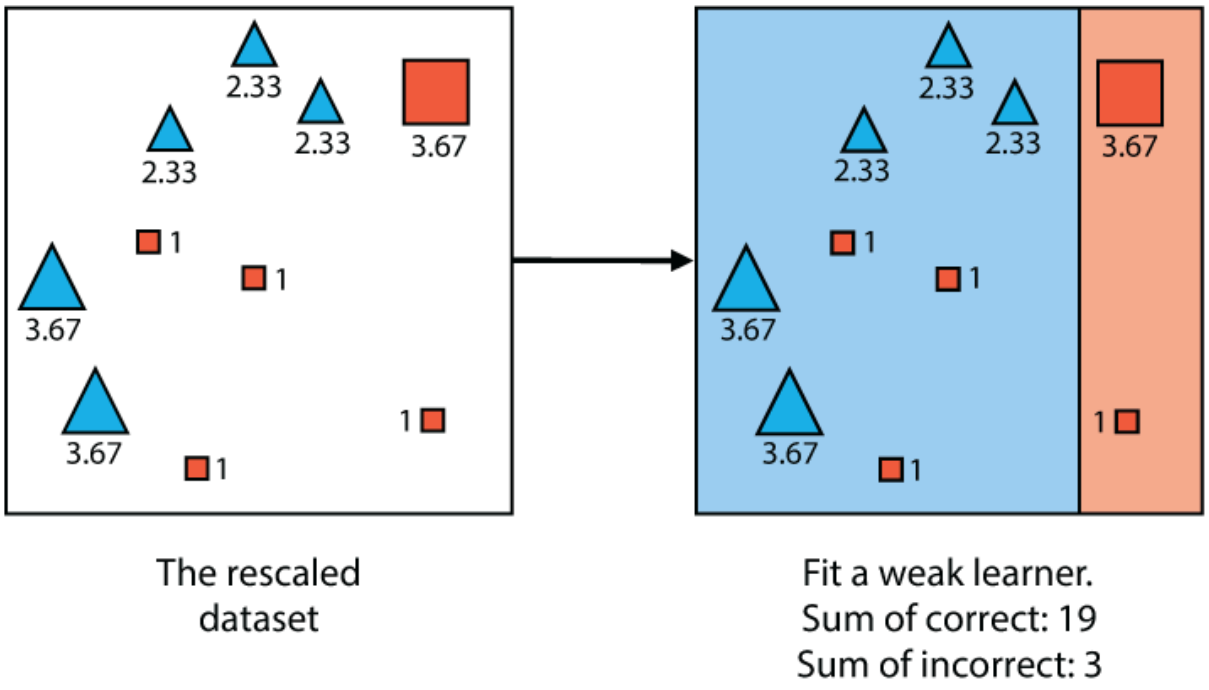


Figure 12.12 Fitting the third weak learner of the AdaBoost model. Left: The rescaled dataset from figure 12.11. Right: A weak learner that best fits this rescaled dataset.

This is how we build the weak learners. Now, we need to combine them into a strong learner. This is similar to what we did with random forests, but using a little more math, as shown in the next section.

Combining the weak learners into a strong learner

Now that we've built the weak learners, in this section, we learn an effective way to combine them into a strong learner. The idea is to get the classifiers to vote, just as they did in the random forest classifier, but this time, good learners get more of a say than poor learners. In the event that a classifier is *really* bad, then its vote will actually be negative.

To understand this, imagine we have three friends: Truthful Teresa, Unpredictable Umbert, and Lying Lenny. Truthful Teresa almost always tells the truth, Lying Lenny almost always lies, and Unpredictable Umbert says the truth roughly half of the time and lies the other half. Out of these three friends, which is the least useful one?

The way I see it, Truthful Teresa is very reliable, because she almost always tells the truth, so we can trust her. Among the other two, I prefer Lying Lenny. If he almost always lies when we ask him a yes-or-no question, we simply take as truth the opposite of what he tells us, and we'll be correct most of the time! On the other hand, Unpredictable Umbert serves us no purpose if we have no idea whether he's telling the truth or lying. In that case, if we were to assign a score to what each friend says, I'd give Truthful Teresa a high positive score, Lying Lenny a high negative score, and Unpredictable Umbert a score of zero.

Now imagine that our three friends are weak learners trained in a dataset with two classes. Truthful Teresa is a classifier with very high accuracy, Lying Lenny is one with very low accuracy, and Unpredictable Umbert is one with an accuracy close to 50%. We want to build a strong learner where the prediction is obtained by a weighted vote from the three weak learners. Thus, to each of the weak learners, we assign a score, and that is how much the vote of the learner will count in the final vote. Furthermore, we want to assign these scores in the following way:

- The Truthful Teresa classifier gets a high positive score.
- The Unpredictable Umbert classifier gets a score close to zero.
- The Lying Lenny classifier gets a high negative score.

In other words, the score of a weak learner is a number that has the following properties:

1. Is positive when the accuracy of the learner is greater than 0.5
2. Is 0 when the accuracy of the model is 0.5
3. Is negative when the accuracy of the learner is smaller than 0.5
4. Is a large positive number when the accuracy of the learner is close to 1
5. Is a large negative number when the accuracy of the learner is close to 0

To come up with a good score for a weak learner that satisfies properties 1–5 above, we use a popular concept in probability called the *logit*, or *log-odds*, which we discuss next.

Probability, odds, and log-odds

You may have seen in gambling that probabilities are never mentioned, but they always talk about *odds*. What are these odds? They are similar to probability in the following sense: if we run an experiment many times and record the number of times a particular outcome occurred, the probability of this outcome is the number of times it occurred divided by the total number of times we ran the experiment. The odds of this outcome are the number of times it occurred divided by the number of times it didn't occur.

For example, the probability of obtaining 1 when we roll a die is 1/6, but the odds are 1/5. If a particular horse wins 3 out of every 4 races, then the probability of that horse winning a race is 3/4, and the odds are 3/1 = 3. The formula for odds is simple: if the probability of an event is x, then the odds are $\frac{x}{1-x}$. For instance, in the dice example, the probability is 1/6 and the odds are

$$\frac{\frac{1}{6}}{1-\frac{1}{6}}=\frac{1}{5}.$$

Notice that because the probability is a number between 0 and 1, then the odds are a number between 0 and ∞.

Now let's get back to our original goal. We are looking for a function that satisfies properties 1–5 above. The odds function is close, but not quite there, because it outputs only positive values. The way to turn the odds into a function that satisfies properties 1–5 above is by taking the logarithm. Thus, we obtain the log-odds, also called the logit, defined as follows:

$$\text{log-odds}(x) = ln\frac{x}{1-x}$$

Figure 12.13 shows the graph of the log-odds function $y = ln\left(\frac{x}{1-x}\right)$. Notice that this function satisfies properties 1–5.

Therefore, all we need to do is use the log-odds function to calculate the score of each of the weak learners. We apply this log-odds function on the accuracy. Table 12.2 contains several values for the accuracy of a weak learner and the log-odds of this accuracy. Notice that, as desired, models with high accuracy have high positive scores, models with low accuracy have high negative scores, and models with accuracy close to 0.5 have scores close to 0.

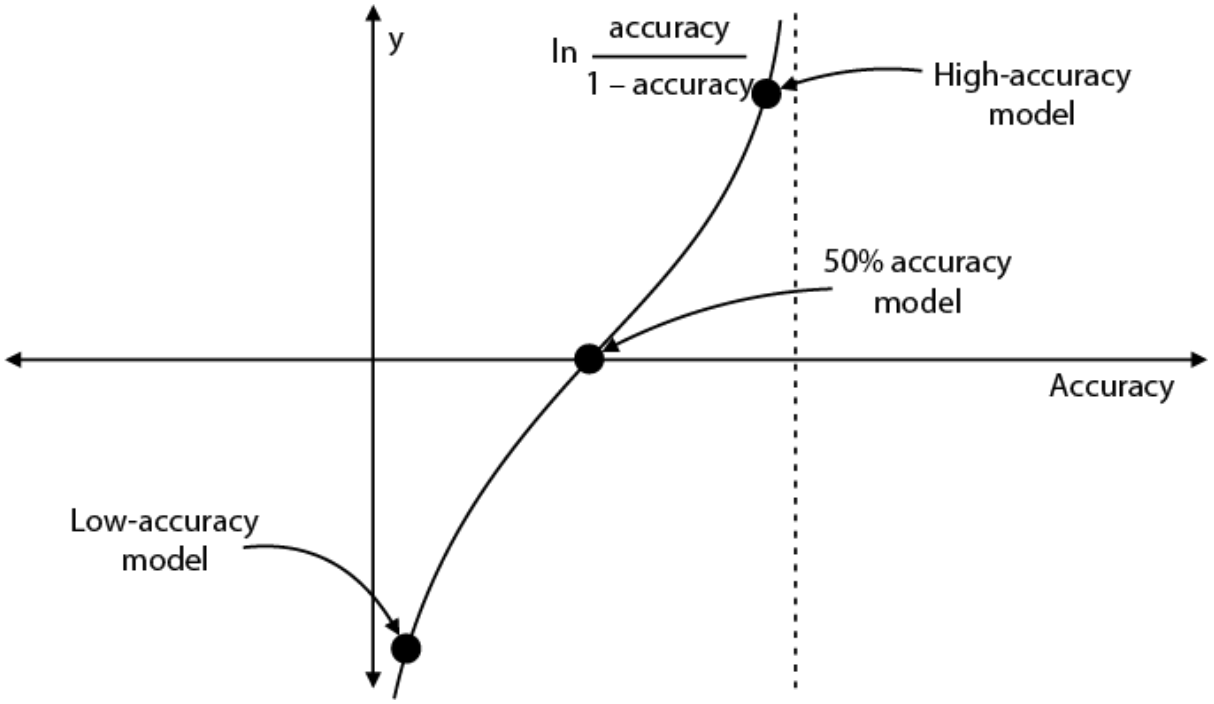


Figure 12.13 The curve shows the plot of the log-odds function with respect to the accuracy. Notice that for small values of the accuracy, the log-odds is a very large negative number, and for higher values of the accuracy, it is a very large positive number. When the accuracy is 50% (or 0.5), the log-odds is precisely zero.

Table 12.2 Several values for the accuracy of a weak classifier, with the corresponding score, calculated using the log-odds. Notice that the models with very low accuracy get large negative scores, the values with very high accuracy get large positive scores, and the values with accuracy close to 0.5 get scores close to 0.

Accuracy	Log-odds (score of the weak learner)
0.01	–4.595
0.1	–2.197

0.2	–1.386
0.5	0
0.8	1.386
0.9	2.197
0.99	4.595

Combining the classifiers

Now that we've settled on the log-odds as the way to define the scores for all the weak learners, we can proceed to join them to build the strong learner. Recall that the accuracy of a weak learner is the sum of the scores of the correctly classified points divided by the sum of the scores of all the points, as shown in figures 12.10–12.12.

- Weak learner 1:
  - Accuracy: $\frac{7}{10}$
  - Score: $ln\left(\frac{7}{3}\right) = 0.847$
- Weak learner 2:
  - Accuracy: $\frac{11}{14}$
  - Score: $ln\left(\frac{11}{3}\right) = 1.299$
- Weak learner 3:
  - Accuracy: $\frac{19}{22}$
  - Score: $ln\left(\frac{19}{3}\right) = 1.846$

The prediction that the strong learner makes is obtained by the weighted vote of the weak classifiers, where each classifier's vote is its score. A simple way to see this is to change the predictions of the weak learners from 0 and 1 to –1 and 1, multiplying each prediction by the score of the weak learner, and adding them. If the resulting prediction is greater than or equal to zero, then the strong learner predicts a 1, and if it is negative, then it predicts a 0. The voting process is illustrated in figure 12.14, and the predictions in figure 12.15. Notice also in figure 12.15 that the resulting classifier classified every point in the dataset correctly.

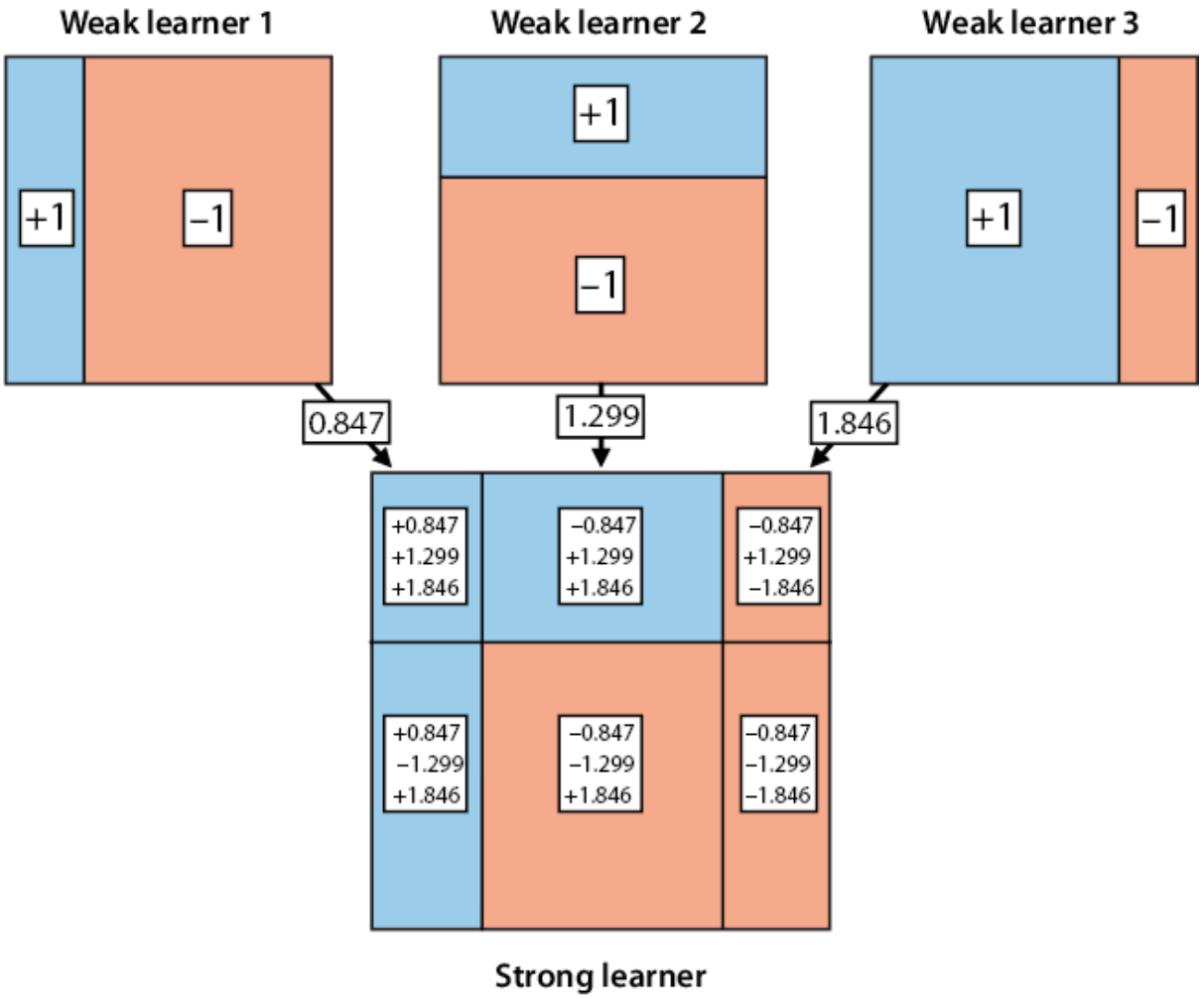


Figure 12.14 How to combine the weak learners into a strong learner in the AdaBoost model. We score each of the weak learners using the log-odds and make them vote based on their scores (the larger the score, the more voting power that particular learner has). Each of the regions in the bottom diagram has the sum of the scores of the weak learners. Note that to simplify our calculations, the predictions from the weak learners are +1 and –1, instead of 1 and 0.

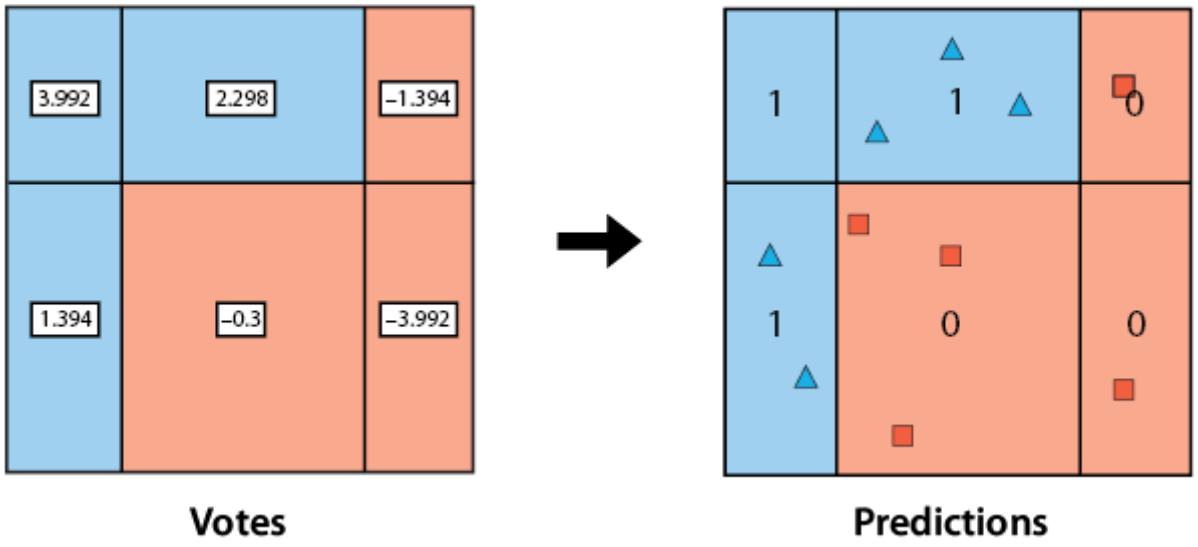


Figure 12.15 How to obtain the predictions for the AdaBoost model. Once we have added the scores coming from the weak learners (shown in figure 12.14), we assign a prediction of 1 if the sum of scores is greater than or equal to 0 and a prediction of 0 otherwise.

Coding AdaBoost in Scikit-Learn

In this section, we see how to use Scikit-Learn to train an AdaBoost model. We train it on the same spam email dataset that we used in the section "Fitting a random forest manually" and plotted in figure 12.16. We continue using the following notebook from the previous sections:

- **Notebook**: Random_forests_and_AdaBoost.ipynb
  - https://github.com/luisguiserrano/manning/blob/master/Chapter_12_Ensemble_Methods/Random_forests_and_AdaBoost.ipynb

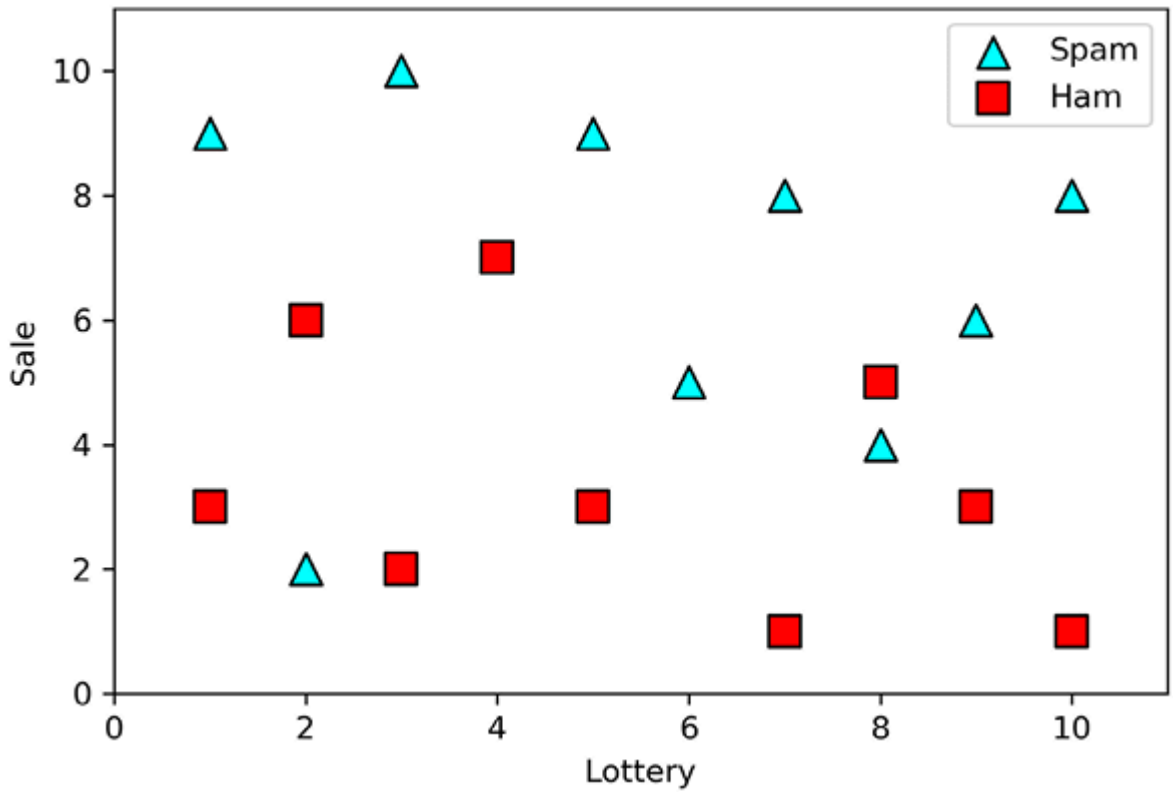


Figure 12.16 In this dataset, we train an AdaBoost classifier using Scikit-Learn. This is the same spam dataset from the section "Bagging," where the features are the number of appearances of the words "lottery" and "spam," and the spam emails are represented by triangles and the ham emails by squares.

The dataset is in two Pandas DataFrames called `features` and `labels`. The training is done using the `AdaBoostClassifier` package in Scikit-Learn. We specify that this model will use six weak learners with the `n_estimators` hyperparameter, as shown next:

```
from sklearn.ensemble import AdaBoostClassifier
adaboost_classifier = AdaBoostClassifier(n_estimators=6)
adaboost_classifier.fit(features, labels)
adaboost_classifier.score(features, labels)
```

The boundary of the resulting model is plotted in figure 12.17.

We can go a bit further and explore the six weak learners and their scores (see notebook for the code). Their boundaries are plotted in figure 12.18, and as is evident in the notebook, the scores of all the weak learners are 1.

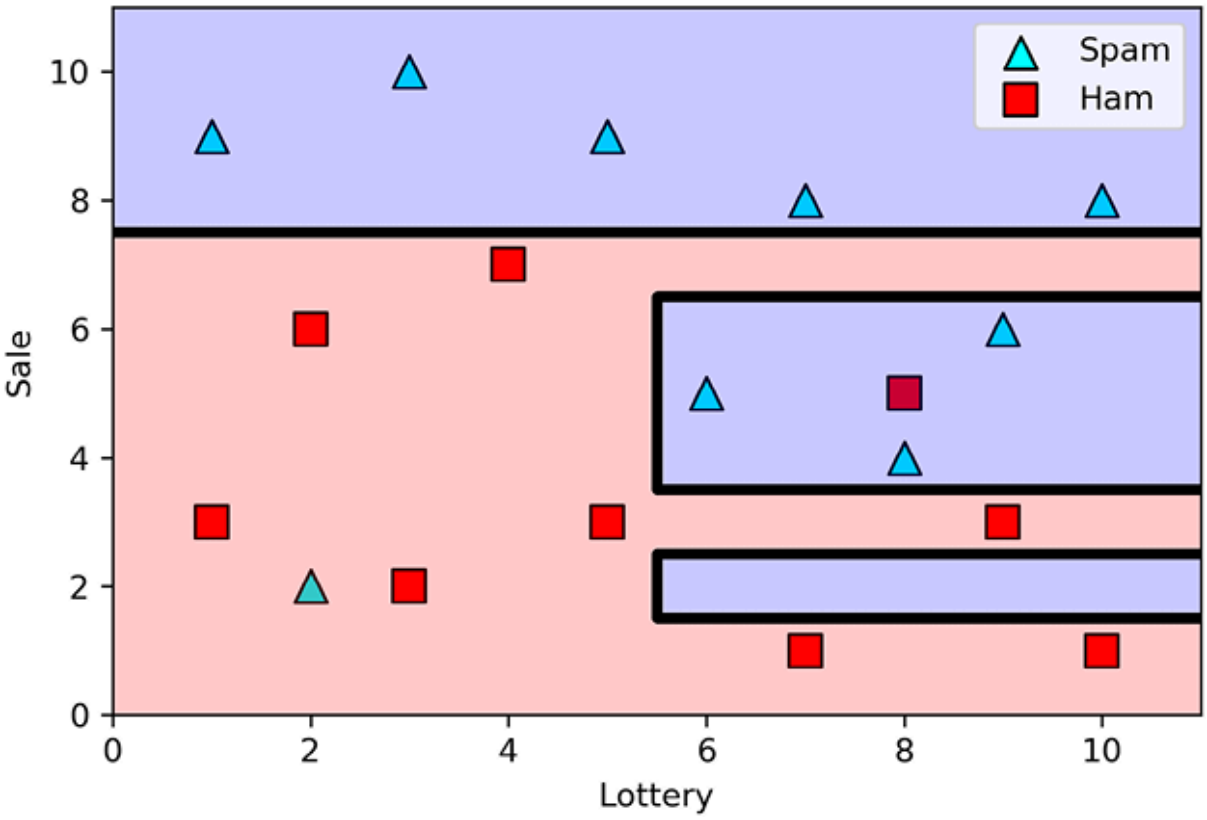


Figure 12.17 The result of the AdaBoost classifier on the spam dataset in figure 12.16. Notice that the classifier does a good job fitting the dataset and doesn't overfit much.

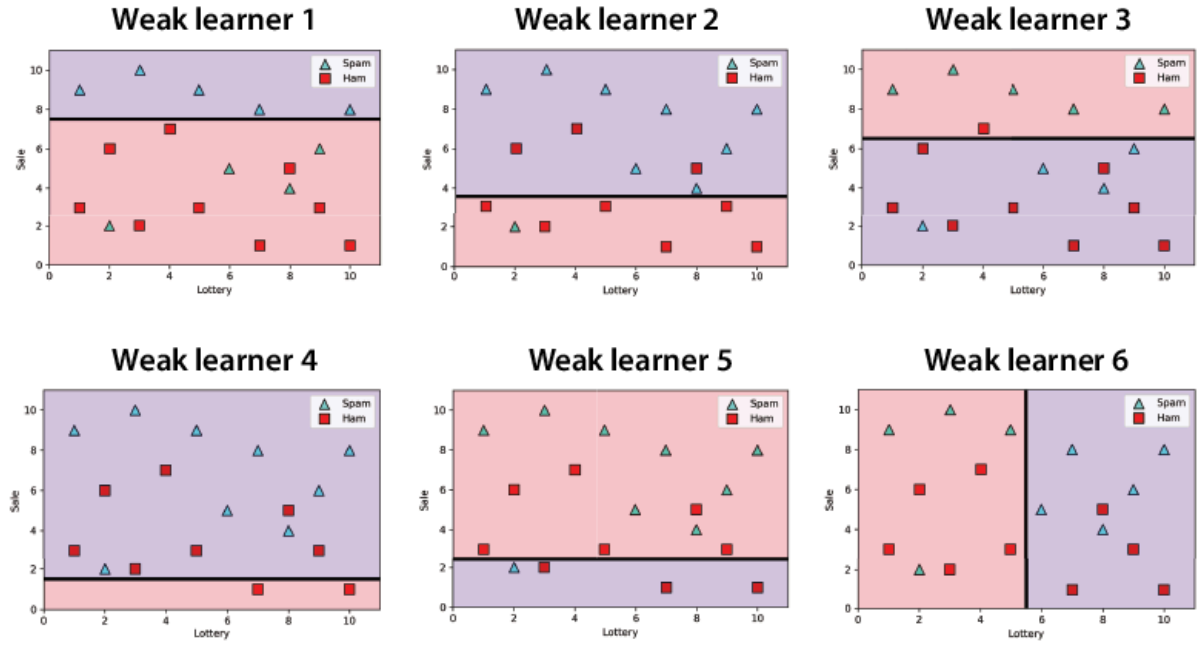


Figure 12.18 The six weak learners in our AdaBoost model. Each one of them is a decision tree of depth 1. They combine into the strong learner in figure 12.17.

Note that the strong learner in figure 12.17 is obtained by assigning a score of 1 to each of the weak learners in figure 12.18 and letting them vote.

Gradient boosting: Using decision trees to build strong learners

In this section, we discuss gradient boosting, one of the most popular and successful machine learning models currently. Gradient boosting is similar to AdaBoost, in that the weak learners are decision trees, and the goal of each weak learner is to learn from the mistakes of the previous ones. One difference between gradient boosting and AdaBoost is that in gradient boosting, we allow decision trees of depth more than 1. Gradient boosting can be used for regression and classification, but for clarity, we use a regression example. To use it for classification, we need to make some small tweaks. To find out more about this, check out links to videos and reading material in appendix C. The code for this section follows:

- **Notebook**: Gradient_boosting_and_XGBoost.ipynb
    - https://github.com/luisguiserrano/manning/blob/master/Chapter_12_Ensemble_Methods/Gradient_boosting_and_XGBoost.ipynb

The example we use is the same one as in the section “Decision trees for regression” in chapter 9, in which we studied the level of engagement of certain users with an app. The feature is the age of the user, and the label is the number of days that the user engages with the app (table 12.3). The plot of the dataset is shown in figure 12.19.

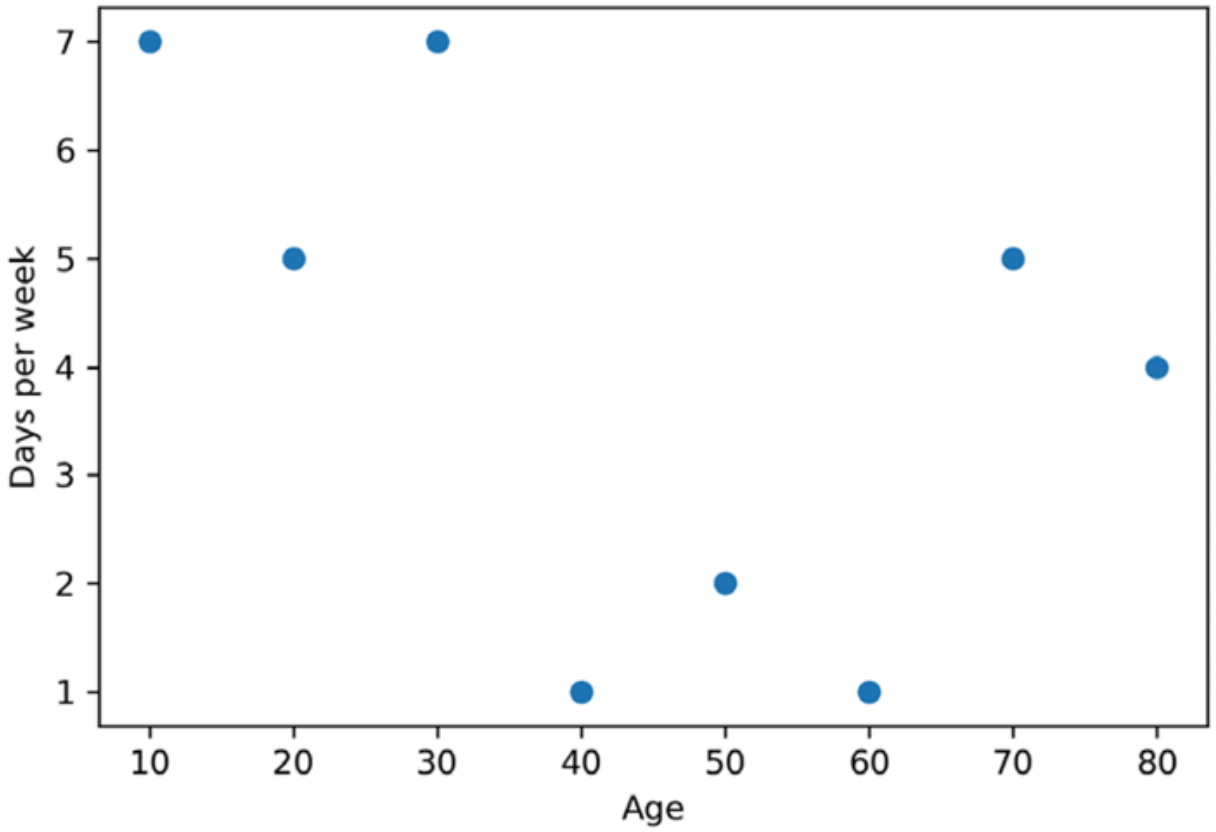


Figure 12.19 The plot of the user engagement dataset from table 12.3. The horizontal axis represents the age of the users, and the vertical axis represents the days per week that the user uses our app.

Table 12.3 A small dataset with eight users, their age, and their engagement with our app. The engagement is measured in the number of days when they opened the app in one week. We'll fit this dataset using gradient boosting.

Feature (age)	Label (engagement)
10	7
20	5
30	7
40	1
50	2
60	1
70	5
80	4

The idea of gradient boosting is that we'll create a sequence of trees that fit this dataset. The two hyperparameters that we'll use for now are the number of trees, which we set to five, and the learning rate, which we set to 0.8. The first weak learner is simple: it is the decision tree of depth 0 that best fits the dataset. A decision tree of depth 0 is simply a node that assigns the same label to each point in the dataset. Because the error function we are minimizing is the mean square error, then this optimal value for the prediction is the average value of the labels. The average value of the labels of this dataset is 4, so our first weak learner is a node that assigns a prediction of 4 to every point.

The next step is to calculate the residual, which is the difference between the label and the prediction made by this first weak learner, and fit a new decision tree to these residuals. As you can see, what this is doing is training a decision tree to fill in the gaps that the first tree has left. The labels, predictions, and residuals are shown in table 12.4.

The second weak learner is a tree that fits these residuals. The tree can be as deep as we'd like, but for this example, we'll make sure all the weak learners are of depth at most 2. This tree is shown in figure 12.20 (together with its boundary), and its predictions are in the rightmost column of table 12.4. This tree has been obtained using Scikit-Learn; see the notebook for the procedure.

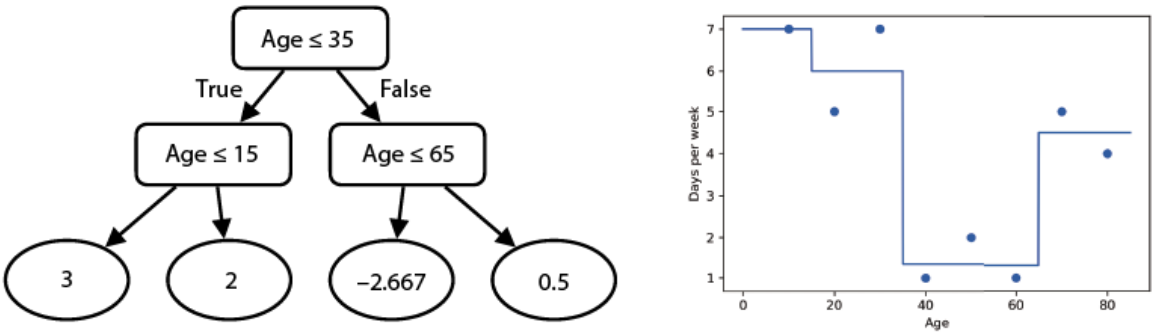


Figure 12.20 The second weak learner in the gradient boosting model. This learner is a decision tree of depth 2 pictured on the left. The predictions of this weak learner are shown on the plot on the right.

Table 12.4 The predictions from the first weak learner are the average of the labels. The second weak learner is trained to fit the residuals of the first weak learner.

Feature (age)	Label (engagement)	Prediction from weak learner 1	Residual	Prediction from weak learner 2
10	7	4	3	3
20	5	4	2	2
30	7	4	3	2
40	1	4	–3	–2.667
50	2	4	–2	–2.667

60	1	4	–3	–2.667
70	5	4	1	0.5
80	4	4	0	0.5

The idea is to continue in this fashion, calculating new residuals and training a new weak learner to fit these residuals. However, there's a small caveat—to calculate the prediction from the first two weak learners, we first multiply the prediction of the second weak learner by the learning rate. Recall that the learning rate we're using is 0.8. Thus, the combined prediction of the first two weak learners is the prediction of the first one (4) plus 0.8 times the prediction of the second one. We do this because we don't want to overfit by fitting our training data too well. Our goal is to mimic the gradient descent algorithm, by slowly walking closer and closer to the solution, and this is what we achieve by multiplying the prediction by the learning rate. The new residuals are the original labels minus the combined predictions of the first two weak learners. These are calculated in table 12.5.

Table 12.5 The labels, the predictions from the first two weak learners, and the residual. The prediction from the first weak learner is the average of the labels. The prediction from the second weak learner is shown in figure 12.20. The combined prediction is equal to the prediction of the first weak learner plus the learning rate (0.8) times the prediction of the second weak learner. The residual is the difference between the label and the combined prediction from the first two weak learners.

Label	Prediction from weak learner 1	Prediction from weak learner 2	Prediction from weak learner 2 times the learning rate	Prediction from weak learners 1 and 2	Residual
7	4	3	2.4	6.4	0.6
5	4	2	1.6	5.6	–0.6
7	4	2	1.6	5.6	1.4
1	4	–2.667	–2.13	1.87	–0.87
2	4	–2.667	–2.13	1.87	0.13
1	4	–2.667	–2.13	1.87	–0.87
5	4	0.5	0.4	4.4	0.6
4	4	0.5	0.4	4.4	–0.4

Now we can proceed to fit a new weak learner on the new residuals and calculate the combined prediction of the first two weak learners. We obtain this by adding the prediction for the first weak learner and 0.8 (the learning rate) times the sum of the predictions of the second and the third weak learner. We repeat this process for every weak learner we want to build. Instead of doing it by hand,

we can use the `GradientBoostingRegressor` package in Scikit-Learn (the code is in the notebook). The next few lines of code show how to fit the model and make predictions. Note that we have set the depth of the trees to be at most 2, the number of trees to be five, and the learning rate to be 0.8. The hyperparameters used for this are `max_depth`, `n_estimators`, and `learning_rate`. Note, too, that if we want five trees, we must set the `n_estimators` hyperparameter to four, because the first tree isn't counted.

```
from sklearn.ensemble import GradientBoostingRegressor
gradient_boosting_regressor = GradientBoostingRegressor(max_depth=2, n_estimators=4,
learning_rate=0.8)
gradient_boosting_regressor.fit(features, labels)
gradient_boosting_regressor.predict(features)
```

The plot for the resulting strong learner is shown in figure 12.21. Notice that it does a good job predicting the values.

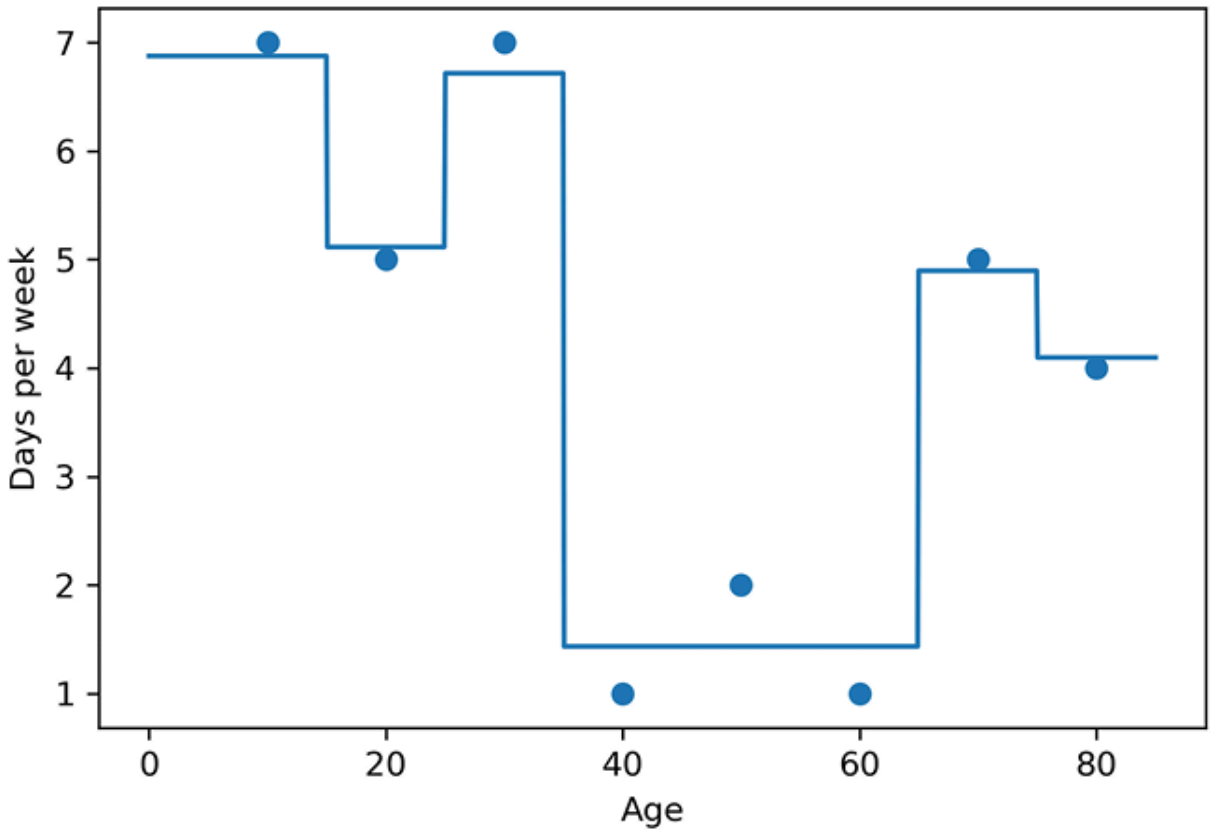


Figure 12.21 The plot of the predictions of the strong learner in our gradient boosting regressor. Note that the model fits the dataset quite well.

However, we can go a little further and actually plot the five weak learners we obtain. The details for this are in the notebook, and the five weak learners are shown in figure 12.22. Notice that the predictions of the last weak learners are much smaller than those of the first ones, because each weak learner is predicting the error of the previous ones, and these errors get smaller and smaller at each step.

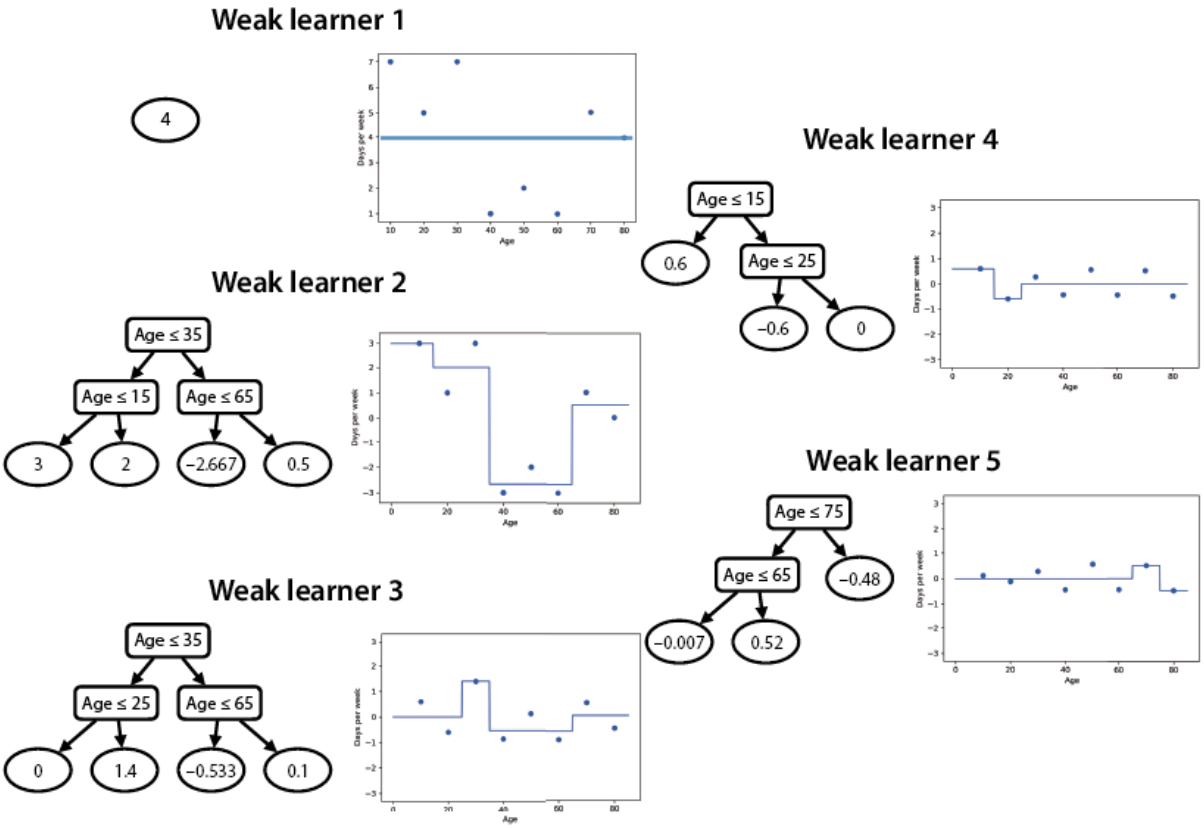


Figure 12.22 The five weak learners in the gradient boosting model. The first one is a decision tree of depth 0 that always predicts the average of the labels. Each successive weak learner is a decision tree of depth at most 2, which fits the residuals from the prediction given by the previous weak learners. Note that the predictions of the weak learners get smaller, because the residuals get smaller when the predictions of the strong learner get closer to the labels.

Finally, we can use Scikit-Learn or a manual calculation to see that the predictions are the following:

- Age = 10, prediction = 6.87
- Age = 20, prediction = 5.11
- Age = 30, prediction = 6.71
- Age = 40, prediction = 1.43
- Age = 50, prediction = 1.43
- Age = 60, prediction = 1.43
- Age = 70, prediction = 4.90
- Age = 80, prediction = 4.10

XGBoost: An extreme way to do gradient boosting

XGBoost, which stands for *extreme gradient boosting*, is one of the most popular, powerful, and effective gradient boosting implementations. Created by Tianqi Chen and Carlos Guestrin in 2016 (see appendix C for the reference), XGBoost models often outperform other classification and regression models. In this section, we discuss how XGBoost works, using the same regression example from the section “Gradient boosting: Using decision trees to build strong learners.”

XGBoost uses decision trees as the weak learners, and just like in the previous boosting methods we learned, each weak learner is designed to focus on the weaknesses of the previous ones. More specifically, each tree is built to fit the residuals of the predictions of the previous trees. However, there are some small differences, such as the way we build the trees, which is using a metric called the *similarity score*. Furthermore, we add a pruning step to prevent overfitting, in which we remove the branches of the trees if they don't satisfy certain conditions. In this section, we cover this in more detail.

XGBoost similarity score: A new and effective way to measure similarity in a set

In this subsection, we see the main building block of XGBoost, which is a way to measure how similar the elements of a set are. This metric is aptly called the *similarity score*. Before we learn it, let's do a small exercise. Among the following three sets, which one has the most amount of similarity, and which one has the least?

- **Set 1**: {10, –10, 4}
- **Set 2**: {7, 7, 7}
- **Set 3**: {7}

If you said that set 2 has the most amount of similarity and set 1 has the least amount, your intuition is correct. In set 1, the elements are very different from each other, so this one has the least amount of similarity. Between sets 2 and 3, it's not so clear, because both sets have the same element, but a different number of times. However, set 2 has the number seven appearing three times, whereas set 3 has it appearing only once. Therefore, in set 2, the elements are more homogeneous, or more similar, than in set 3.

To quantify similarity, consider the following metric. Given a set $\{a_1, a_2, \ldots, a_n\}$, the similarity score is the square of the sum of the elements, divided by the number of elements, namely, $\frac{(a_1 + a_2 + \cdots + a_n)^2}{n}$. Let's calculate the similarity score for the three sets above, shown next:

- Set 1: $\textit{Similarity score} = \frac{(10 - 10 + 4)^2}{3} = 5.33$
- Set 2: $\textit{Similarity score} = \frac{(7 + 7 + 7)^2}{3} = 147$
- Set 3: $\textit{Similarity score} = \frac{7^2}{1} = 49$

Note that as expected, the similarity score of set 2 is the highest, and that of set 1 is the lowest.

note This similarity score is not perfect. One can argue that the set {1, 1, 1} is more similar than the set {7, 8, 9}, yet the similarity score of {1, 1, 1} is 3, and the similarity score of {7, 8, 9} is 192. However, for the purposes of our algorithm, this score still works. The main goal of the similarity score is to be able to separate the large and small values well, and this goal is met, as we'll see in the current example.

There is a hyperparameter λ associated with the similarity score, which helps prevent overfitting. When used, it is added to the denominator of the similarity score, which gives the formula $\frac{(a_1 + a_2 + \cdots + a_n)^2}{n + \lambda}$. Thus, for example, if λ = 2, the similarity score of set 1 is now $\frac{(10-10+4)^2}{3+2} = 3.2$. We won't use the λ hyperparameter in our example, but when we get to the code, we'll see how to set it to any value we want.

Building the weak learners

In this subsection, we see how to build each one of the weak learners. To illustrate this process, we use the same example from the section "Gradient boosting," shown in table 12.3. For convenience, the same dataset is shown in the two leftmost columns of table 12.6. This is a dataset of users of an app, in which the feature is the age of the users, and the label is the number of days per week in which they engage with the app. The plot of this dataset is shown in figure 12.19.

Table 12.6 The same dataset as in table 12.3, containing users, their age, and the number of days per week in which they engaged with our app. The third column contains the predictions from the first weak learner in our XGBoost model. These predictions are all 0.5 by default. The last column contains the residual, which is the difference between the label and the prediction.

Feature (age)	Label (engagement)	Prediction from the first weak learner	Residual
10	7	0.5	6.5
20	5	0.5	4.5
30	7	0.5	6.5
40	1	0.5	0.5
50	2	0.5	1.5
60	1	0.5	0.5
70	5	0.5	4.5
80	4	0.5	3.5

The process of training an XGBoost model is similar to that of training gradient boosting trees. The first weak learner is a tree that gives a prediction of 0.5 to each data point. After building this weak learner, we calculate the residuals, which are the differences between the label and the predicted label. These two quantities can be found in the two rightmost columns of table 12.6.

Before we start building the remaining trees, let's decide how deep we want them to be. To keep this example small, let's again use a maximum depth of 2. That means that when we get to depth 2, we

stop building the weak learners. This is a hyperparameter, which we'll see in more detail in the section "Training an XGBoost model in Python."

To build the second weak learner, we need to fit a decision tree to the residuals. We do this using the similarity score. As usual, in the root node, we have the entire dataset. Thus, we begin by calculating the similarity score of the entire dataset as follows:

$$Similarity = \frac{(6.5+4.5+6.5+0.5+1.5+0.5+4.5+3.5)^2}{8} = 98$$

Now, we proceed to split the node using the age feature in all the possible ways, as we did with decision trees. For each split, we calculate the similarity score of the subsets corresponding to each of the leaves and add them. That is the combined similarity score corresponding to that split. The scores are the following:

Split for the root node, with dataset {6.5, 4.5, 6.5, 0.5, 1.5, 0.5, 4.5, 3.5}, and similarity score = 98:

- Split at 15:
    - Left node: {6.5}; similarity score: 42.25
    - Right node: {4.5, 6.5, 0.5, 1.5, 0.5, 4.5, 3.5}; similarity score: 66.04
    - Combined similarity score: 108.29
- Split at 25:
    - Left node: {6.5, 4.5}; similarity score: 60.5
    - Right node: {6.5, 0.5, 1.5, 0.5, 4.5, 3.5}; similarity score: 48.17
    - Combined similarity score: 108.67
- Split at 35:
    - Left node: {6.5, 4.5, 6.5}; similarity score: 102.08
    - Right node: {0.5, 1.5, 0.5, 4.5, 3.5}; similarity score: 22.05
    - **Combined similarity score: 124.13**
- Split at 45:
    - Left node: {6.5, 4.5, 6.5, 0.5}; similarity score: 81
    - Right node: {1.5, 0.5, 4.5, 3.5}; similarity score: 25
    - Combined similarity score: 106
- Split at 55:
    - Left node: {6.5, 4.5, 6.5, 0.5, 1.5}; similarity score: 76.05
    - Right node: {0.5, 4.5, 3.5}; similarity score: 24.08
    - Combined similarity score: 100.13
- Split at 65:
    - Left node: {6.5, 4.5, 6.5, 0.5, 1.5, 0.5}; similarity score: 66.67
    - Right node: {4.5, 3.5}; similarity score: 32
    - Combined similarity score: 98.67
- Split at 75:
    - Left node: {6.5, 4.5, 6.5, 0.5, 1.5, 0.5, 4.5}; similarity score: 85.75
    - Right node: {3.5}; similarity score: 12.25
    - Combined similarity score: 98

As shown in these calculations, the split with the best combined similarity score is at age = 35. This is going to be the split at the root node.

Next, we proceed to split the datasets at each of the nodes in the same way.

Split for the left node, with dataset {6.5, 4.5, 6.5} and similarity score 102.08:

- Split at 15:
    - Left node: {6.5}; similarity score: 42.25
    - Right node: {4.5, 6.5}; similarity score: 60.5
    - Similarity score: 102.75
- Split at 25:
    - Left node: {6.5, 4.5}; similarity score: 60.5
    - Right node: {6.5}; similarity score: 42.25
    - Similarity score: 102.75

Both splits give us the same combined similarity score, so we can use any of the two. Let's use the split at 15. Now, on to the right node.

Split for the right node, with dataset {0.5, 1.5, 0.5, 4.5, 3.5} and similarity score 22.05:

- Split at 45:
    - Left node: {0.5}; similarity score: 0.25
    - Right node: {1.5, 0.5, 4.5, 3.5}; similarity score: 25
    - Similarity score: 25.25
- Split at 55:
    - Left node: {0.5, 1.5}; similarity score: 2
    - Right node: {0.5, 4.5, 3.5}; similarity score: 24.08
    - Similarity score: 26.08
- Split at 65:
    - Left node: {0.5, 1.5, 0.5}; similarity score: 2.08
    - Right node: {4.5, 3.5}; similarity score: 32
    - **Similarity score: 34.08**
- Split at 75:
    - Left node: {0.5, 1.5, 0.5, 4.5}; similarity score: 12.25
    - Right node: {3.5}; similarity score: 12.25
    - Similarity score: 24.5

From here, we conclude that the best split is at age = 65. The tree now has depth 2, so we stop growing it, because this is what we decided at the beginning of the algorithm. The resulting tree, together with the similarity scores at the nodes, is shown in figure 12.23.

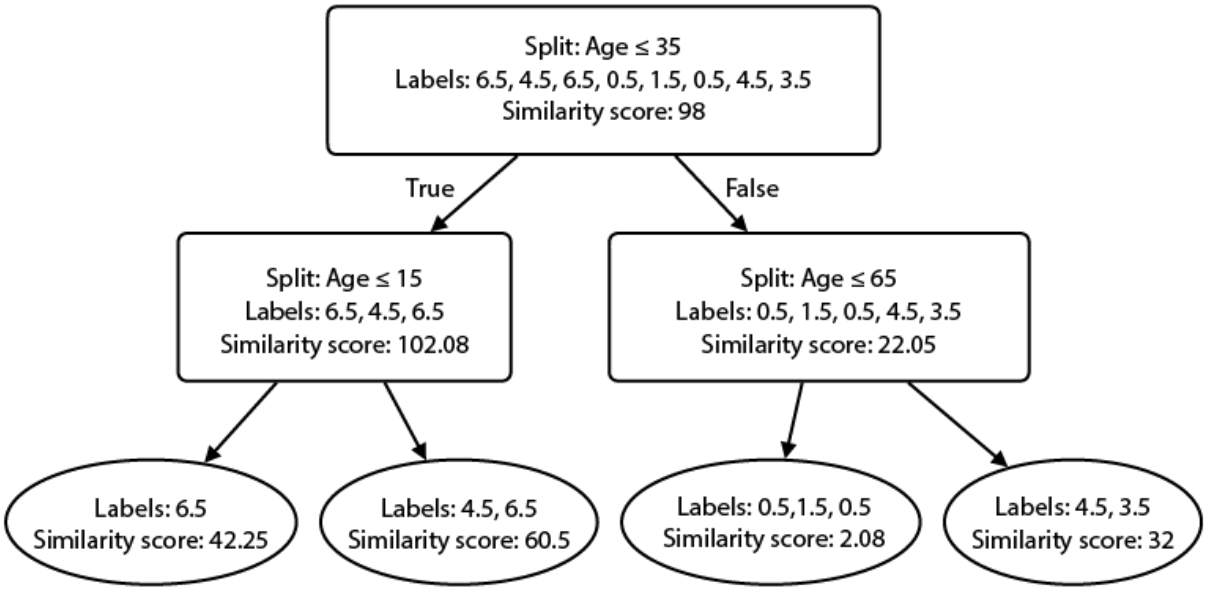


Figure 12.23 The second weak learner in our XGBoost classifier. For each of the nodes, we can see the split based on the age feature, the labels corresponding to that node, and the similarity score for each set of labels. The split chosen for each node is the one that maximizes the combined similarity score of the leaves. For each of the leaves, you can see the corresponding labels and their similarity score.

That's (almost) our second weak learner. Before we continue building more weak learners, we need to do one more step to help reduce overfitting.

Tree pruning: A way to reduce overfitting by simplifying the weak learners

A great feature of XGBoost is that it doesn't overfit much. For this, it uses several hyperparameters that are described in detail in the section "Training an XGBoost model in Python." One of them, the minimum split loss, prevents a split from happening if the combined similarity scores of the resulting nodes are not significantly larger than the similarity score of the original node. This difference is called the similarity gain. For example, in the root node of our tree, the similarity score is 98, and the combined similarity score of the nodes is 124.13. Thus, the similarity gain is 124.13 – 98 = 26.13. Similarly, the similarity gain of the left node is 0.67, and that of the right node is 12.03, as shown in figure 12.24.

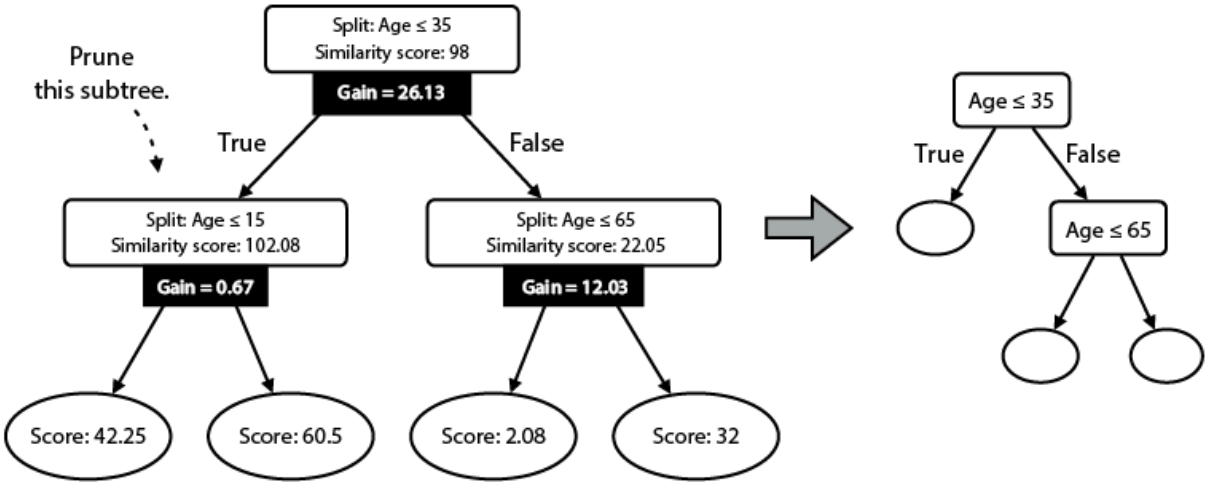


Figure 12.24 On the left, we have the same tree from figure 12.23, with an extra piece of information: the similarity gain. We obtain this by subtracting the similarity score for each node from the combined similarity score of the leaves. We only allow splits with a similarity gain higher than 1 (our minimum split loss hyperparameter), so one of the splits is no longer permitted. This results in the pruned tree on the right, which now becomes our weak learner.

We'll set the minimum split loss to 1. With this value, the only split that is prevented is the one on the left node (age ≤ 15). Thus, the second weak learner looks like the one on the right of figure 12.24.

Making the predictions

Now that we've built our second weak learner, it's time to use it to make predictions. We obtain predictions the same way we obtain them from any decision tree, namely, by averaging the labels in the corresponding leaf. The predictions for our second weak learner are seen in figure 12.25.

Now, on to calculate the combined prediction for the first two weak learners. To avoid overfitting, we use the same technique that we used in gradient boosting, which is multiplying the

prediction of all the weak learners (except the first one) by the learning rate. This is meant to emulate the gradient descent method, in which we slowly converge to a good prediction after several iterations. We use a learning rate of 0.7. Thus, the combined prediction of the first two weak learners is equal to the prediction of the first weak learner plus the prediction of the second weak learner times 0.7. For example, for the first data point, this prediction is

0.5 + 5.83 . 0.7 = 4.58.

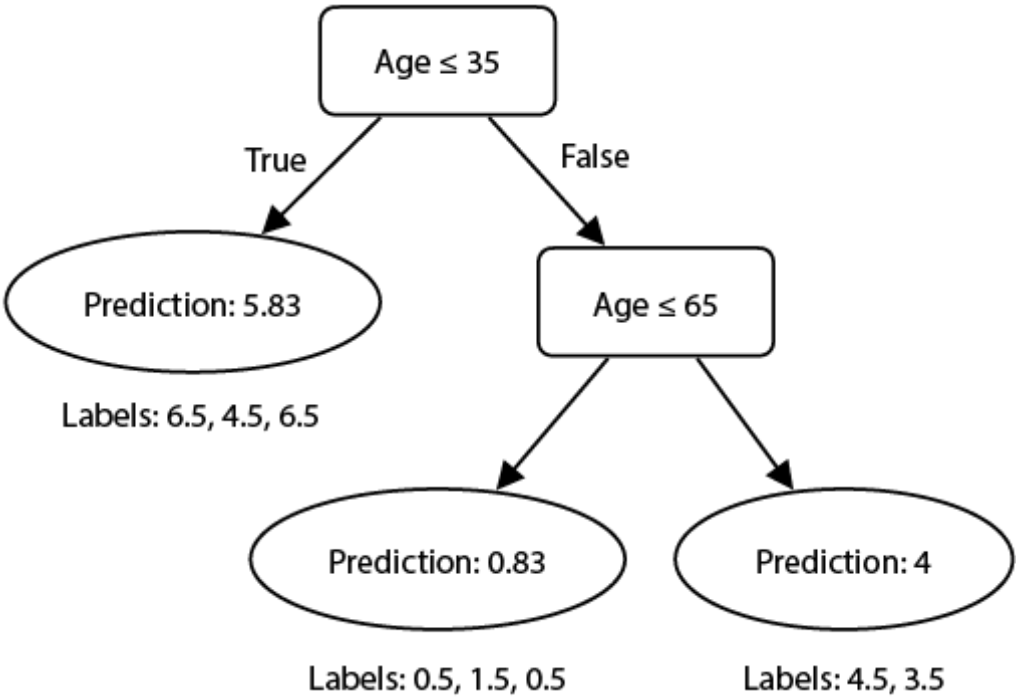


Figure 12.25 The second weak learner in our XGBoost model after being pruned. This is the same tree from figure 12.24, with its predictions. The prediction at each leaf is the average of the labels corresponding to that leaf.

The fifth column of table 12.7 contains the combined prediction of the first two weak learners.

Table 12.7 The labels, predictions from the first two weak learners, and the residual. The combined prediction is obtained by adding the prediction from the first weak learner (which is always 0.5) plus the learning rate (0.7) times the prediction from the second weak learner. The residual is again the difference between the label and the combined prediction.

Label (engagement)	Prediction from weak learner 1	Prediction from weak learner 2	Prediction from weak learner 2 times the learning rate	Combined prediction	Residual
7	0.5	5.83	4.08	4.58	2.42
5	0.5	5.83	4.08	4.58	0.42
7	0.5	5.83	4.08	4.58	2.42
1	0.5	0.83	0.58	1.08	–0.08
2	0.5	0.83	0.58	1.08	0.92
1	0.5	0.83	0.58	1.08	–0.08

5	0.5	4	2.8	3.3	1.7
4	0.5	4	2.8	3.3	0.7

Notice that the combined predictions are closer to the labels than the predictions of the first weak learner. The next step is to iterate. We calculate new residuals for all the data points, fit a tree to them, prune the tree, calculate the new combined predictions, and continue in this fashion. The number of trees we want is another hyperparameter that we can choose at the start. To continue building these trees, we resort to a useful Python package called `xgboost`.

Training an XGBoost model in Python

In this section, we learn how to train the model to fit the current dataset using the `xgboost` Python package. The code for this section is in the same notebook as the previous one, shown here:

- **Notebook**: Gradient_boosting_and_XGBoost.ipynb
    - https://github.com/luisguiserrano/manning/blob/master/Chapter_12_Ensemble_Methods/Gradient_boosting_and_XGBoost.ipynb

Before we start, let's revise the hyperparameters that we've defined for this model:

number of estimators The number of weak learners. Note: in the `xgboost` package, the first weak learner is not counted among the estimators. For this example, we set it to 3, which will give us four weak learners.

maximum depth The maximum depth allowed for each one of the decision trees (weak learners). We set it to 2.

lambda parameter The number added to the denominator of the similarity score. We set it to 0.

minimum split loss The minimum gain in similarity score to allow for a split to happen. We set it to 1.

learning rate The predictions from the second to last weak learners are multiplied by the learning rate. We set it to 0.7.

With the following lines of code, we import the package, build a model called `XGBRegressor`, and fit it to our dataset:

```
import xgboost
from xgboost import XGBRegressor
xgboost_regressor = XGBRegressor(random_state=0,
                                 n_estimators=3,
                                 max_depth=2,
                                 reg_lambda=0,
                                 min_split_loss=1,
                                 learning_rate=0.7)
xgboost_regressor.fit(features, labels)
```

The plot of the model is shown in figure 12.26. Notice that it fits the dataset well.

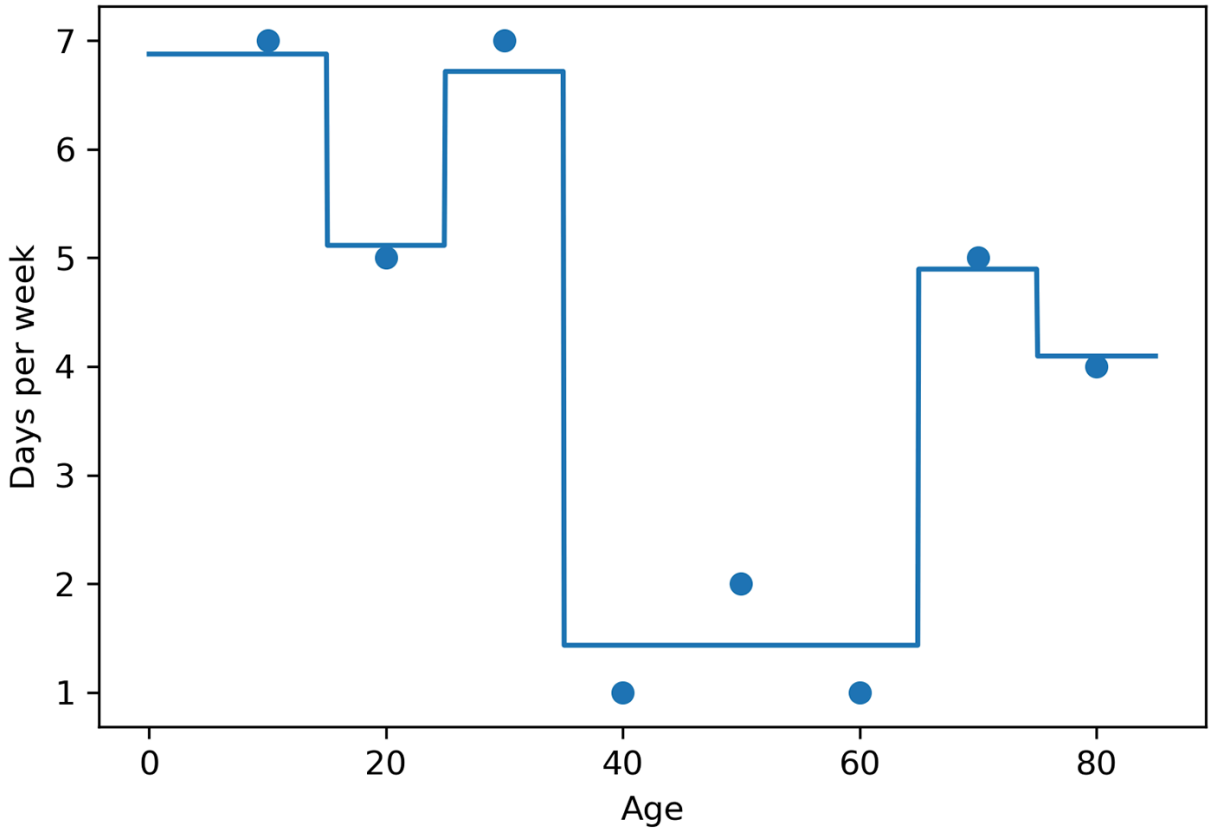


Figure 12.26 The plot of the predictions of our XGBoost model. Note that it fits the dataset well.

The `xgboost` package also allows us to look at the weak learners, and they appear in figure 12.24. The trees obtained in this fashion already have the labels multiplied by the learning rate of 0.7, which is clear when compared with the predictions of the tree obtained manually in figure 12.25 and the second tree from the left in figure 12.27.

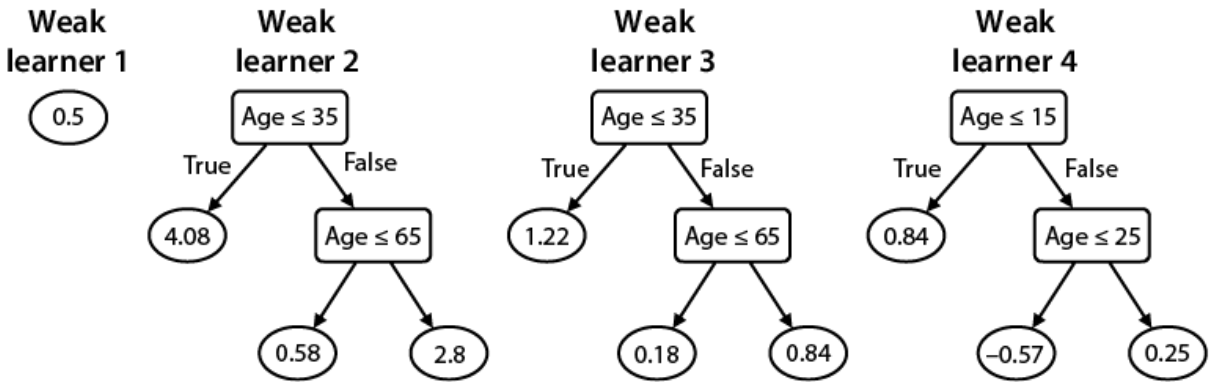


Figure 12.27 The four weak learners that form the strong learner in our XGBoost model. Note that the first one always predicts 0.5. The other three are quite similar in shape, which is a coincidence. However, notice that the predictions from each of the trees get smaller, because each time we are fitting smaller residuals. Furthermore, notice that the second weak learner is the same tree we obtained manually in figure 12.25, where the only difference is that in this tree, the predictions are already multiplied by the learning rate of 0.7.

Thus, to obtain the predictions of the strong learner, we need to add only the prediction of every tree. For example, for a user who is 20 years old, the predictions are the following:

- Weak learner 1: 0.5
- Weak learner 2: 4.08
- Weak learner 3: 1.22
- Weak learner 4: –0.57

Thus, the prediction is 0.5 + 5.83 + 1.22 – 0.57 = 5.23. The predictions for the other points follow:

- Age = 10; prediction = 6.64
- Age = 20; prediction = 5.23
- Age = 30; prediction = 6.05
- Age = 40; prediction = 1.51
- Age = 50; prediction = 1.51
- Age = 60; prediction = 1.51
- Age = 70; prediction = 4.39
- Age = 80; prediction = 4.39

Applications of ensemble methods

Ensemble methods are some of the most useful machine learning techniques used nowadays because they exhibit great levels of performance with relatively low cost. One of the places where ensemble methods are used most is in machine learning challenges, such as the Netflix Challenge. The Netflix Challenge was a competition that Netflix organized, where they anonymized some data and made it public. The competitors' goal was to build a better recommendation system than Netflix itself; the best system would win one million dollars. The winning team used a powerful combination of learners in an ensemble to win. For more information on this, check the reference in appendix C.

Summary

- Ensemble methods consist of training several weak learners and combining them into a strong one. They are an effective way to build powerful models that have had great results with real datasets.
- Ensemble methods can be used for regression and for classification.
- There are two major types of ensemble methods: bagging and boosting.
- Bagging, or bootstrap aggregating, consists of building successive learners on random subsets of our data and combining them into a strong learner that makes predictions based on a majority vote.
- Boosting consists of building a sequence of learners, where each learner focuses on the weaknesses of the previous one, and combining them into a strong classifier that makes predictions based on a weighted vote of the learners.
- AdaBoost, gradient boosting, and XGBoost are three advanced boosting algorithms that produce great results with real datasets.
- Applications of ensemble methods range widely, from recommendation algorithms to applications in medicine and biology.

Exercises

Exercise 12.1

A boosted strong learner L is formed by three weak learners, L_1, L_2, and L_3. Their weights are 1, 0.4, and 1.2, respectively. For a particular point, L_1 and L_2 predict that its label is positive, and L_3 predicts that it's negative. What is the final prediction the learner L makes on this point?

Exercise 12.2

We are in the middle of training an AdaBoost model on a dataset of size 100. The current weak learner classifies 68 out of the 100 data points correctly. What is the weight that we'll assign to this learner in the final model?

Cheat Sheet: Building Supervised Learning Models
Common supervised learning models

Process Name	**Brief Description**	**Code Syntax**
One vs One classifier (using logistic regression)	**Process:** This method trains one classifier for each pair of classes. **Key hyperparameters:** - `estimator`: Base classifier (e.g., logistic regression) **Pros:** Can work well for small datasets. **Cons:** Computationally expensive for large datasets. **Common applications:** Multiclass classification problems where the number of classes is relatively small.	1. from sklearn.multiclass import OneVsOneClassifier 2. from sklearn.linear_model import LogisticRegression 3. model = OneVsOneClassifier(LogisticRegression())

One vs All classifier (using logistic regression)	**Process:** Trains one classifier per class, where each classifier distinguishes between one class and the rest. **Key hyperparameters:** - `estimator`: Base classifier (e.g., Logistic Regression) - `multi_class`: Strategy to handle multiclass classification (`ovr`) **Pros:** Simpler and more scalable than One vs One. **Cons:** Less accurate for highly imbalanced classes. **Common applications:** Common in multiclass classification problems such as image classification.	1. from sklearn.multiclass import OneVsRestClassifier 2. from sklearn.linear_model import LogisticRegression 3. model = OneVsRestClassifier(LogisticRegression()) or 1. from sklearn.linear_model import LogisticRegression 2. model_ova = LogisticRegression(multi_class='ovr')

Decision tree classifier	**Process:** A tree-based classifier that splits data into smaller subsets based on feature values. **Key hyperparameters:** - `max_depth`: Maximum depth of the tree **Pros:** Easy to interpret and visualize. **Cons:** Prone to overfitting if not pruned properly. **Common applications:** Classification tasks, such as credit risk assessment.	1. from sklearn.tree import DecisionTreeClassifier 2. model = DecisionTreeClassifier(max_depth=5)
Decision tree regressor	**Process:** Similar to the decision tree classifier, but used for regression tasks to predict continuous values. **Key hyperparameters:** - `max_depth`: Maximum depth of the tree **Pros:** Easy to interpret, handles nonlinear data. **Cons:** Can overfit and perform poorly on noisy data. **Common applications:** Regression tasks, such as predicting housing prices.	1. from sklearn.tree import DecisionTreeRegressor 2. model = DecisionTreeRegressor(max_depth=5)

Linear SVM classifier	**Process:** A linear classifier that finds the optimal hyperplane separating classes with a maximum margin. **Key hyperparameters:** - `C`: Regularization parameter - `kernel`: Type of kernel function (`linear`, `poly`, `rbf`, etc.) - `gamma`: Kernel coefficient (only for `rbf`, `poly`, etc.) **Pros:** Effective for high-dimensional spaces. **Cons:** Not ideal for nonlinear problems without kernel tricks. **Common applications:** Text classification and image recognition.	1. from sklearn.svm import SVC 2. model = SVC(kernel='linear', C=1.0)

K-nearest neighbors classifier	**Process:** Classifies data based on the majority class of its nearest neighbors. **Key hyperparameters:** - `n_neighbors`: Number of neighbors to use - `weights`: Weight function used in prediction (`uniform` or `distance`) - `algorithm`: Algorithm used to compute the nearest neighbors (`auto`, `ball_tree`, `kd_tree`, `brute`) **Pros:** Simple and effective for small datasets. **Cons:** Computationally expensive as the dataset grows. **Common applications:** Recommendation systems, image recognition.	1. from sklearn.neighbors import KNeighborsClassifier 2. model = KNeighborsClassifier(n_neighbors=5, weights='uniform')

Random Forest regressor	**Process:** An ensemble method using multiple decision trees to improve accuracy and reduce overfitting. **Key hyperparameters:** - `n_estimators`: Number of trees in the forest - `max_depth`: Maximum depth of each tree **Pros:** Less prone to overfitting than individual decision trees. **Cons:** Model complexity increases with the number of trees. **Common applications:** Regression tasks such as predicting sales or stock prices.	1. from sklearn.ensemble import RandomForestRegressor 2. model = RandomForestRegressor(n_estimators=100, max_depth=5)

XGBoost regressor	**Process:** A gradient boosting method that builds trees sequentially to correct errors from previous trees. **Key hyperparameters:** - `n_estimators`: Number of boosting rounds - `learning_rate`: Step size to improve accuracy - `max_depth`: Maximum depth of each tree **Pros:** High accuracy and works well with large datasets. **Cons:** Computationally intensive, complex to tune. **Common applications:** Predictive modeling, especially in Kaggle competitions.	1. import xgboost as xgb 2. model = xgb.XGBRegressor(n_estimators=100, learning_rate=0.1, max_depth=5)

Associated functions used

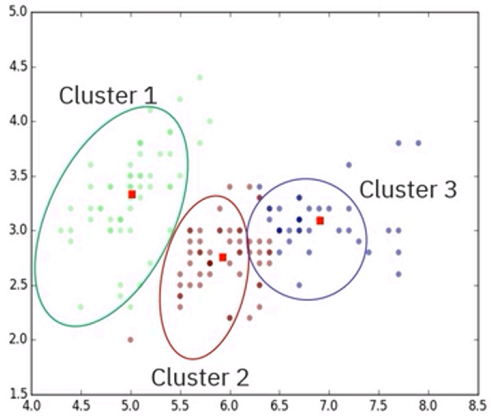
Method Name	Brief Description	Code Syntax
OneHotEncoder	Transforms categorical features into a one-hot encoded matrix.	1. from sklearn.preprocessing import OneHotEncoder 2. encoder = OneHotEncoder(sparse=False) 3. encoded_data = encoder.fit_transform(categorical_data)

accuracy_score	Computes the accuracy of a classifier by comparing predicted and true labels.	1. from sklearn.metrics import accuracy_score 2. accuracy = accuracy_score(y_true, y_pred)
LabelEncoder	Encodes labels (target variable) into numeric format.	1. from sklearn.preprocessing import LabelEncoder 2. encoder = LabelEncoder() 3. encoded_labels = encoder.fit_transform(labels)
plot_tree	Plots a decision tree model for visualization.	1. from sklearn.tree import plot_tree 2. plot_tree(model, max_depth=3, filled=True)

normalize	Scales each feature to have zero mean and unit variance (standardization).	1. from sklearn.preprocessing import normalize 2. normalized_data = normalize(data, norm='l2')
compute_sample_weight	Computes sample weights for imbalanced datasets.	1. from sklearn.utils.class_weight import compute_sample_weight 2. weights = compute_sample_weight(class_weight='balanced', y=y)
roc_auc_score	Computes the Area Under the Receiver Operating Characteristic Curve (AUC-ROC) for binary classification models.	1. from sklearn.metrics import roc_auc_score 2. auc = roc_auc_score(y_true, y_score)

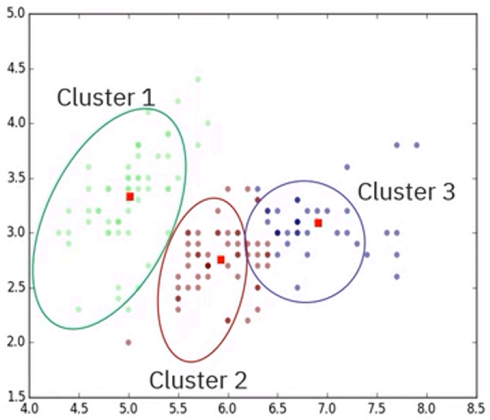
Clustering.

What is clustering?



- Clustering
  - Machine learning technique
  - Automatically groups data points based on similarities

What is clustering?



- Applications
  - Identifying music genres
  - Segmenting user groups
  - Analyzing market segments

Clustering and classification

Labeled dataset

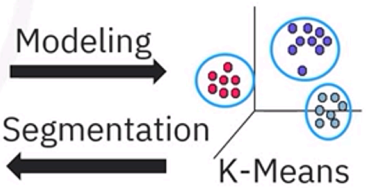
Customer Id	Age	Edu	Years Employed	Income	Card Debt	Other Debt	Address	DebtIncomeRatio	Defaulted
1	41	2	6	19	0.124	1.073	NBA001	6.3	0
2	47	1	26	100	4.582	8.218	NBA021	12.8	0
3	33	2	10	57	6.111	5.802	NBA013	20.9	1
4	29	2	4	19	0.681	0.516	NBA009	6.3	0
5	47	1	31	253	9.308	8.908	NBA008	7.2	0
6	40	1	23	81	0.998	7.831	NBA016	10.9	1
7	38	2	4	56	0.442	0.454	NBA013	1.6	0
8	42	3	0	64	0.279	3.945	NBA009	6.6	0
9	26	1	5	18	0.575	2.215	NBA006	15.5	1

K-means model

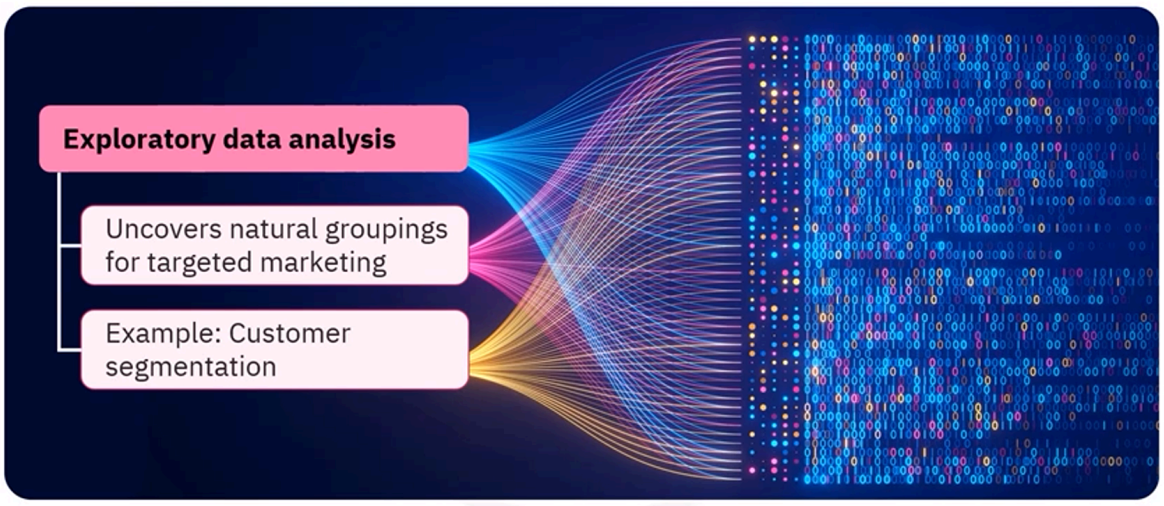
- Segments customers by characteristics
- Identifies three distinct customer clusters

Unlabeled dataset

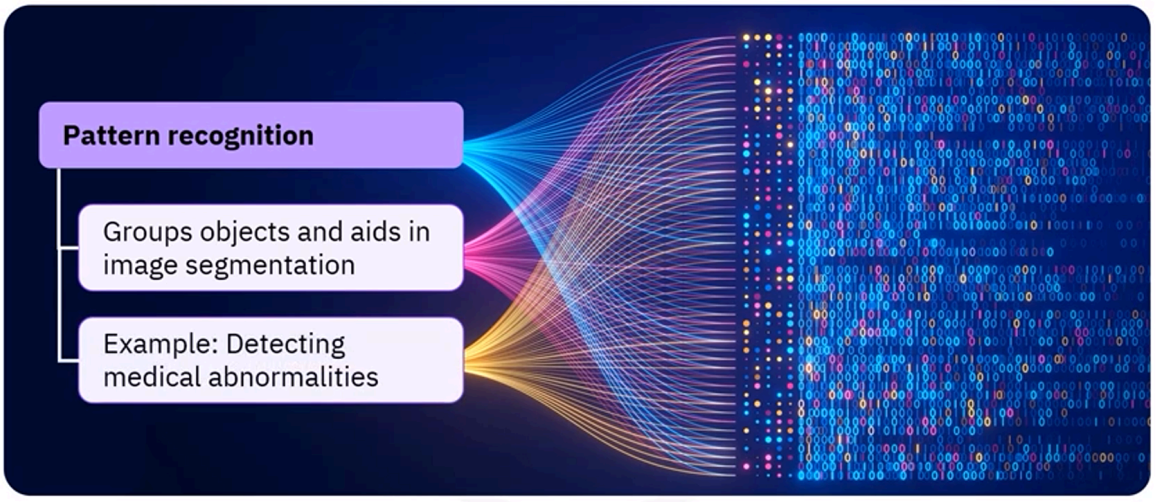
Customer Id	Age	Edu	Years Employed	Income	Card Debt	Other Debt	Address	DebtIncomeRatio
1	41	2	6	19	0.124	1.073	NBA001	6.3
2	47	1	26	100	4.582	8.218	NBA021	12.8
3	33	2	10	57	6.111	5.802	NBA013	20.9
4	29	2	4	19	0.681	0.516	NBA009	6.3
5	47	1	31	253	9.308	8.908	NBA008	7.2
6	40	1	23	81	0.998	7.831	NBA016	10.9
7	38	2	4	56	0.442	0.454	NBA013	1.6
8	42	3	0	64	0.279	3.945	NBA009	6.6
9	26	1	5	18	0.575	2.215	NBA006	15.5



Common applications of clustering

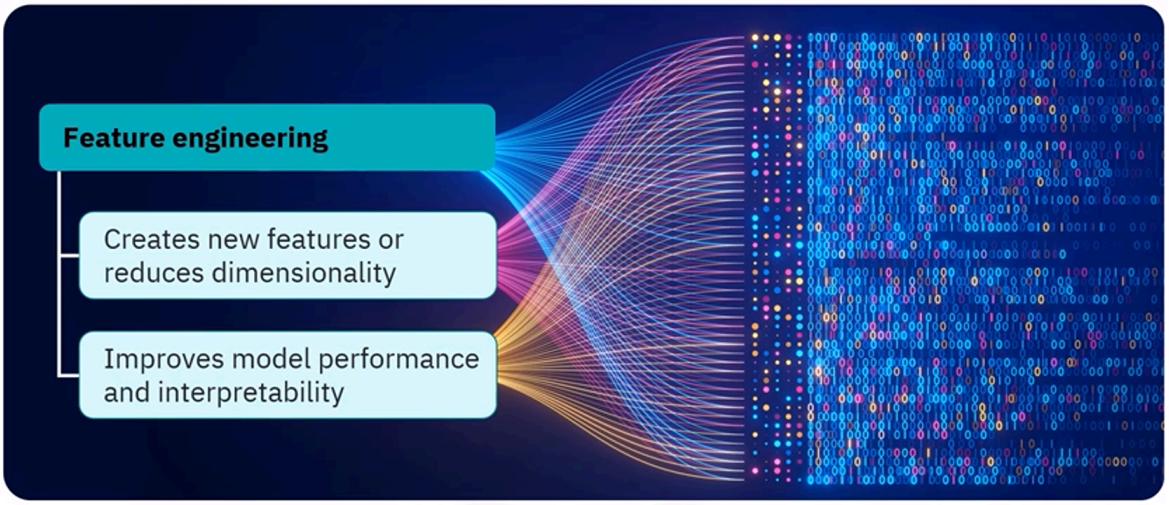


Common applications of clustering

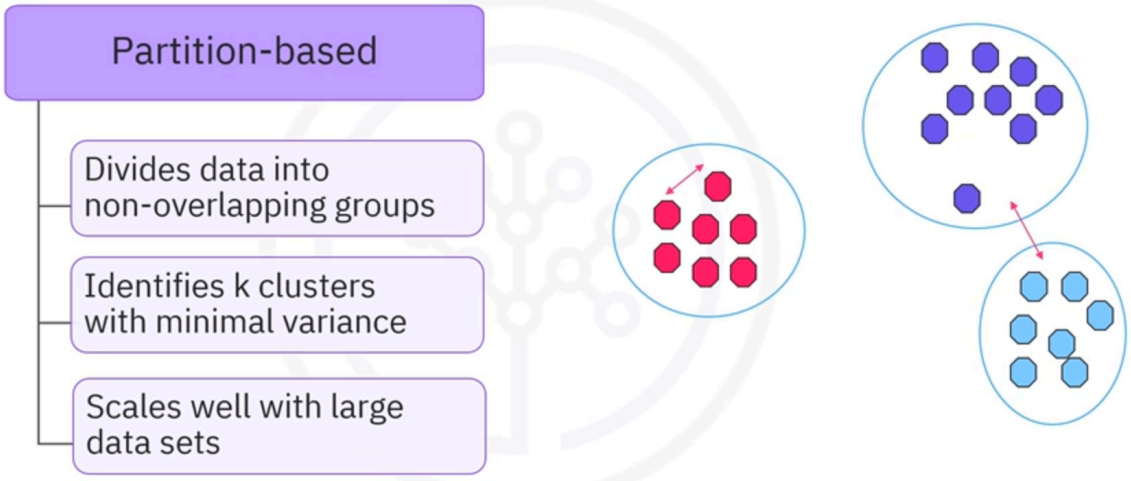


Common applications of clustering

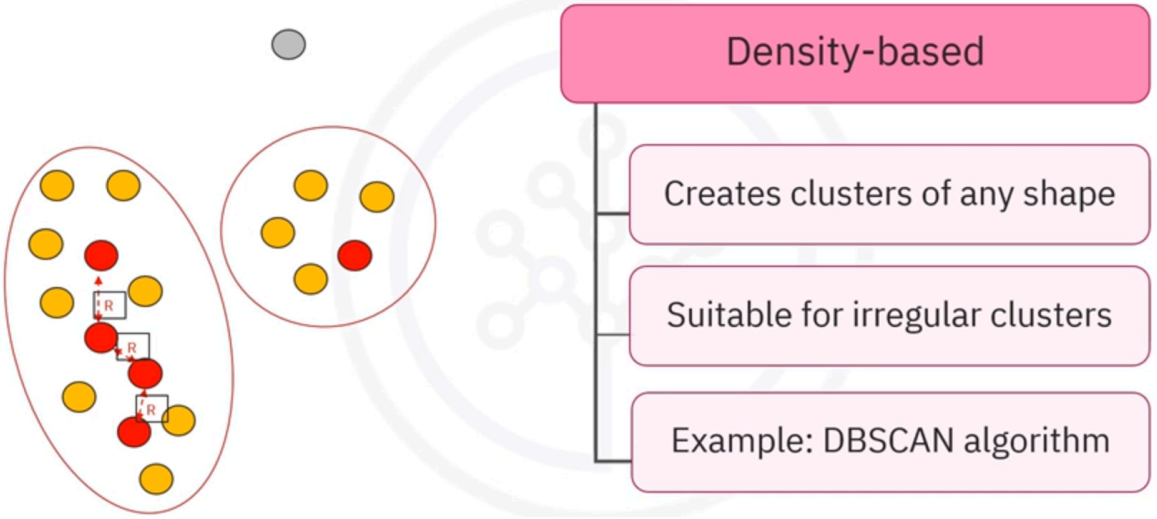
Common applications of clustering



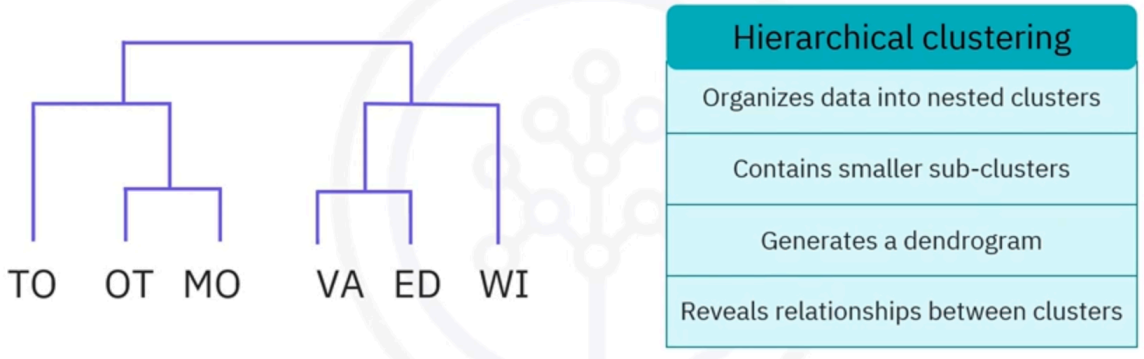
Types of clustering



Types of clustering

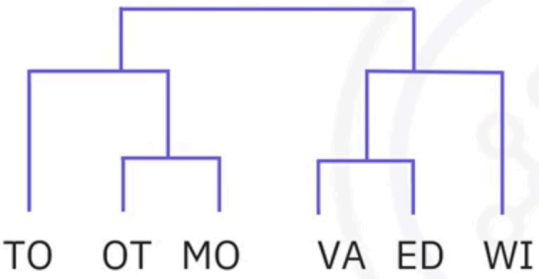


Types of clustering



Two main algorithms for hierarchical clustering:

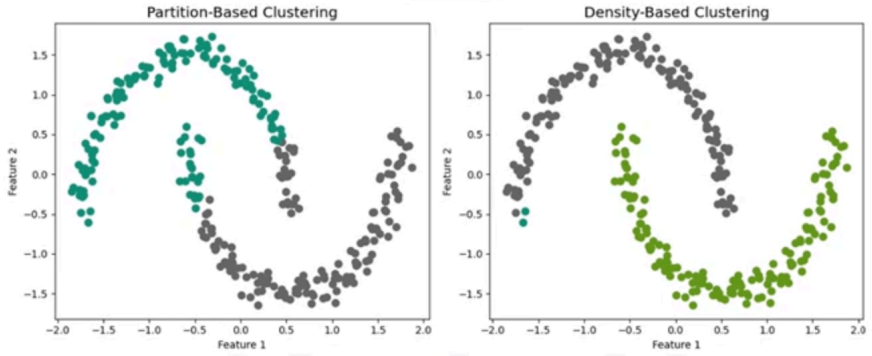
Types of clustering



Agglomerative: Merges clusters

Divisive: Splits clusters

Partition and density-based clustering



Uses the "make moons" function

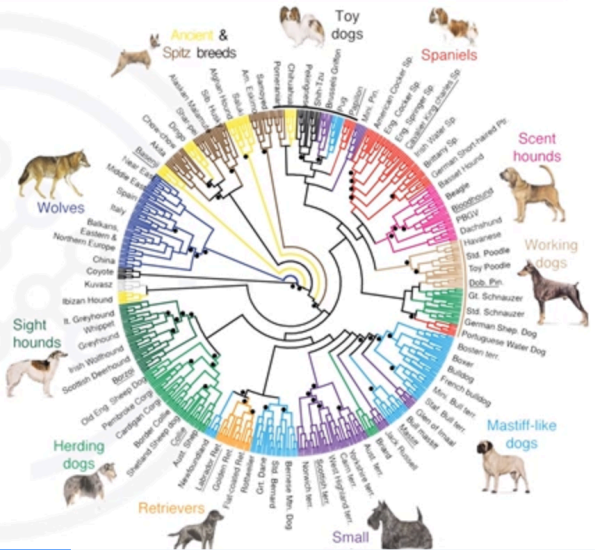
Generates interlocking half-circles

Distinguishes clusters using color

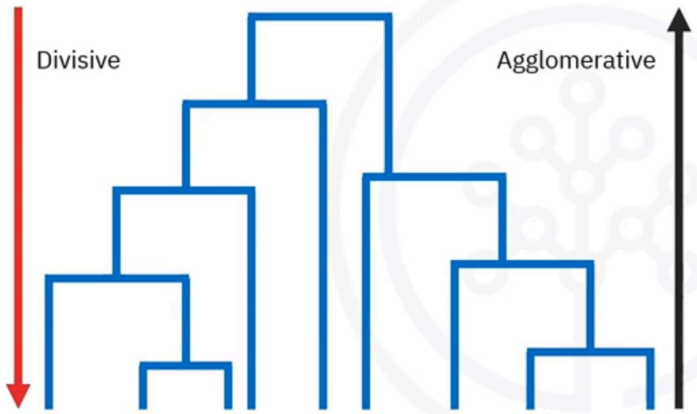
Hierarchical clustering

Groups animals based on genetic similarities

Displays tree-like structure of nested clusters



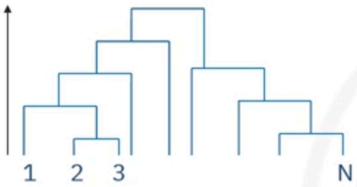
Hierarchical clustering



Divisive clustering

Uses a top-down approach

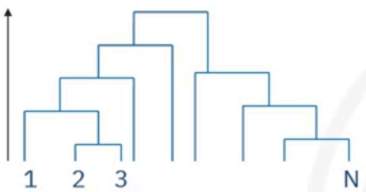
Agglomerative hierarchical clustering



$$\begin{bmatrix} 0 & & & & \\ d(2,1) & 0 & & & \\ d(3,1) & d(3,2) & 0 & & \\ \vdots & \vdots & \vdots & & \\ d(n,1) & d(n,2) & \dots & \dots & 0 \end{bmatrix}$$

- Initialize N clusters with single data points
- Compute a distance matrix for clusters
- Display distances between each pair of points

Agglomerative hierarchical clustering



$$\begin{bmatrix} 0 & & & & \\ d(2,1) & 0 & & & \\ d(3,1) & d(3,2) & 0 & & \\ \vdots & \vdots & \vdots & & \\ d(n,1) & d(n,2) & \dots & \dots & 0 \end{bmatrix}$$

- Initialize N clusters with single data points
- Compute a distance matrix for clusters
- Display distances between each pair of points

Step-by-Step: Agglomerative Clustering

---

1. Start with each data point as its own cluster

- If you have n data points, you start with **n clusters**.
- Each cluster contains **just one point**.

---

2. Compute the distance (or similarity) between all clusters

- Use a **distance metric**, typically:
    - **Euclidean distance**
    - **Manhattan distance**
    - Or others depending on the problem
- For clusters with more than one point, define inter-cluster distance using a **linkage method**:
    - **Single linkage**: minimum distance between points in the two clusters
    - **Complete linkage**: maximum distance
    - **Average linkage**: average pairwise distance
    - **Ward's method**: minimizes the increase in total within-cluster variance

---

3. Merge the two closest clusters

- Identify the **two clusters** with the **smallest distance**.
- **Merge them** into a single new cluster.

---

4. Update the distance matrix

- Recalculate distances between the **new cluster** and all other existing clusters.

- Use the chosen **linkage method** again for these calculations.

---

5. Repeat steps 3–4 until all points are in a single cluster

- Continue merging until there's **only one big cluster** (the root of the hierarchy).
- This forms a **dendrogram** (tree-like structure).

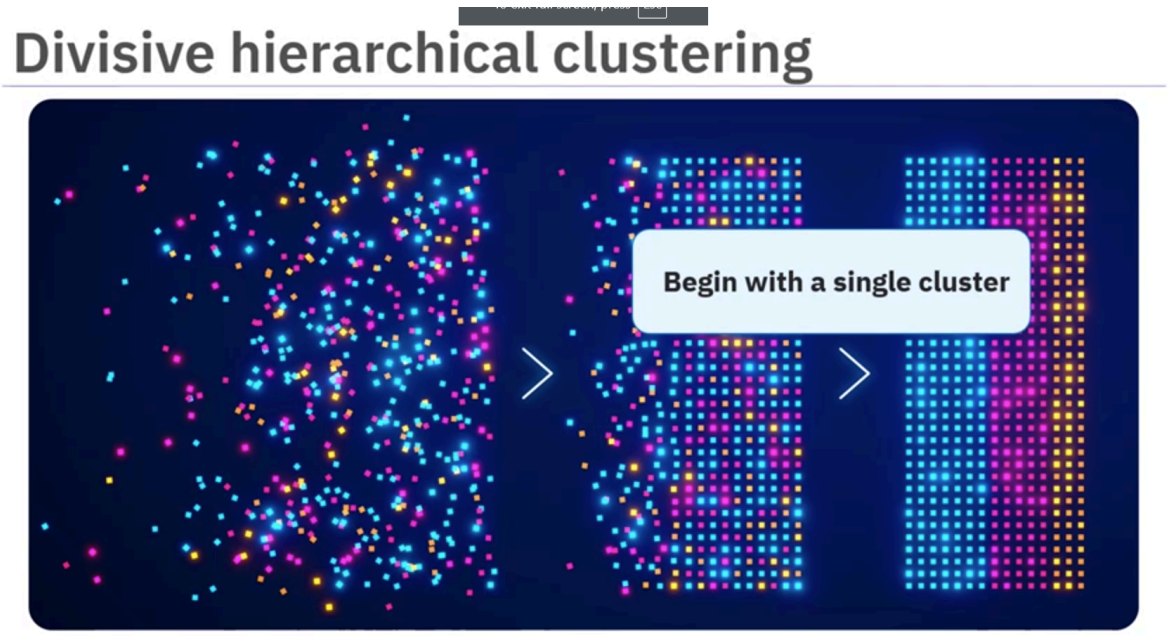
---

6. Choose the number of clusters (optional)

- You can **cut the dendrogram** at a specific level to get the desired number of clusters.
- This is often done visually or based on a distance threshold.

✅ Summary:

> Start with all points as separate clusters → Repeatedly merge the closest clusters → Stop when you have one cluster or the desired number.



1. Begin with a single cluster
2. Consider the entire data set as one cluster.
3. Divide the cluster in single cluster into smaller clusters based on similarities or similarities.

4. Recursive splitting.
5. Split each cluster in two smaller cluster until a stopping criterion.
6. Defined as a specific number of clusters.

Example:

📍 Step 0: What is Agglomerative Clustering?

Agglomerative clustering is a type of **hierarchical clustering** where:

- Each city starts as its own cluster.
- In each step, the **two closest clusters** are **merged**.
- This continues until all cities are merged into a single cluster or until a desired number of clusters is reached.

🗺️ Example Dataset: Cities and Pairwise Distances

We'll use a small subset of cities with simplified distances (in km):

City	London	Paris	Berlin	Rome	Madrid
London	0	344	930	1430	1260
Paris	344	0	878	1105	1054
Berlin	930	878	0	1181	1860
Rome	1430	1105	1181	0	1365
Madrid	1260	1054	1860	1365	0

🧮 Step-by-Step Agglomerative Clustering

We'll use **single linkage** (minimum distance between points in clusters). Here's how the clustering unfolds:

✅ Step 1: Initialization

Each city is its own cluster:

css
CopyEdit

Clusters: {London}, {Paris}, {Berlin}, {Rome}, {Madrid}

🔍 Step 2: Find Closest Pair of Clusters

Check the shortest distance between any two clusters (cities).

- Minimum distance: **London - Paris = 344**

👉 Merge {London} and {Paris}

css
CopyEdit
Clusters: {London, Paris}, {Berlin}, {Rome}, {Madrid}

🔍 Step 3: Update Distances

Now calculate the distance from the new cluster {London, Paris} to all other cities using **single linkage** (minimum distance):

- {London, Paris} to Berlin = min(930, 878) = **878**
- {London, Paris} to Rome = min(1430, 1105) = **1105**
- {London, Paris} to Madrid = min(1260, 1054) = **1054**

🔍 Step 4: Find Closest Clusters Again

Smallest distance is:

- {London, Paris} to Madrid = **1054**

👉 Merge {London, Paris} and {Madrid}

css
CopyEdit
Clusters: {London, Paris, Madrid}, {Berlin}, {Rome}

🔍 Step 5: Update Distances Again

From {London, Paris, Madrid} to:

- Berlin = min(930, 878, 1860) = **878**
- Rome = min(1430, 1105, 1365) = **1105**

🔍 Step 6: Find Closest Pair

Minimum: {London, Paris, Madrid} to **Berlin = 878**

👉 Merge {London, Paris, Madrid} and {Berlin}

css
CopyEdit
Clusters: {London, Paris, Madrid, Berlin}, {Rome}

🔍 Step 7: Final Merge

Distance from {London, Paris, Madrid, Berlin} to Rome = min(1430, 1105, 1365, 1181) = **1105**

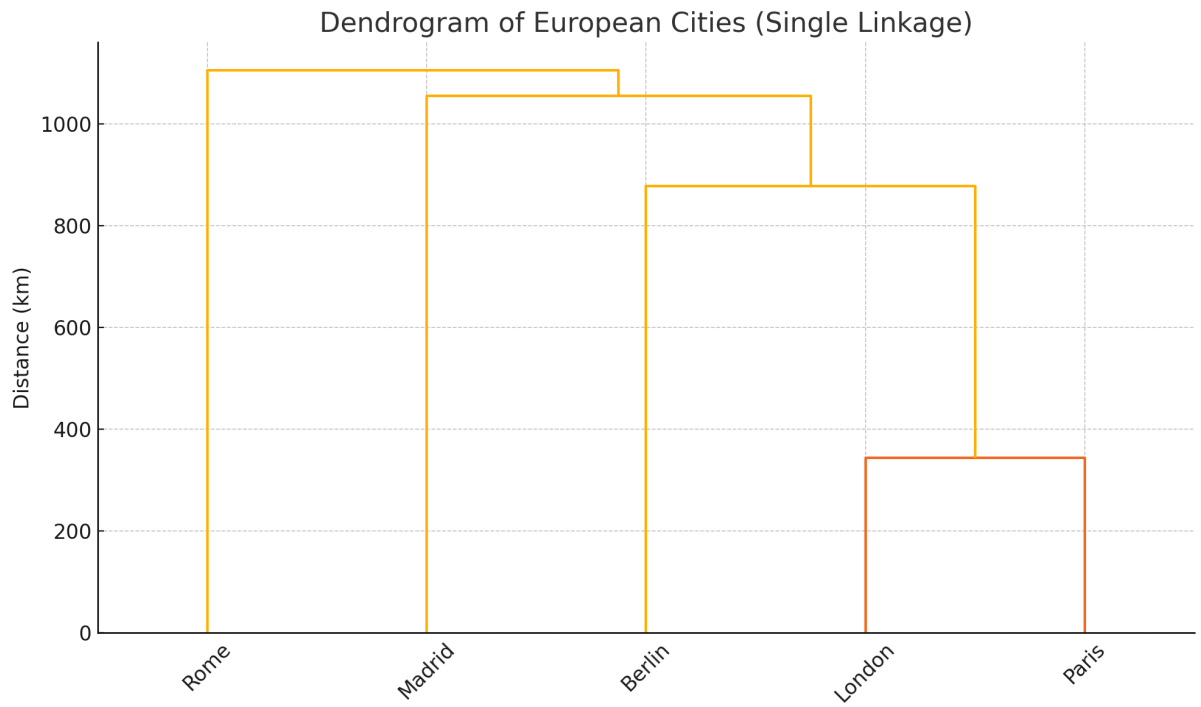
👉 Merge all into one:

Clusters: {London, Paris, Madrid, Berlin, Rome}

🗒️ Pseudo-Code
python
CopyEdit

```
cities = ["London", "Paris", "Berlin", "Rome", "Madrid"]
distances = {
   ("London", "Paris"): 344,
   ("London", "Berlin"): 930,
   ("London", "Rome"): 1430,
   ("London", "Madrid"): 1260,
   ("Paris", "Berlin"): 878,
   ("Paris", "Rome"): 1105,
   ("Paris", "Madrid"): 1054,
   ("Berlin", "Rome"): 1181,
   ("Berlin", "Madrid"): 1860,
   ("Rome", "Madrid"): 1365,
}

clusters = [{city} for city in cities]
```

```
while len(clusters) > 1:
    # Step 1: Find the closest pair of clusters
    min_dist = inf
    closest_pair = None

    for c1 in clusters:
        for c2 in clusters:
            if c1 == c2:
                continue
            # Use single linkage: min distance between any two cities
            dist = min(
                distances.get((a, b), distances.get((b, a), inf))
                for a in c1 for b in c2
            )
            if dist < min_dist:
                min_dist = dist
                closest_pair = (c1, c2)

    # Step 2: Merge closest clusters
    c1, c2 = closest_pair
    clusters.remove(c1)
    clusters.remove(c2)
    clusters.append(c1.union(c2))

    print(f"Merged {c1} and {c2} with distance {min_dist}")
```



Let's walk through a **divisive clustering** example using the same set of European cities and distances. While **agglomerative clustering** builds up from individual points, **divisive clustering** starts with **all cities in one big cluster** and **recursively splits** it into smaller clusters.

🧠 Divisive Clustering Overview

- Start with **all data in a single cluster**.
- At each step, choose a cluster to **split** based on some rule (e.g. maximizing distance between groups).
- Continue until each point is its own cluster, or until a desired number of clusters is reached.

🗺️ Same Cities & Distance Matrix

City	London	Paris	Berlin	Rome	Madrid
London	0	344	930	1430	1260
Paris	344	0	878	1105	1054
Berlin	930	878	0	1181	1860
Rome	1430	1105	1181	0	1365
Madrid	1260	1054	1860	1365	0

🧩 Step-by-Step Divisive Clustering

🔸 Step 1: Start with all cities

css
CopyEdit
Cluster: {London, Paris, Berlin, Rome, Madrid}

🔸 Step 2: Split based on distance

Choose the pair of cities **furthest apart**:
Berlin ↔ Madrid = 1860 km

We'll use that as a seed to split the cluster into two groups:

- Group A (closer to Berlin): **Berlin, Paris, London**

- Group B (closer to Madrid): **Madrid, Rome**

yaml
CopyEdit
Clusters:
  C1: {Berlin, Paris, London}
  C2: {Madrid, Rome}

🔸 Step 3: Split Cluster C1

C1 = {Berlin, Paris, London}

Furthest cities in C1: **Berlin ↔ London = 930 km**
Split:

- Group A1 (closer to London): **London, Paris**
- Group A2: **Berlin**

yaml
CopyEdit
Clusters:
  C1.1: {London, Paris}
  C1.2: {Berlin}
  C2: {Madrid, Rome}

🔸 Step 4: Split Cluster C2

C2 = {Madrid, Rome}
Distance = 1365 km
Split into individual cities:

yaml
CopyEdit
Clusters:
  C1.1: {London, Paris}
  C1.2: {Berlin}
  C2.1: {Madrid}
  C2.2: {Rome}

🔸 Step 5: Final Split

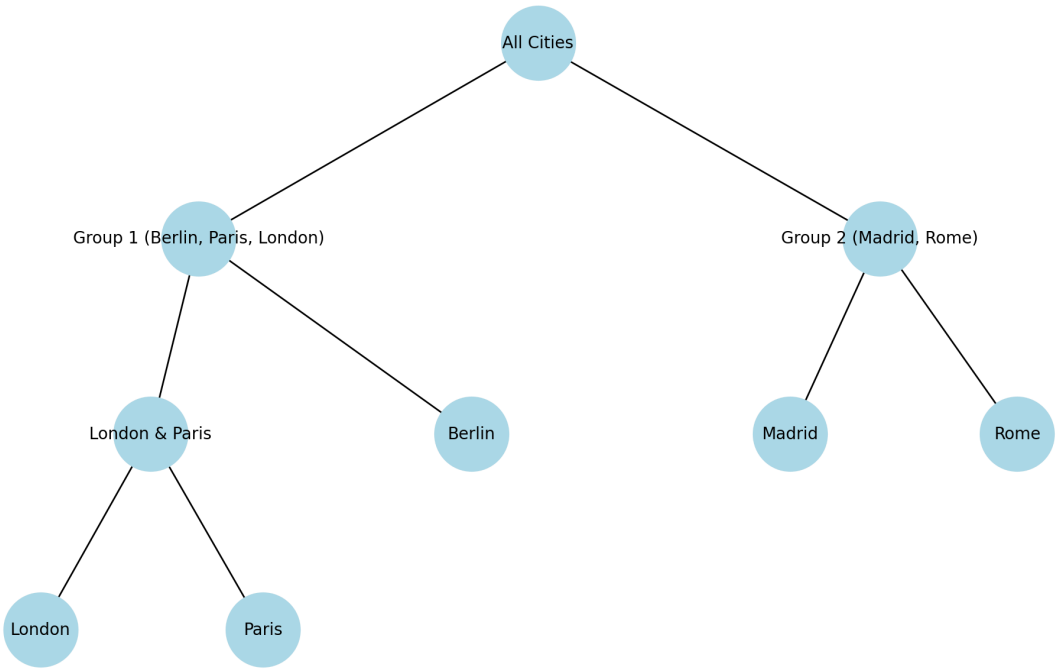
C1.1 = {London, Paris} → Distance = 344 km → Final split

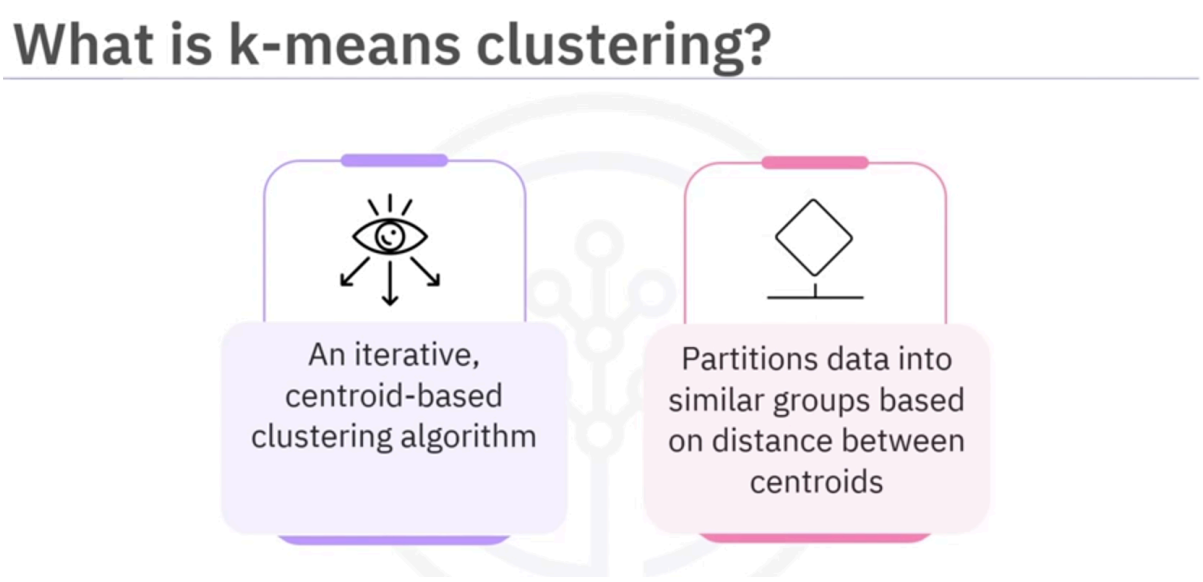
css

CopyEdit
Clusters:
{London}, {Paris}, {Berlin}, {Madrid}, {Rome}

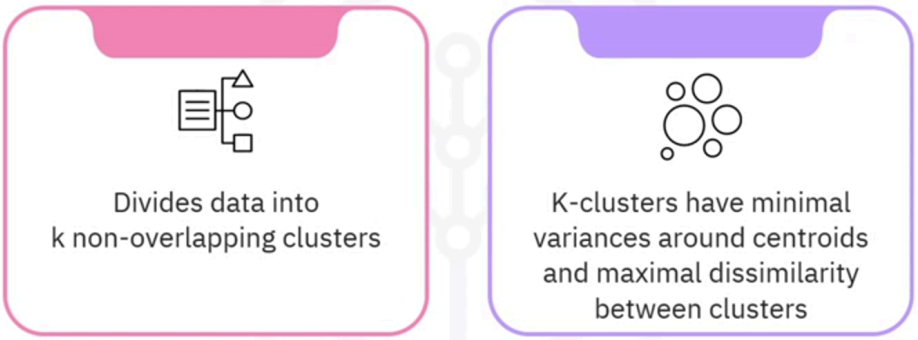
🗒️ Pseudo-Code (Simplified)
python
CopyEdit

```
# Initial cluster
clusters = [{"London", "Paris", "Berlin", "Rome", "Madrid"}]

def furthest_pair(cluster, distances):
    max_dist = -1
    pair = None
    for a in cluster:
        for b in cluster:
            if a != b:
                d = distances.get((a, b), distances.get((b, a), 0))
                if d > max_dist:
                    max_dist = d
                    pair = (a, b)
    return pair

def split_cluster(cluster, a, b, distances):
    group_a = set()
    group_b = set()
    for city in cluster:
        d_to_a = distances.get((city, a), distances.get((a, city), 0))
        d_to_b = distances.get((city, b), distances.get((b, city), 0))
        if d_to_a < d_to_b:
            group_a.add(city)
        else:
            group_b.add(city)
    return group_a, group_b

# Repeat splitting until all clusters are singletons
```


Divisive Clustering Tree of European Cities
All Cities
Group 1 (Berlin, Paris, London)
Group 2 (Madrid, Rome)
London & Paris
Berlin
Madrid
Rome
London
Paris

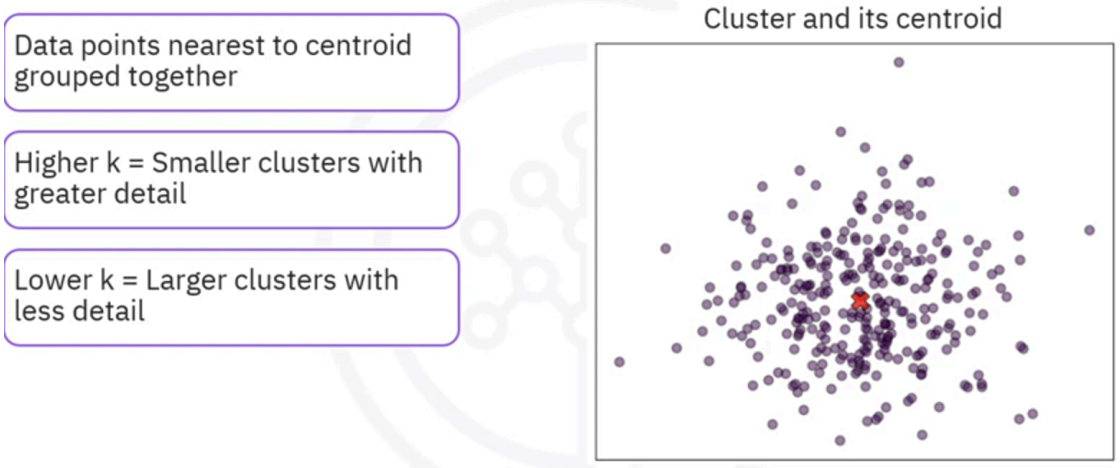

What is k-means clustering?
An iterative, centroid-based clustering algorithm
Partitions data into similar groups based on distance between centroids

What is k-means clustering?



Let’s understand what is a centroid. At the center of the cluster there is a centroid.

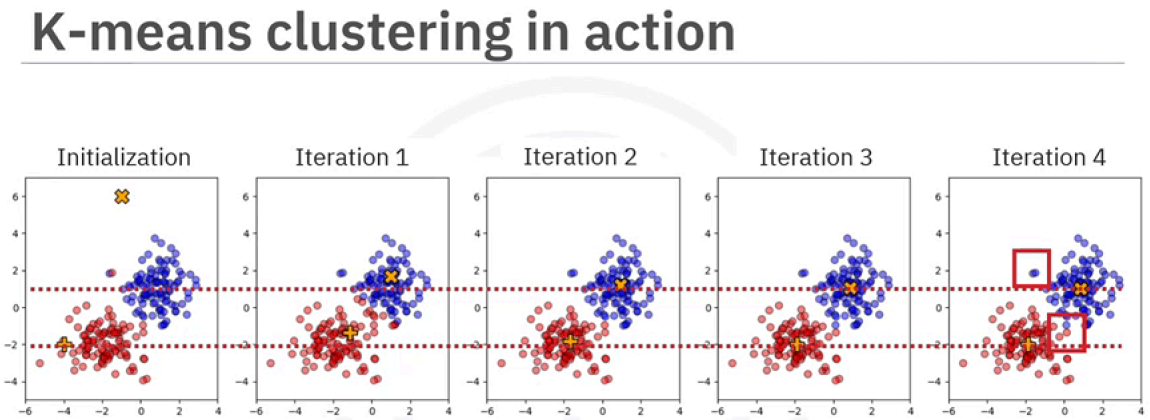
K-means algorithm



Centroids can be data point in the feature space. These data centroids can be data points or other points from the feature space.
Assign iteratively points to cluster and update centroids:

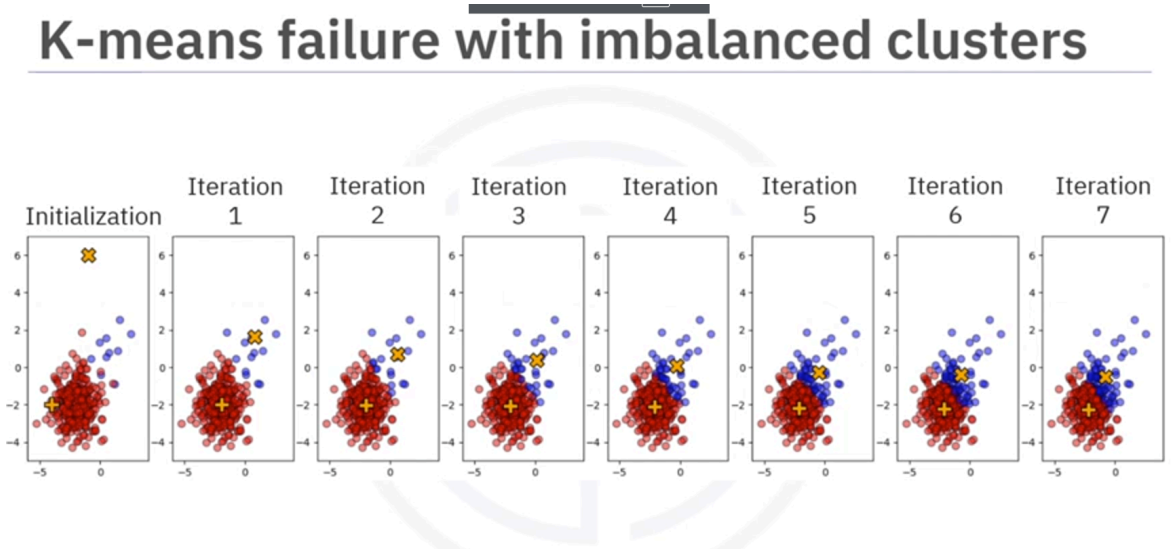
1. Compute the distance matrix. Consisting to the distance of each point to each centroid.
2. Assign each data point to cluster with nearest centroid.
3. Update cluster centroids as the mean of the cluster

Repeat until centroids stabilize or max iterations reached. The algorithms converge when centroids stop moving.



In each sample we see that the centroid get closer to the final destination. Iteration 3 and Iteration 4 are almost identical, thus k-means already converged by iteration 3.

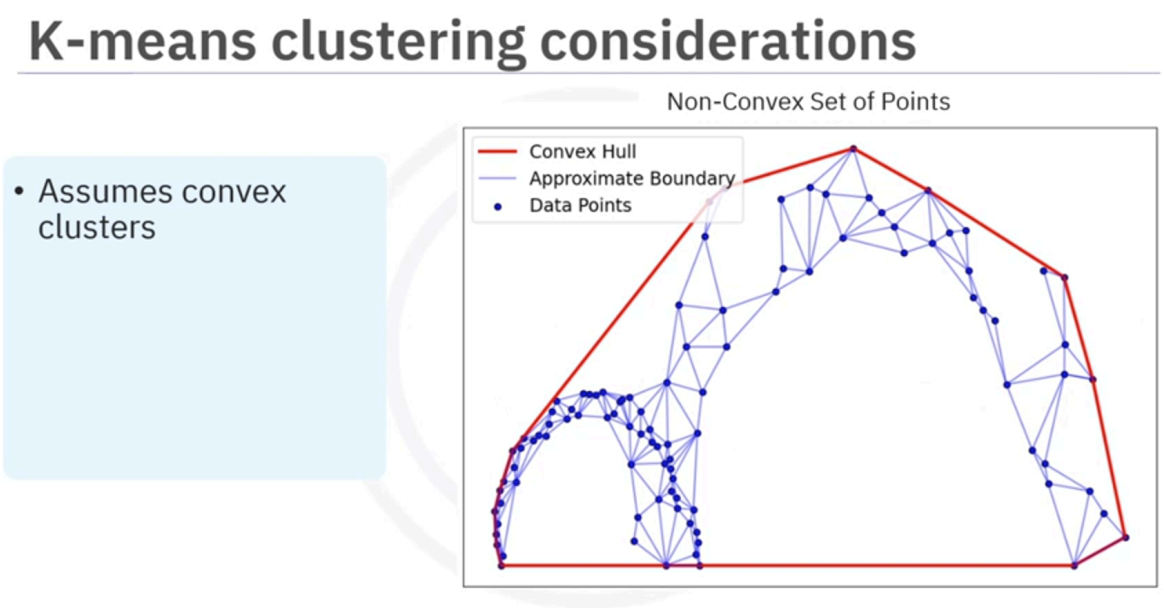
K-means however doens't perform very well on imbalance clusters.



In this example, the **red cluster contains 200 points**, while the **blue cluster contains only 10**.

During the **first iteration**, the clustering result appears quite reasonable. The **centroids quickly stabilize**, but a problem arises: the **centroid of the smaller blue cluster is pulled closer to the centroids of the larger red cluster** due to its limited number of points.

As a result, the blue centroid starts to **"steal" points from the red cluster**, gradually **consuming more and more of its neighboring red points**, leading to a degraded clustering outcome over subsequent iterations.



K-means assumes clusters are **convex** because of how it works:

- It assigns each point to the **nearest centroid**, using **Euclidean distance**.
- Then it updates centroids by computing the **mean** of all points in each cluster.

This logic works well only if each cluster is **convex-shaped** — meaning there's no complicated structure like holes, spirals, or separated sub-groups.

A **convex cluster** is a group of points where:

> If you take **any two points** in the cluster, the **entire line segment** connecting them lies **completely within the cluster**.

✅ Example: Convex

- A **circle**, **ellipse**, or **square** is convex.
- K-means works well here.

❌ Example: Non-convex

- A **crescent**, **donut (ring)**, or **spiral** is non-convex.
- K-means struggles because the mean of such shapes may fall **outside the actual cluster**.

If clusters are non-convex:

- The **mean** doesn't represent the cluster well.
- Points from different parts of the same true cluster might get assigned to different centroids.
- K-means may split or merge clusters incorrectly.

Imagine trying to fit a **donut shape** with K-means:
It'll either:

- Put one centroid in the center hole (which has no points), or
- Split the ring into several parts, even though it should be one cluster.

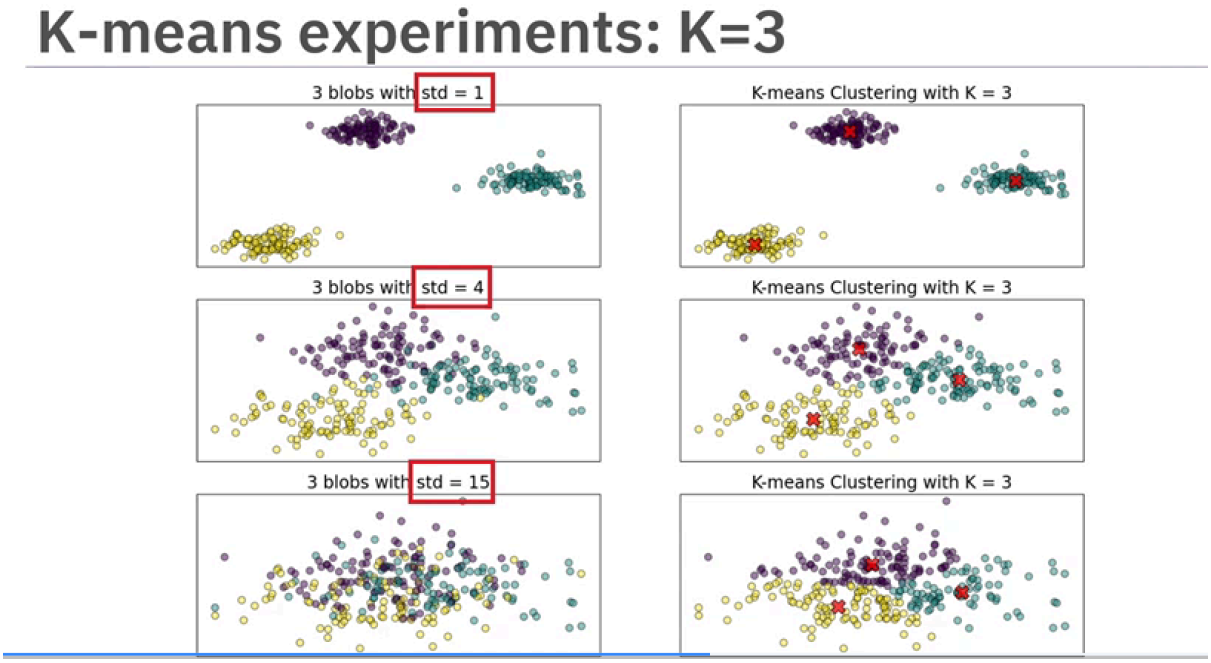
The algorithm also makes several key assumptions and exhibits certain behaviors:

- **Assumes balanced cluster sizes**:
  K-means tends to work best when clusters have **similar densities and sizes**. If clusters are imbalanced (e.g., one large and one small), the larger cluster can dominate the centroid updates, causing smaller clusters to be absorbed or misclassified over time.

- **Sensitive to outliers and noise**:
  Since K-means uses the **mean** to determine cluster centers, even a single outlier can significantly shift a centroid, leading to poor clustering. Outliers can distort the true structure of the data.

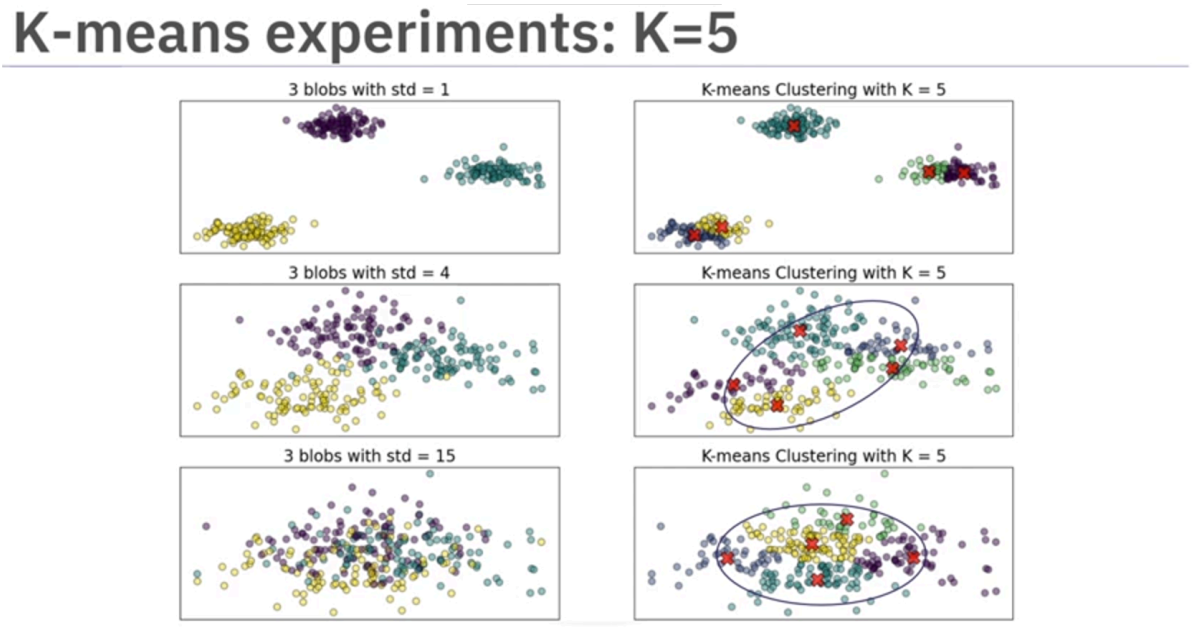
- **Scales very well on big data** *(as a partition-based algorithm)*:
  K-means is computationally efficient because it has **linear complexity** with respect to the number of data points (O(n)), assuming a fixed number of clusters and dimensions. It simply partitions data into kkk subsets using fast distance calculations and centroid updates, which makes it highly suitable for **large-scale datasets**, especially when combined with techniques like **mini-batch K-means** or **parallelization**.

Goal of K-means



The difference here is the standard deviation of the points in each blob. K-means doesn't know anything about classes (unsupervised learning). The goal of k-means is to uncover these classes. K-means has distinguished quite well the blob for standard deviation 1 and 4.

When K is too large, K-means returns unexceptable results.

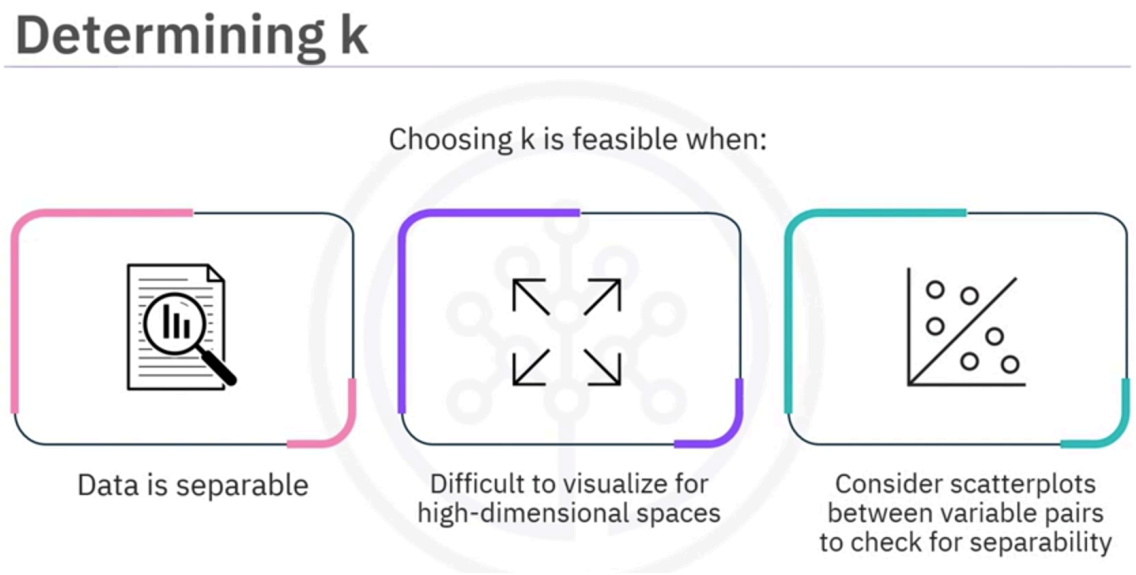


How do you find the best value for K when you don't know much about your data?

Determining K.

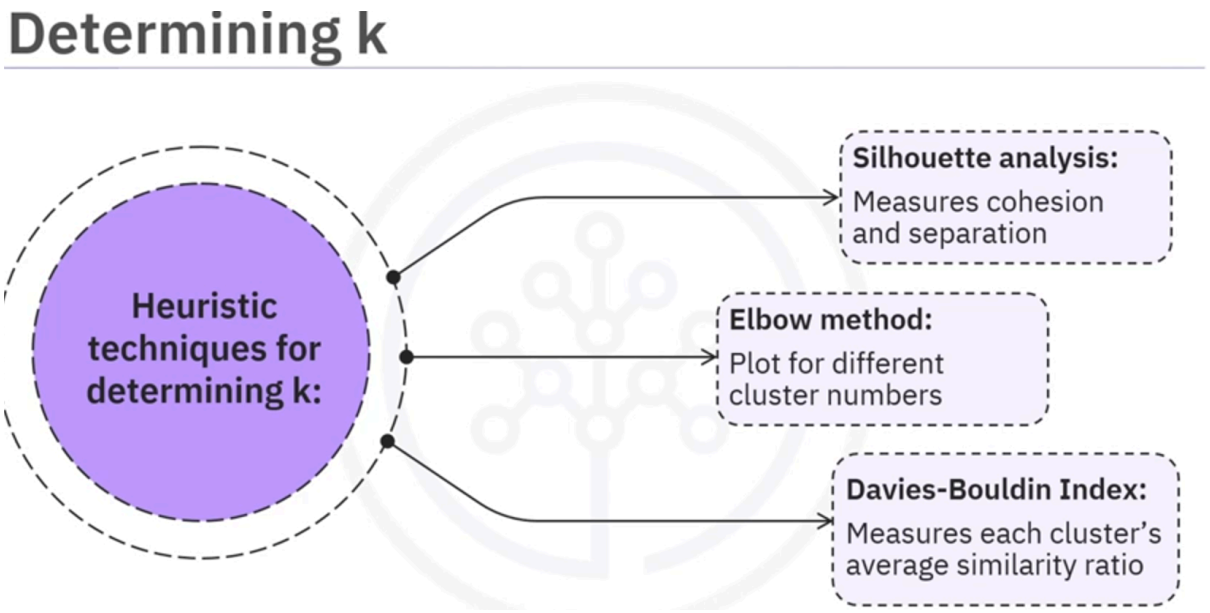
Choosing K is feasible when:

- Data is sperable.



You can check insights on the scatterplot between variable pairs to see if they are separable.

There are also heuristics for determining k-mean



📉 1. Elbow Method

🧠 Idea:

Measure how the **Within-Cluster Sum of Squares (WCSS)** decreases as kkk increases. WCSS is the total distance between points and their assigned centroid.

📍 How it works:

- Run K-means for a range of kkk values (e.g., 1 to 10).

- Plot kkk vs. **WCSS**.
- Look for an **"elbow" point** — a point where the rate of improvement sharply drops.
- That elbow suggests the optimal number of clusters.

📊 Interpretation:

- Before the elbow: adding clusters improves compactness significantly.
- After the elbow: improvements are minimal.

📏 2. Silhouette Analysis

🧠 Idea:

Measures how **well each point fits** in its assigned cluster vs. other clusters.

📍 Formula:

For each point:

s=b−amax(a,b)s = \frac{b - a}{\max(a, b)}s=max(a,b)b−a

Where:

- aaa = average intra-cluster distance (to points in the same cluster)
- bbb = average distance to the **nearest neighboring cluster**

📊 Interpretation:

- s≈1s \approx 1s≈1: well clustered
- s≈0s \approx 0s≈0: on the boundary
- s<0s < 0s<0: probably in the wrong cluster

🧪 **Choose kkk** with the **highest average silhouette score**.

📉 3. Davies-Bouldin Index (DBI)

🧠 Idea:

Measures the **average similarity between clusters** — based on **intra-cluster dispersion** and **inter-cluster distance**.

📍 Formula:

DBI=1k∑i=1kmaxj≠i(σi+σjdij)DBI = \frac{1}{k} \sum_{i=1}^{k} \max_{j \neq i} \left(\frac{\sigma_i + \sigma_j}{d_{ij}} \right)DBI=k1i=1∑kj=imax(dijσi+σj)

Where:

- σi\sigma_iσi = scatter (avg distance of points in cluster iii to centroid)
- dijd_{ij}dij = distance between centroids of clusters iii and jjj

📊 Interpretation:

- **Lower DBI is better** — implies clusters are compact and far apart.
- Choose the kkk with the **lowest Davies-Bouldin Index**.

✅ Summary Table

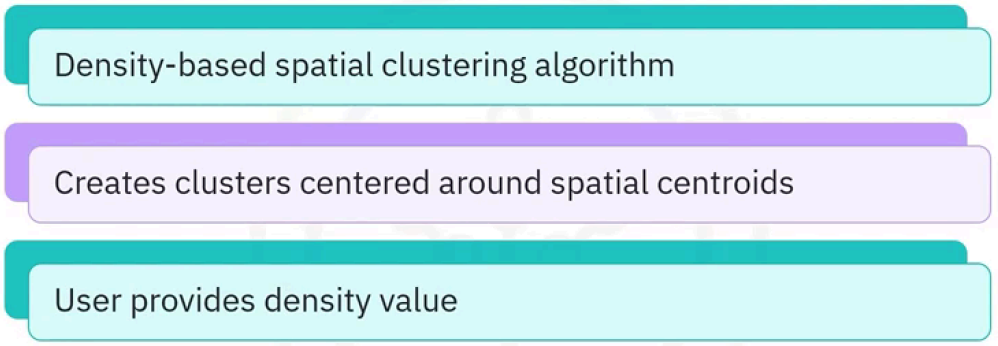
Method	Goal	What to Look For
Elbow Method	Minimize within-cluster distance	Sharp bend ("elbow") in WCSS curve
Silhouette Score	Maximize cohesion/separation	Highest average silhouette
Davies-Bouldin	Minimize intra-cluster & maximize inter-cluster distances	Lowest DBI

Recap

K-means:

- Iterative, centroid-based clustering algorithm
- Partitions data set into similar groups
- Clustering algorithm categorizes data points into clusters
- Doesn't perform well on imbalanced clusters and assumes that clusters are convex
- Objective: Minimize within-cluster variance
- Heuristic techniques for gauging k-means performance:
  - Silhouette analysis
  - Elbow method
  - Davies-Bouldin index

DBSCAN clustering



The density value is positioned around a special centroid, the area immediately near to the centroid is referred as neighbourhood and DBScan tends to define neighbourhoods of clusters with specified density. DBSCAN can discover clusters of any shape, size or density in your data.

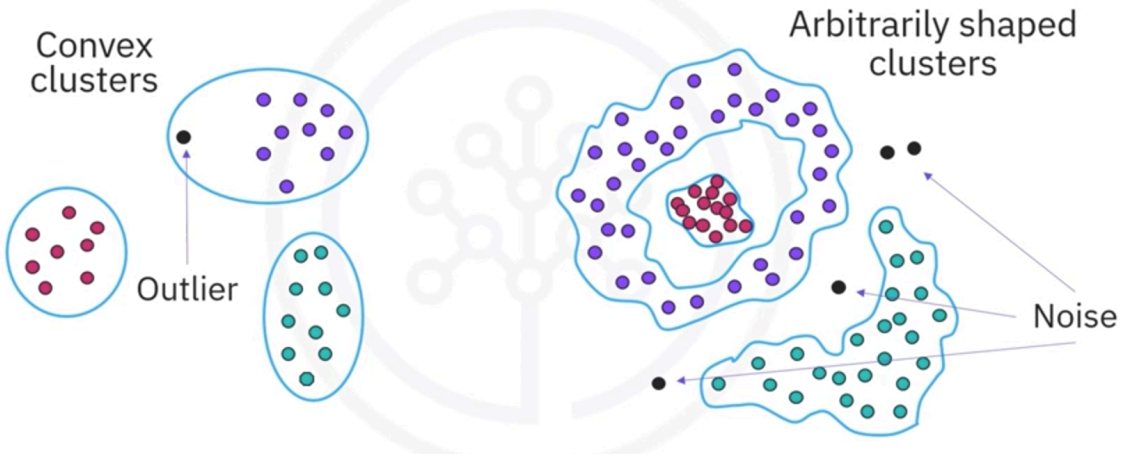
DBSCAN clustering

Discovers clusters of any shape, size, or density

Distinguishes between data points that are part of a cluster and noise

Useful for data with outliers or when cluster number is unknown

Centroid and density-based clustering



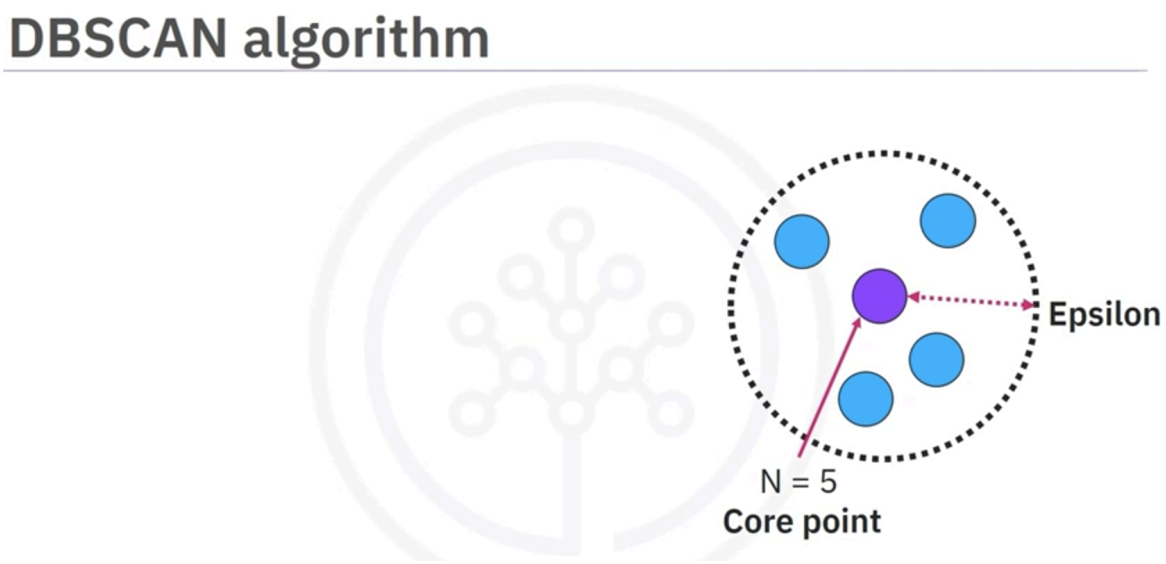
DBSCAN is particularly useful not just for clustering, but also for **detecting outliers** in a dataset. Unlike centroid-based methods like **K-means**, which work well only when clusters have a **convex structure**, DBSCAN can handle **arbitrarily shaped clusters** — making it a more flexible choice in real-world scenarios.

In practice, many datasets contain clusters that are **not convex** and may also include **noise points** (i.e. outliers or irrelevant data).
To cope with this, DBSCAN classifies points into:

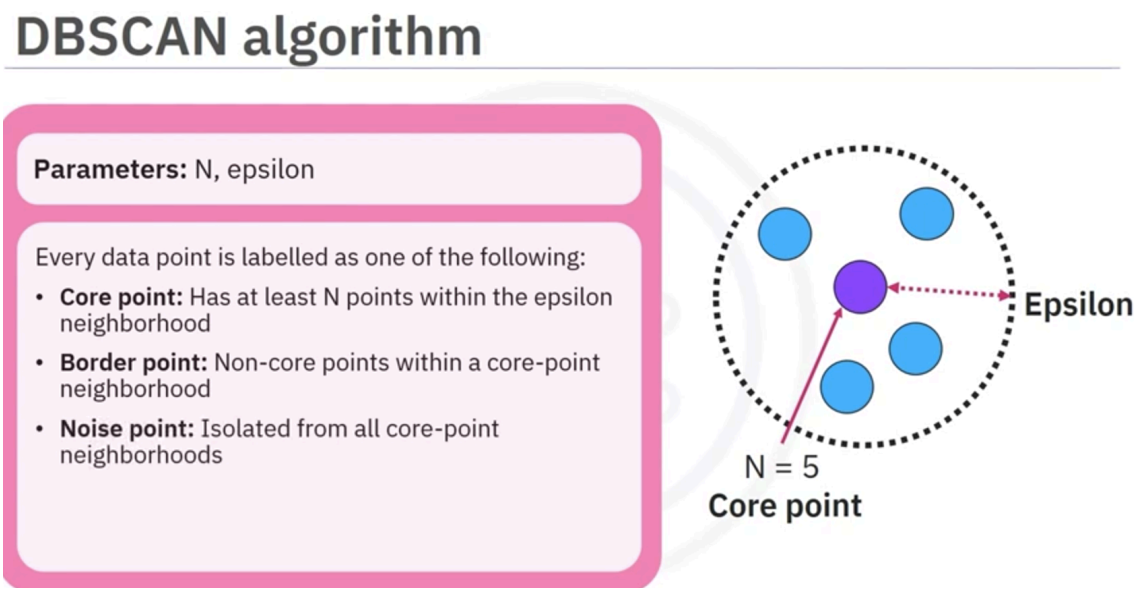
- **Core points** (densely packed)
- **Border points** (near core points)
- **Noise** (points that don't belong to any cluster)

This makes DBSCAN highly effective for:

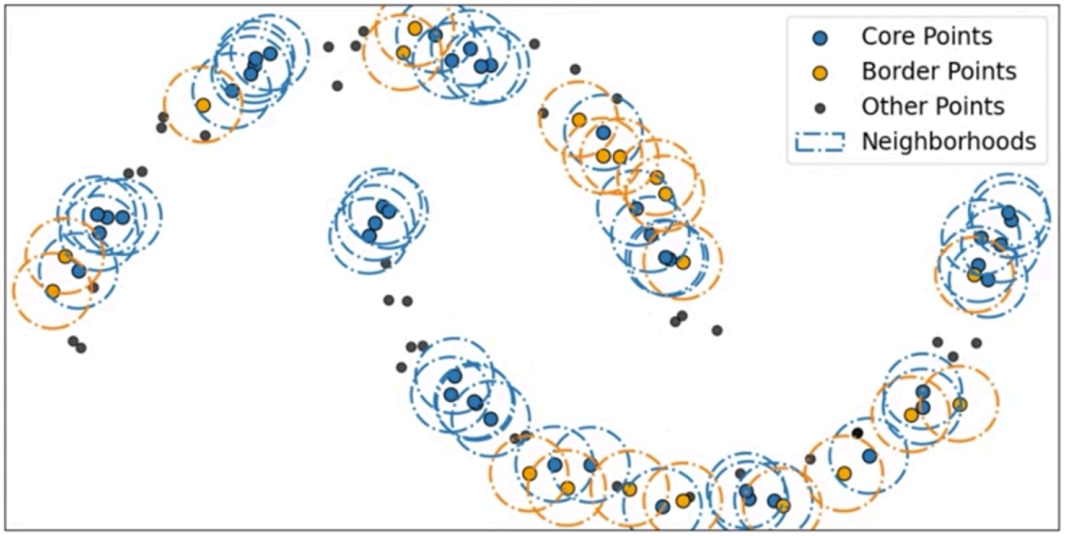
- Identifying **non-convex clusters**
- Automatically **filtering out noise**
- Avoiding the need to predefine the number of clusters



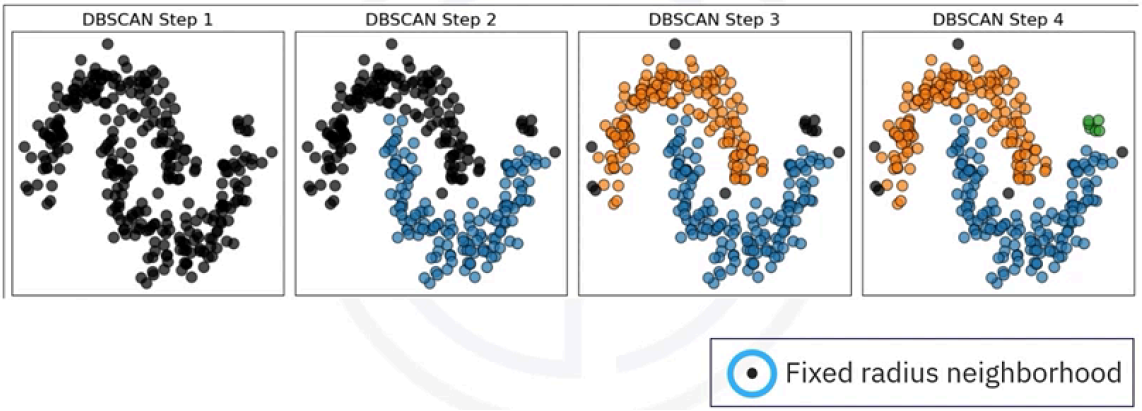
How does it work?



Core and border points



DBSCAN experiment



Step 1.All points are labeled as noise.

Step2.