9 Splitting data by asking questions: Decision trees

published book

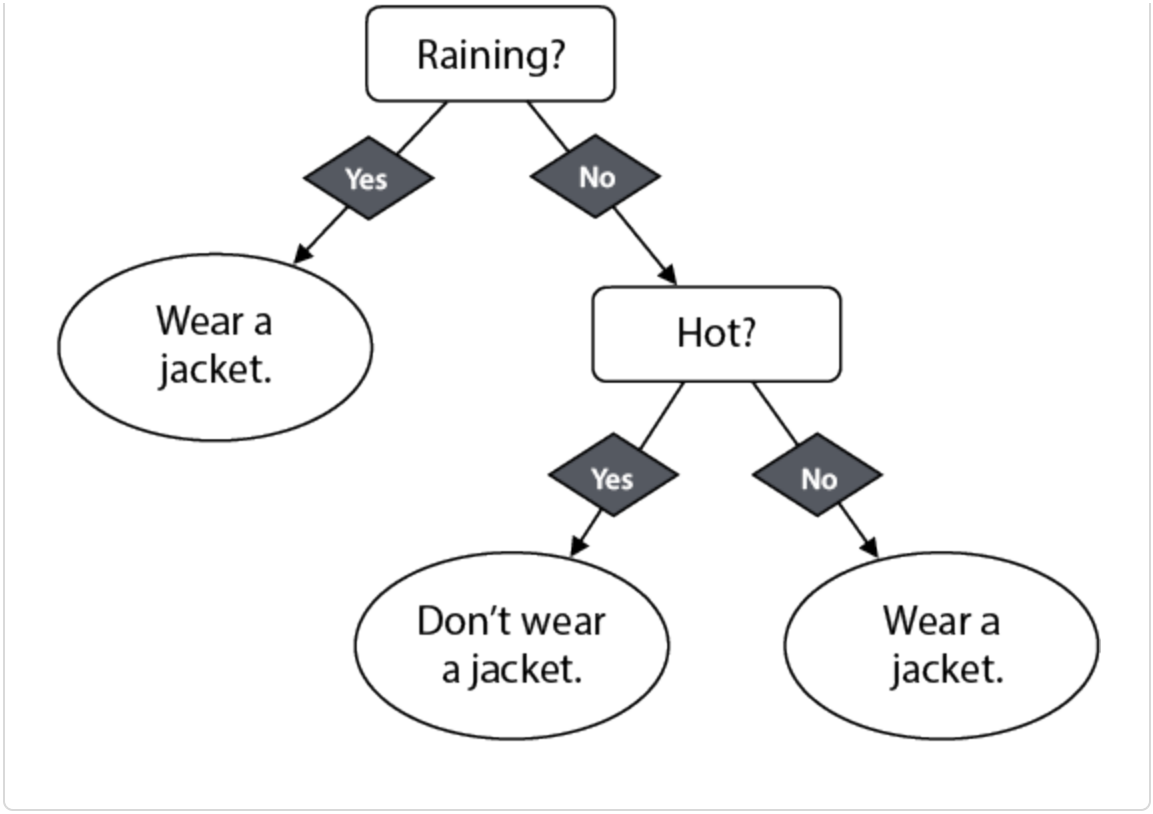
In this chapter

- what is a decision tree
- using decision trees for classification and regression
- building an app-recommendation system using users' information
- accuracy, Gini index, and entropy, and their role in building decision trees
- using Scikit-Learn to train a decision tree on a university admissions dataset

In this chapter, we cover decision trees. Decision 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

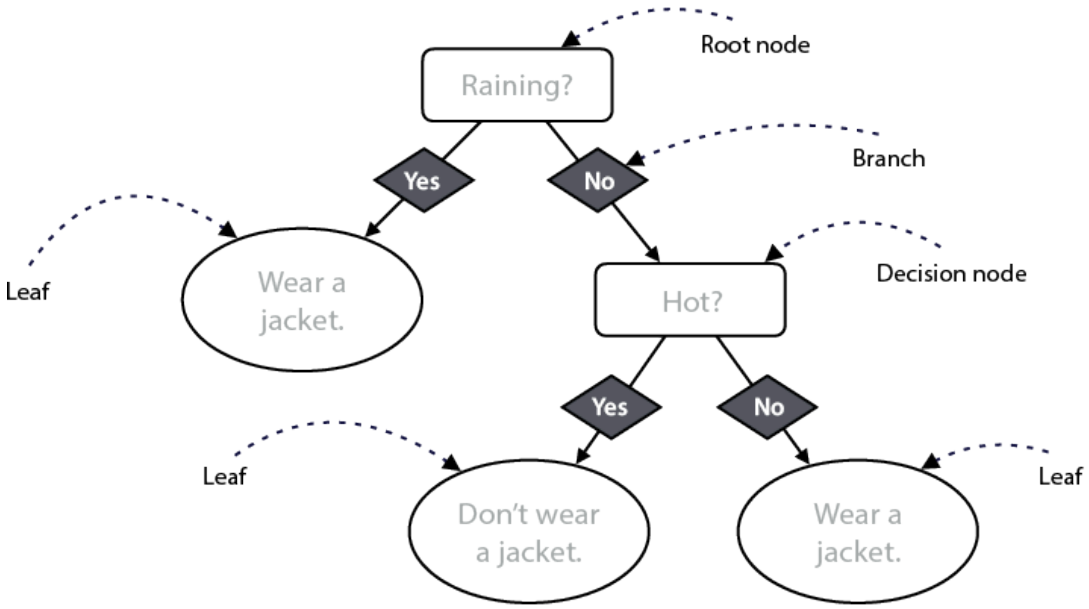
The two edges emanating from each decision node, corresponding to the "yes" and "no" answers to the question in the node. In this chapter, by convention, the branch to the left corresponds to "yes" and the branch to the right to "no."

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.

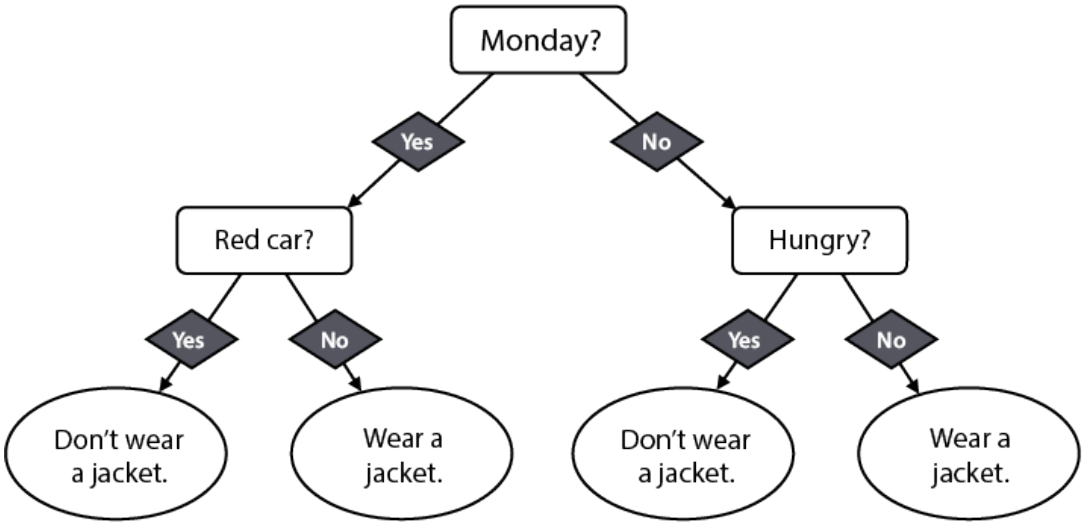
Throughout this chapter, nodes are drawn as rectangles with rounded edges, the answers in the branches as diamonds, and leaves as ovals. Figure 9.2 shows how a decision tree looks in general.

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:

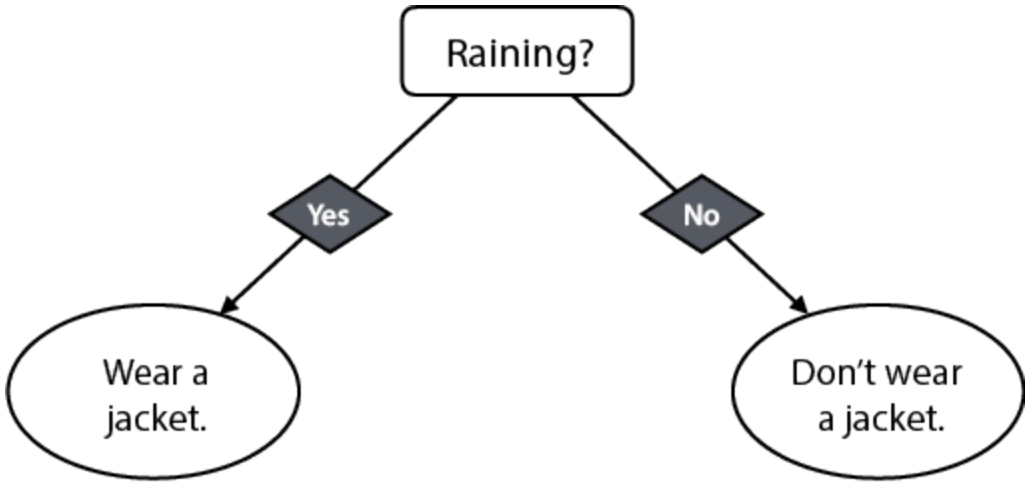
PICKING A GOOD FIRST QUESTION

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?

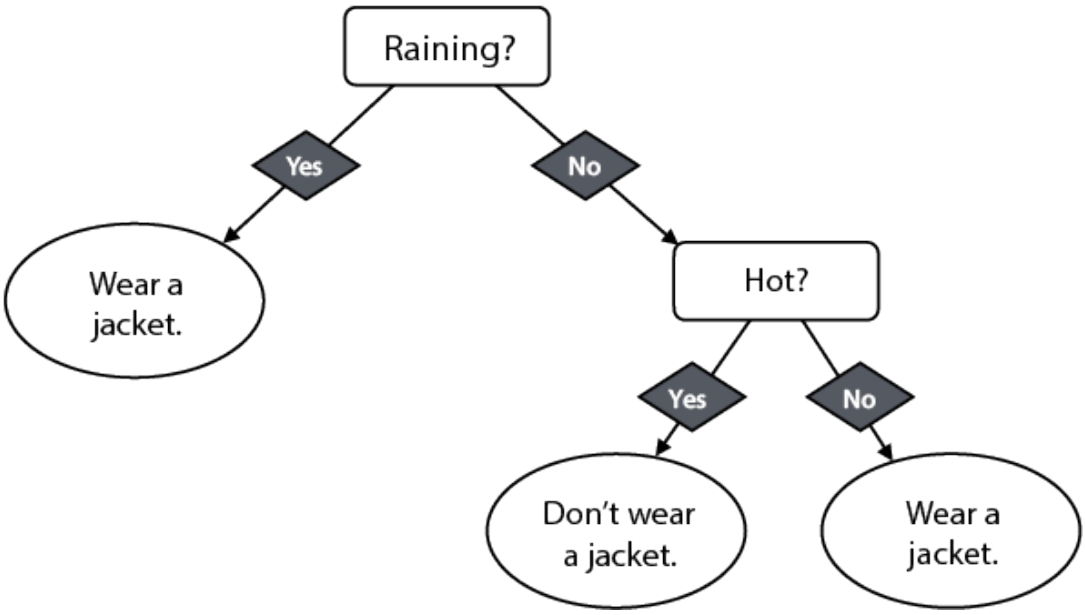
Out of these five questions, which one seems like the best one to help us decide whether we should wear a jacket? Our intuition says that the last three questions are useless to help us decide. Let's say that from experience, we've noticed that among the first two, the first one is more useful. We use 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

Atom Count

Beehive Finder

Check Mate Mate

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. (view table figure)

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" (view table figure)

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.

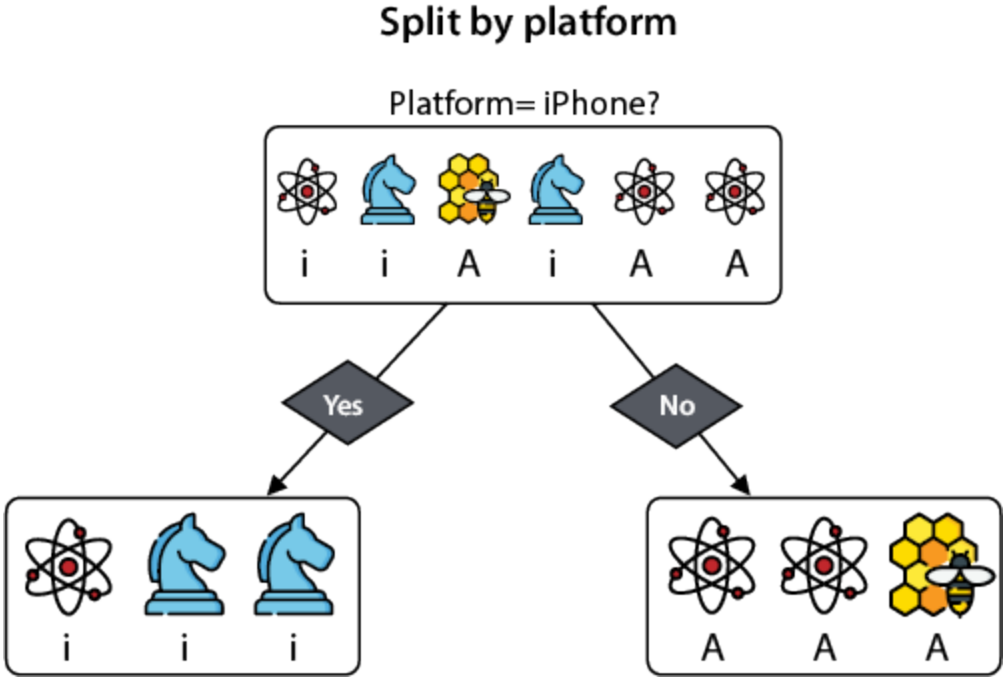
FIRST QUESTION: DOES THE USER USE AN IPHONE OR ANDROID?

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.

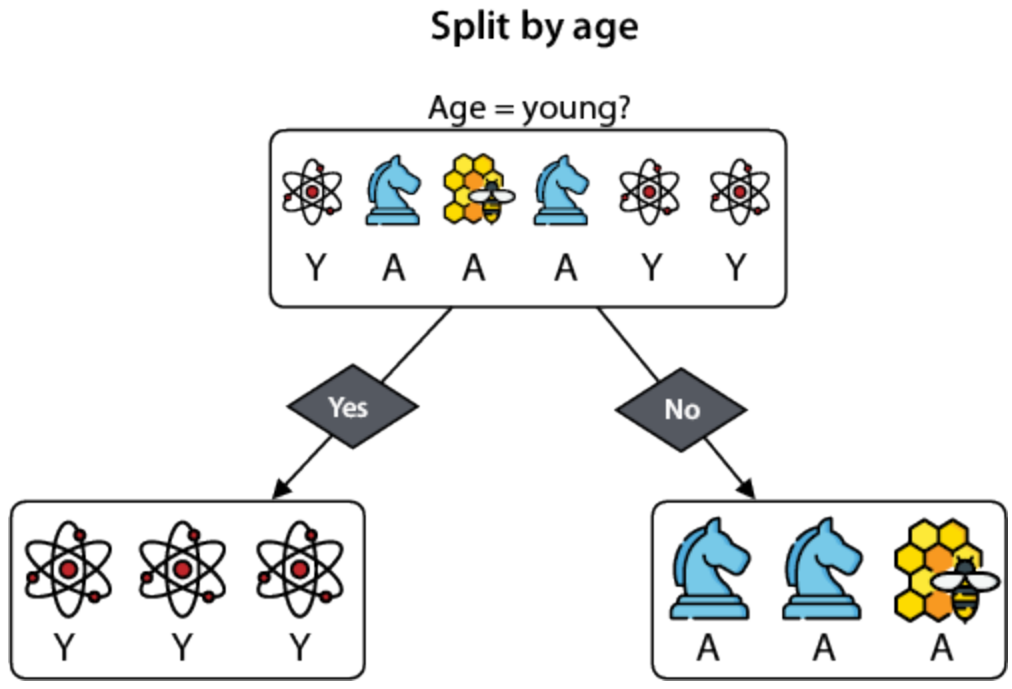
SECOND QUESTION: IS THE USER YOUNG OR ADULT?

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**.

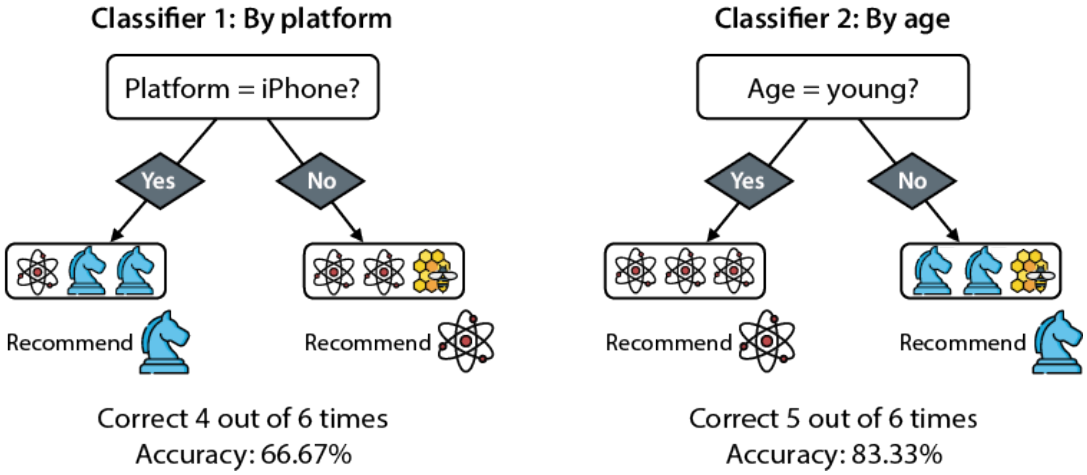
Classifier 2: What is your age?

- 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

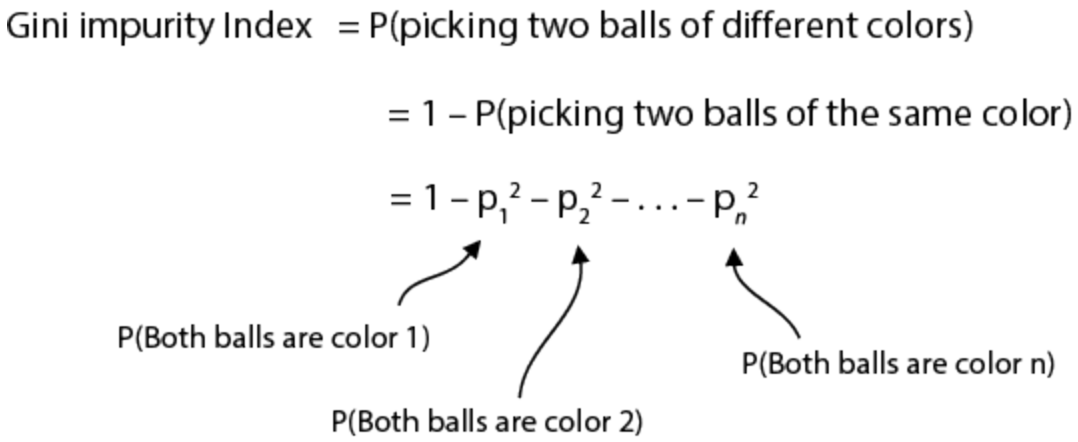
$$\text{P(both balls are color } i) = p_i^2.$$

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

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

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)

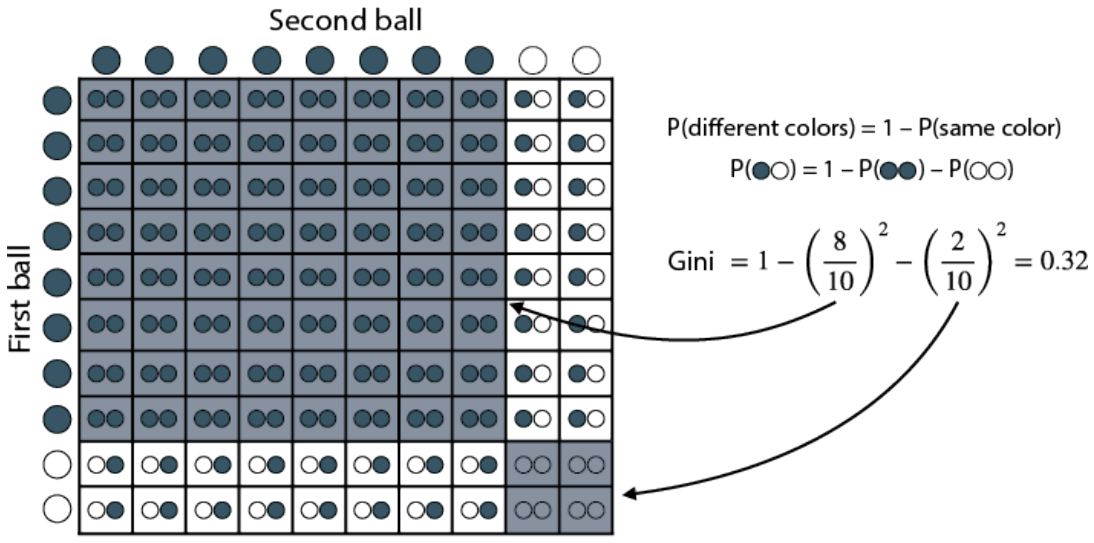
$$\text{Gini impurity index} = 1-\left(\frac{8}{10}\right)^2-\left(\frac{2}{10}\right)^2 = 1-\frac{68}{100}=0.32$$

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:

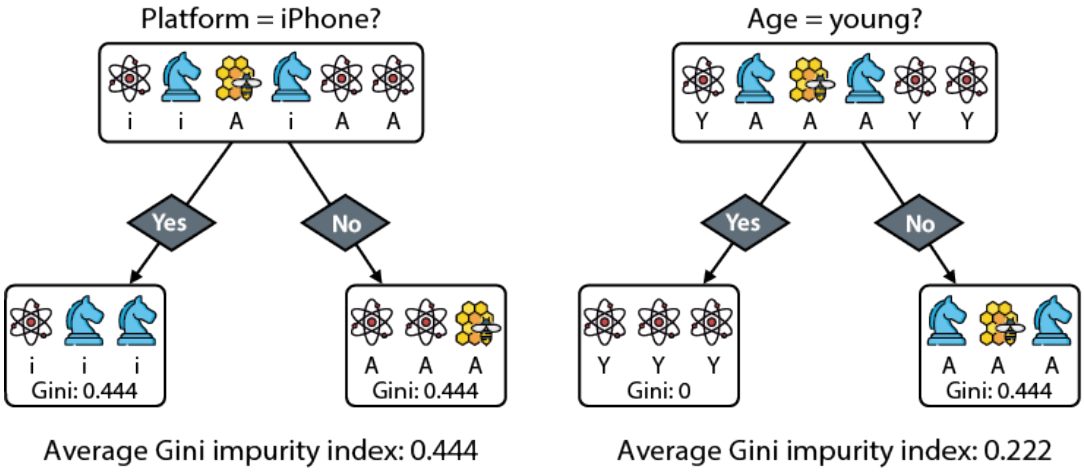
Classifier 1 (by platform):

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):

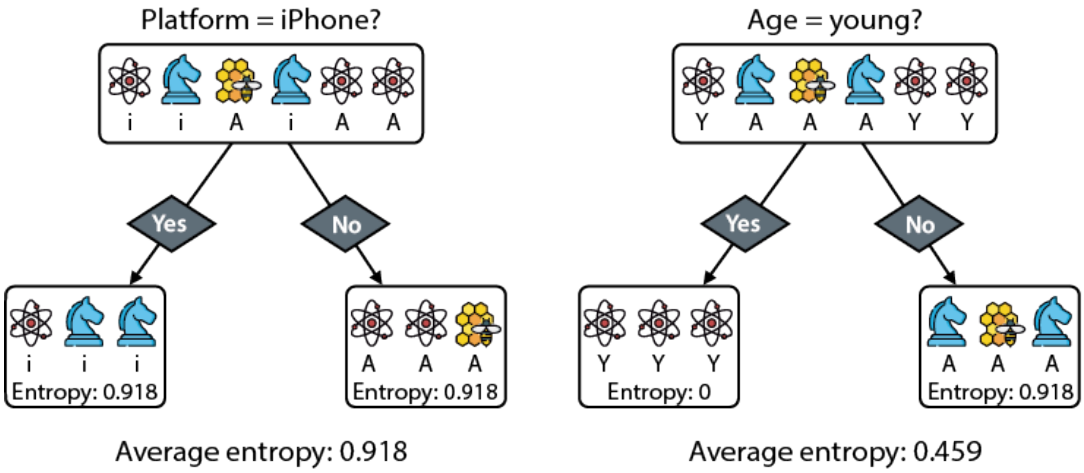
Classifier 1 (by platform):

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.

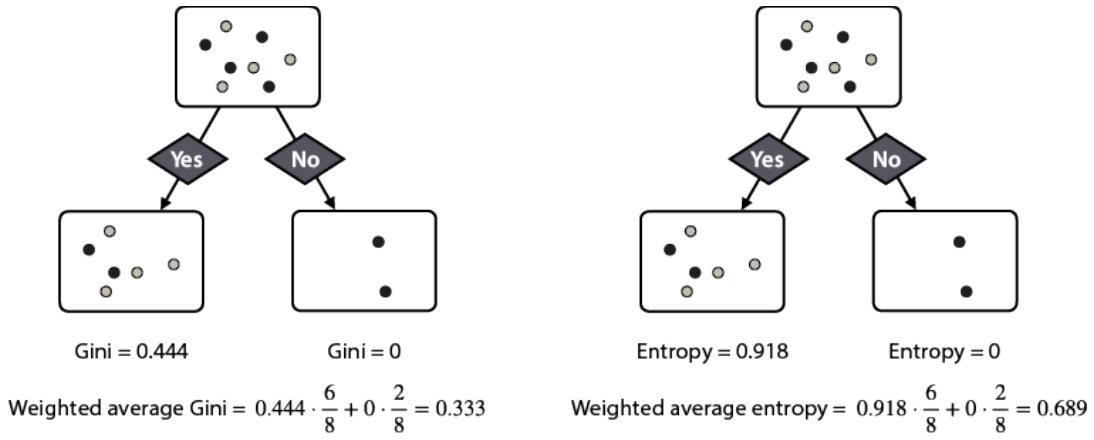
Entropy is a tremendously important concept in probability and statistics, because it has strong connections with information theory, mostly thanks to the work of Claude Shannon. In fact, an important concept called *information gain* is precisely the change in entropy. To learn more on the

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.



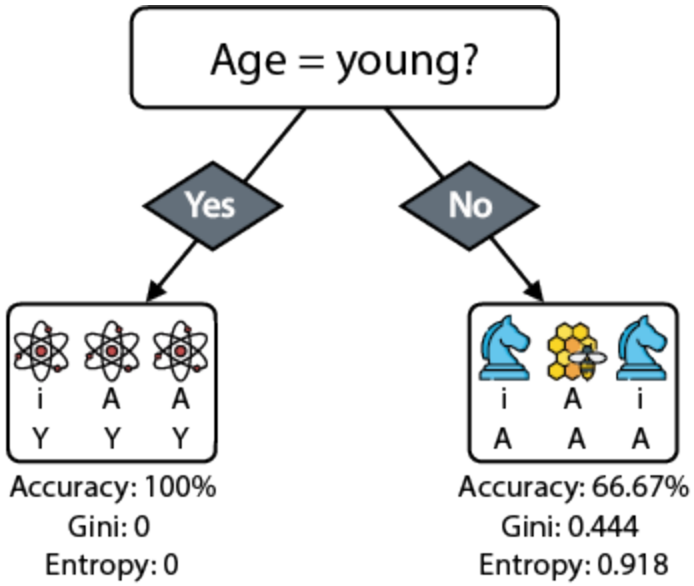
Now that we've learned three ways (accuracy, Gini index, and entropy) to pick the best split, all we need to do is iterate this process many times to build the decision tree! This is detailed in the next section.

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.

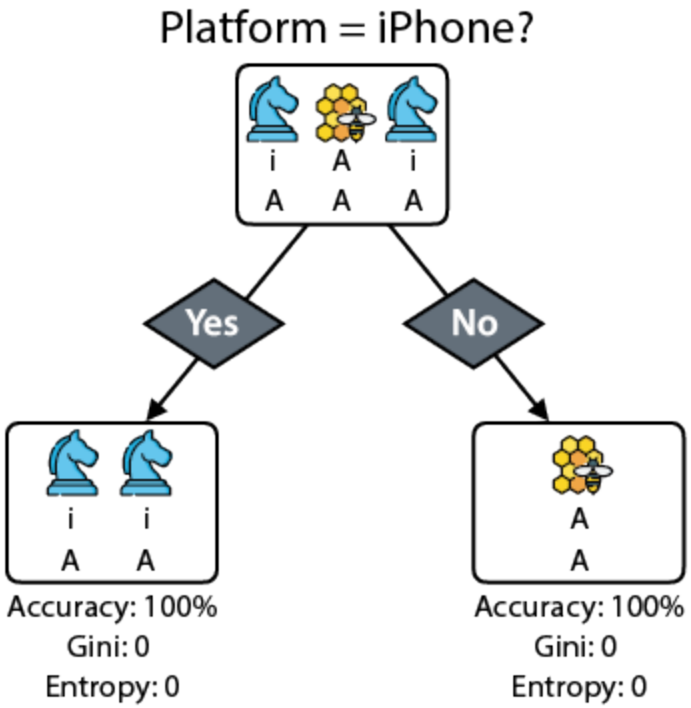
Figure 9.15 When we split our dataset by age, we get two datasets. The one on the left has three users who downloaded Atom Count, and the one on the right has one user who downloaded Beehive Count and two who downloaded Check Mate Mate.



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."

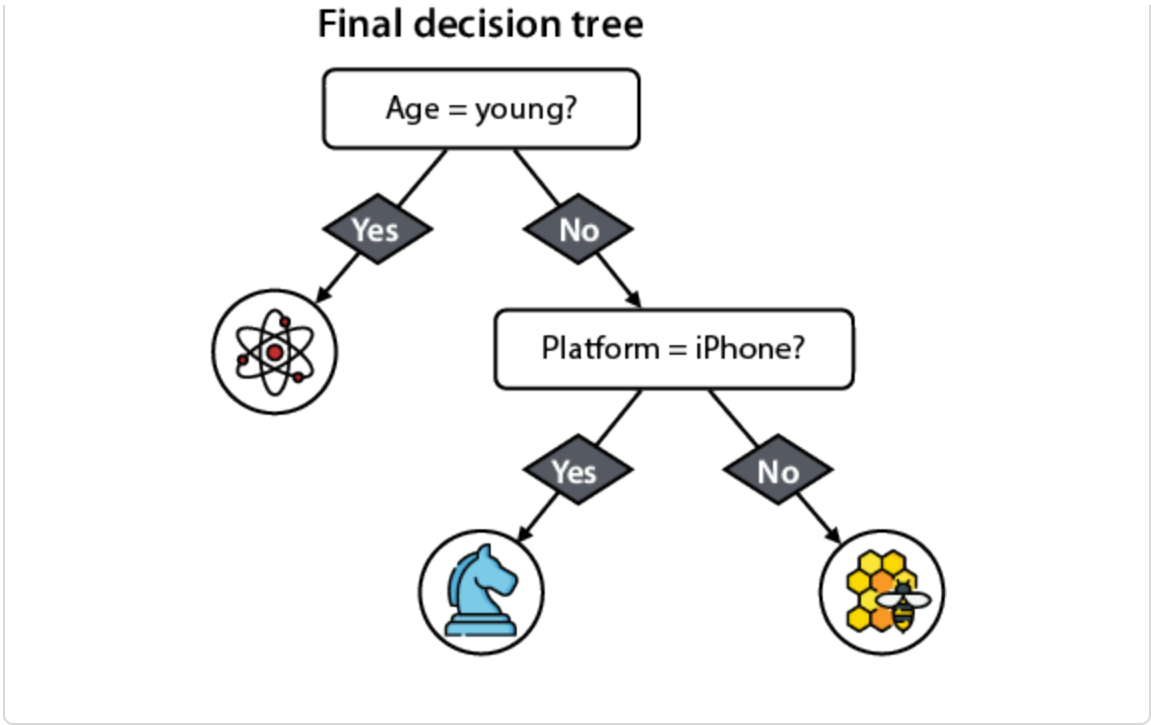
The dataset on the right can still be divided, because it has two labels: "Beehive Count" and "Check Mate Mate." We've used the age feature already, so let's try using the platform feature. It turns out that we're in luck, because the Android user downloaded Beehive Count, and the two iPhone users downloaded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