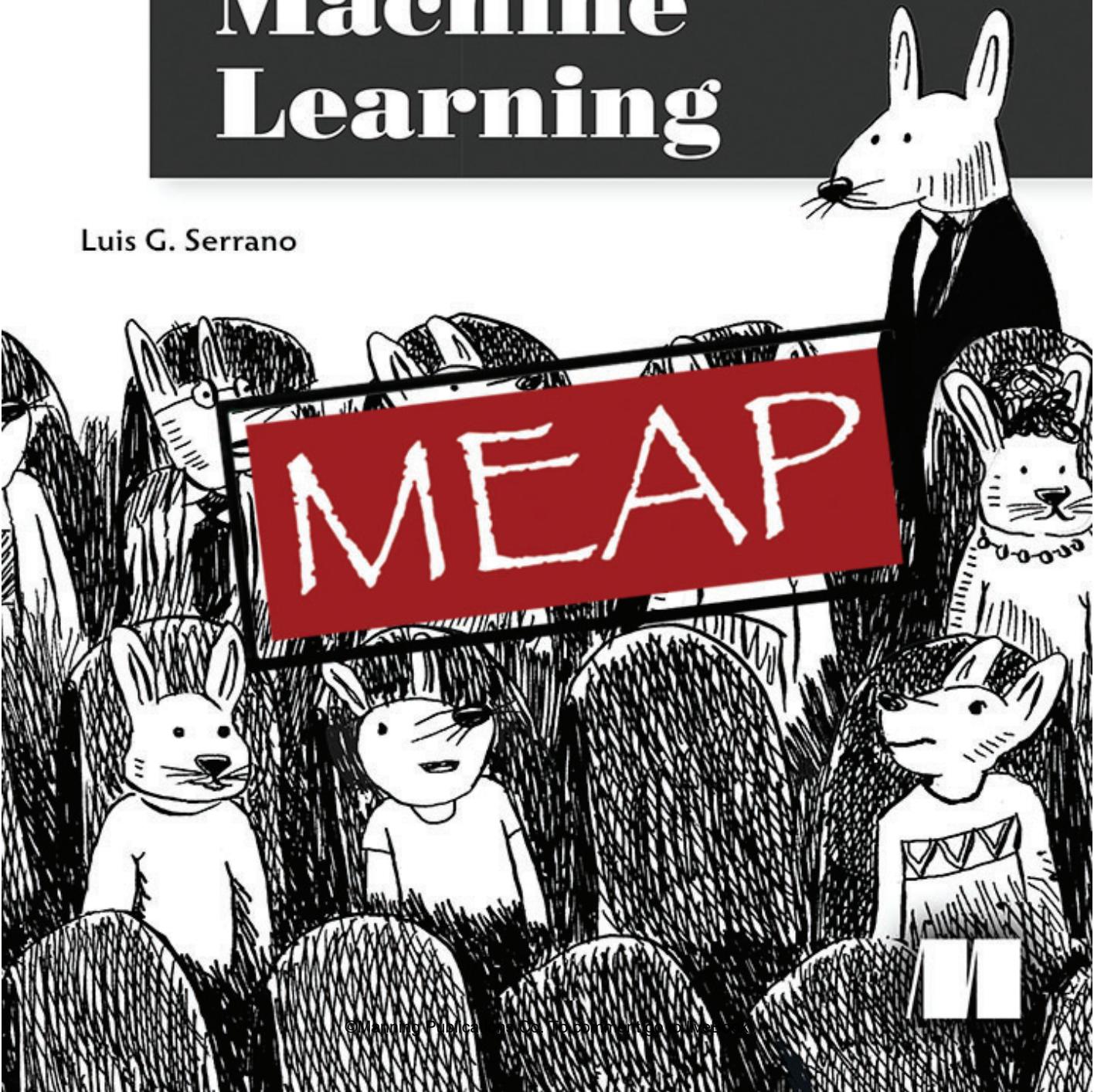


**grokking**

# Machine Learning

Luis G. Serrano





**MEAP Edition**  
**Manning Early Access Program**  
**Grokking Machine Learning**  
**Version 9**

Copyright 2020 Manning Publications

For more information on this and other Manning titles go to  
[manning.com](http://manning.com)

# welcome

---

Thank you for purchasing the MEAP edition of Grokking Machine Learning.

Machine learning is, without a doubt, one of the hottest topics in the world right now. Most companies are using it, or planning to use it, for many applications. Some people dub machine learning as the new electricity, or the new industrial revolution. I would go a bit farther and call it the new renaissance. Why? Because in the Renaissance, progress was made in the arts, the sciences, engineering, mathematics, and almost all the fields by the same people. With machine learning, this is finally possible again. With a strong knowledge of machine learning, one is able to derive cutting edge results in almost any field one decides to apply them, and this is fascinating. And that is what this book is for, to get you up to speed with the fast-moving world of machine learning!

But what is machine learning? I define it as "common sense, but for a computer." What does this mean? It means that machine learning is the practice of getting computers to make decisions using the decision-making process that we, humans, utilize in our daily life. Humans make many decisions based on past experiences, and we can teach this decision-making process to the computer, with the difference that computers call their past experiences "data."

"Most approaches to machine learning require a heavy amount of mathematics, in particular, linear algebra, calculus, and probability. While a solid understanding of these topics is very useful for learning machine learning, I strongly believe that they are not absolutely necessary. What is needed to understand machine learning is a visual mind, an intuition of basic probability, and a strong desire to learn."

In this book, I present machine learning as a series of exercises of increasing difficulty, in which the final goal is to teach a computer how to take a particular decision. Each chapter is dedicated to a different machine learning algorithm, and is focused in one use-case of this algorithm, such as spam detection, language analysis, image recognition, and so on. For the readers who are interested in programming, I also code the algorithms in Python, and teach some useful packages that are used in industry and research. The code is also shared in Github for easy download.

I really hope that you enjoy this book, and that it is a first step on your journey toward becoming a machine learning expert!

I encourage you to share questions, comments, or suggestions about this book in [liveBook's Discussion Forum](#) for the book.

—Luis Serrano, PhD

# *brief contents*

---

- 1 What is machine learning?*
- 2 Types of machine learning*
- 3 Drawing a line close to our points: linear regression*
- 4 Using lines to split our points: the perceptron algorithm*
- 5 A continuous approach to splitting points: logistic regression*
- 6 Using probability to its maximum: naive Bayes algorithm*
- 7 Splitting data by asking questions: decision trees*
- 8 Combining building blocks to gain more power: neural networks*
- 9 Finding boundaries with style: Support vector machines and the kernel method*
- 10 Combining models to maximize results: Ensemble learning*

## **APPENDIX**

- The math behind the algorithms*

# 1

## *What is machine learning?*

**It is common sense, except done by a computer**

**This chapter covers:**

- What is machine learning?
- Is machine learning hard? (Spoiler: No)
- Why you should read this book?
- What will we learn in this book?
- How do humans think, how do machines think, and what does this have to do with machine learning?

**I am super happy to join you in your learning journey!**

Welcome to this book! I'm super happy to be joining you in this journey through understanding machine learning. At a high level, machine learning is a process in which the computer solves problems and makes decisions in a similar way that humans do.

In this book, I want to bring one message to you, and it is: Machine learning is easy! You do not need to have a heavy math knowledge or a heavy programming background to understand it. What you need is common sense, a good visual intuition, and a desire to learn and to apply these methods to anything that you are passionate about and where you want to make an improvement in the world. I've had an absolute blast writing this book, as I love understanding these topics more and more, and I hope you have a blast reading it and diving deep into machine learning!

**Machine learning is everywhere, and you can do it.**

Machine learning is everywhere. This statement seems to be more true every day. I have a hard time imagining a single aspect of life that cannot be improved in some way or another by

machine learning. Anywhere there is a job that requires repetition, that requires looking at data and gathering conclusions, machine learning can help. Especially in the last few years, where computing power has grown so fast, and where data is gathered and processed pretty much anywhere. Just to name a few applications of machine learning: recommendation systems, image recognition, text processing, self-driving cars, spam recognition, anything. Maybe you have a goal or an area in which you are making, or want to make an impact on. Very likely, machine learning can be applied to this field, and hopefully that brought you to this book. So, let's find out together!

## 1.1 Why this book?

### We play the music of machine learning; the formulas and code come later.

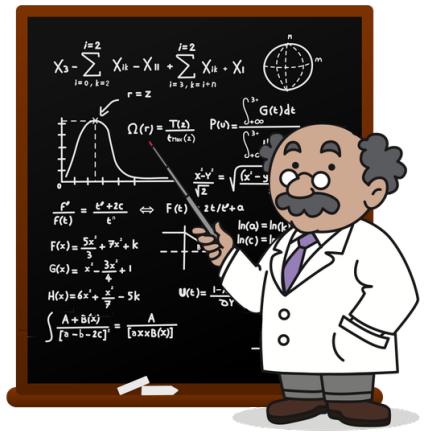
Most of the times, when I read a machine learning book or attend a machine learning lecture, I see either a sea of complicated formulas, or a sea of lines of code. For a long time, I thought this was machine learning, and it was only reserved for those who had a very solid knowledge of both.

I try to compare machine learning with other subjects, such as music. Musical theory and practice are complicated subjects. But when we think of music, we do not think of scores and scales, we think of songs and melodies. And then I wondered, is machine learning the same? Is it really just a bunch of formulas and code, or is there a melody behind that?

With this in mind, I embarked in a journey for understanding the melody of machine learning. I stared at formulas and code for months, drew many diagrams, scribbled drawings on napkins with my family, friends, and colleagues, trained models on small and large datasets, experimented, until finally some very pretty mental pictures started appearing. But it doesn't have to be that hard for you. You can learn more easily without having to deal with the math from the start. Especially since the increasing sophistication of ML tools removes much of the math burden. My goal with this book is to make machine learning fully understandable to every human, and this book is a step on that journey, that I'm very happy you're taking with me!



Music



Machine learning

Figure 1.1. Music is not only about scales and notes. There is a melody behind all the technicalities.

In the same way, machine learning is not about formulas and code.

There is also a melody, and in this book we sing it.

## 1.2 Is machine learning hard?

No.

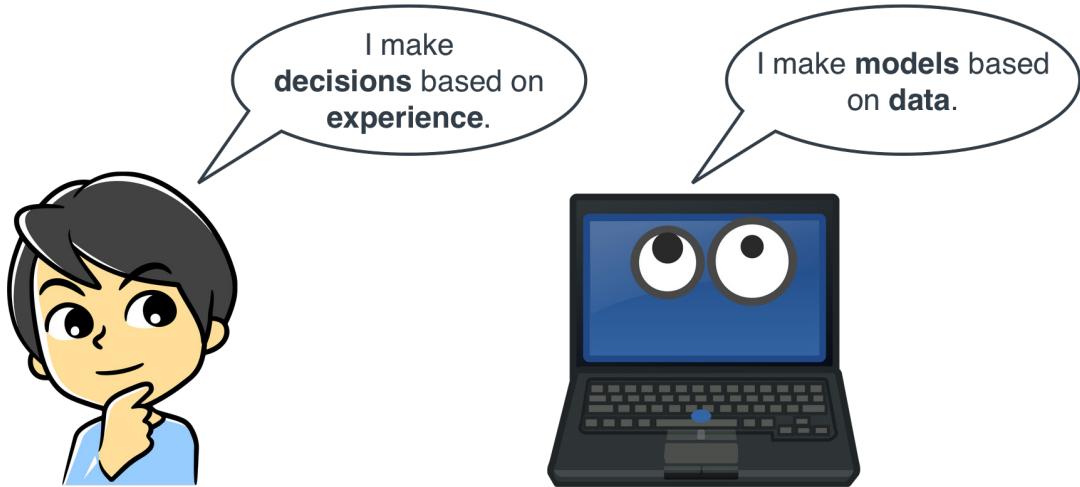
Machine learning requires imagination, creativity, and a visual mind. This is all. It helps a lot if we know mathematics, but the formulas are not required. It helps if we know how to code, but nowadays, there are many packages and tools that help us use machine learning with minimal coding. Each day, machine learning is more available to everyone in the world. All you need is an idea of how to apply it to something, and some knowledge about how to handle data. The goal of this book is to give you this knowledge.

## 1.3 But what exactly is machine learning?

Once upon a time, if we wanted to make a computer perform a task, we had to write a program, namely, a whole set of instructions for the computer to follow. This is good for simple tasks, but how do we get a computer to, for example, identify what is on an image? For example, is there a car on it, is there a person on it. For these kind of tasks, all we can do is give the computer lots of images, and make it learn attributes about them, that will help it recognize them. This is machine learning, it is teaching computers how to do something by experience, rather than by instructions. It is the equivalent of when, as humans, we take decisions based on our intuition, which is based on previous experience. In a way, machine

learning is about teaching the computer how to think like a human. Here is how I define machine learning in the most concise way:

Machine learning is common sense, except done by a computer.



**Figure 1.2. Machine learning is about computers making decisions based on experience.**

In the same way that humans make decisions based on previous experiences, computers can make decisions based on previous data. The rules computers use to make decisions are called models.

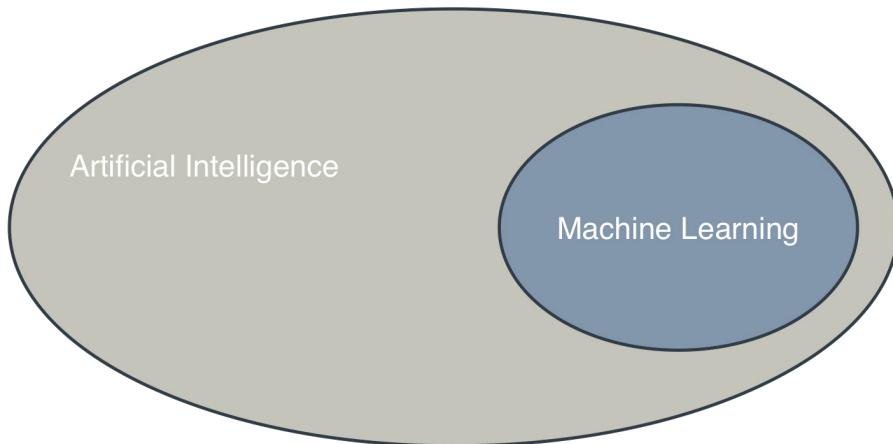
### **Not a huge fan of formulas? You are in the right place**

In most machine learning books, each algorithm is explained in a very formulaic way, normally with an error function, another formula for the derivative of the error function, and a process that will help us minimize this error function in order to get to the solution. These are the descriptions of the methods that work well in the practice, but explaining them with formulas is the equivalent of teaching someone how to drive by opening the hood and frantically pointing at different parts of the car, while reading their descriptions out of a manual. This doesn't show what really happens, which is, the car moves forward when we press the gas pedal, and stops when we hit the breaks. In this book, we study the algorithms in a different way. We do not use error functions and derivatives. Instead, we look at what is really happening with our data, and how is it that we are modeling it.

Don't get me wrong, I think formulas are wonderful, and when needed, we won't shy away from them. But I don't think they form the big picture of machine learning, and thus, we go over the algorithms in a very conceptual way that will show us what really is happening in machine learning.

### 1.3.1 What is the difference between artificial intelligence and machine learning?

First things first, machine learning is a part of artificial intelligence. So anytime we are doing machine learning, we are also doing artificial intelligence.



**Figure 1.3. Machine learning is a part of artificial intelligence.**

I think of artificial intelligence in the following way:

Artificial intelligence encompasses all the ways in which a computer can make decisions.

When I think of how to teach the computer to make decisions, I think of how we as humans make decisions. There are mainly two ways we use to make most decisions:

1. By using reasoning and logic.
2. By using our experience.

Both of these are mirrored by computers, and they have a name: *Artificial intelligence*. Artificial intelligence is the name given to the process in which the computer makes decisions, mimicking a human. So in short, points 1 and 2 form artificial intelligence.

Machine learning, as we stated before, is when we only focus on point 2. Namely, when the computer makes decisions based on experience. And experience has a fancy term in computer lingo: *data*. Thus, machine learning is when the computer makes decisions, based on previous data. In this book, we focus on point 2, and study many ways in which machine can learn from data.

A small example would be how Google maps finds a path between point A and point B. There are several approaches, for example the following:

1. Looking into all the possible roads, measuring the distances, adding them up in all possible ways, and finding which combination of roads gives us the shortest path between points A and B.
2. Watching many cars go through the road for days and days, recording which cars get there in less time, and finding patterns on what their routes were.

As you can see, approach 1 uses logic and reasoning, whereas approach 2 uses previous data. Therefore, approach 2 is machine learning. Approaches 1 and 2 are both artificial intelligence.

### **1.3.2 What about deep learning?**

Deep learning is arguably the most commonly used type of machine learning. The reason is simply that it works really well. If you are looking at any of the cutting edge applications, such as image recognition, language generation, playing Go, or self driving cars, very likely you are looking at deep learning in some way or another. But what exactly is deep learning? This term applies to every type of machine learning that uses *Neural Networks*. Neural networks are one type of algorithm, which we learn in Chapter 5.

So in other words, deep learning is simply a part of machine learning, which in turn is a part of artificial intelligence. If this book was about vehicles, then AI would be motion, ML would be cars, and deep learning (DL) would be Ferraris.

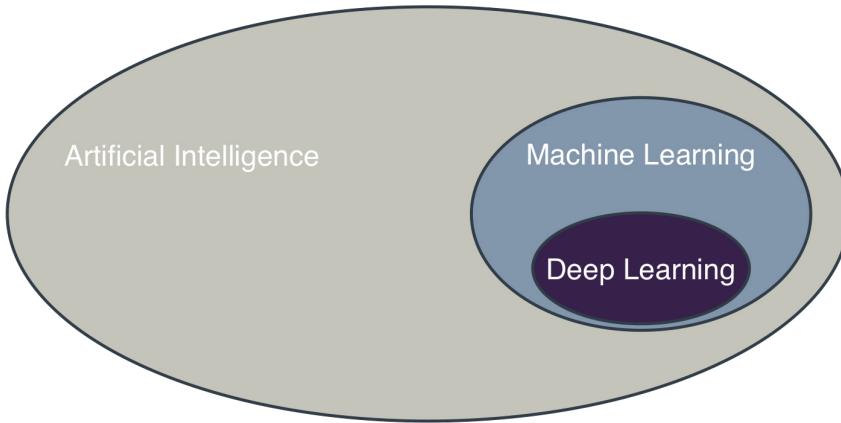


Figure 1.4. Deep learning is a part of machine learning.

## **1.4 Humans use the remember-formulate-predict framework to make decisions (and so can machines!)**

How does the computer make decisions based on previous data? For this, let's first see the process of how humans make decisions based on experience. And this is what I call the

*remember-formulate-predict framework.* The goal of machine learning is to teach computers how to think in the same way, following the same framework.

### 1.4.1 How do humans think?

When we humans need to make a decision based on our experience, we normally use the following framework:

1. We **remember** past situations that were similar.
2. We **formulate** a general rule.
3. We use this rule to **predict** what will happen if we take a certain decision.

For example, if the question is: "Will it rain today?", the process to make a guess will be the following:

1. We **remember** that last week it rained most of the time.
2. We **formulate** that in this place, it rains most of the time.
3. We **predict** that today it will rain.

We may be right or wrong, but at least, we are trying to make an accurate prediction.

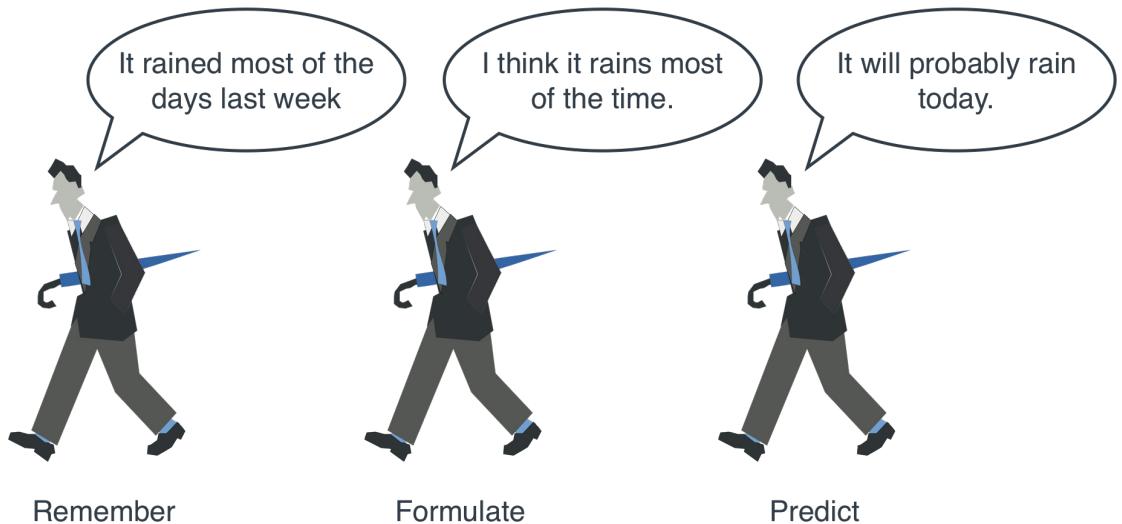


Figure 1.2. The remember-formulate-predict framework.

Let us put this in practice with an example.

#### Example 1: An annoying email friend

Here is an example. We have a friend called Bob, who likes to send us a lot of email. In particular, a lot of his emails are spam, in the form of chain letters, and we are starting to get

a bit annoyed at him. It is Saturday, and we just got a notification of an email from him. Can we guess if it is spam or not without looking at the email?

**SPAM AND HAM** Spam is the common term used for junk or unwanted email, such as chain letters, promotions, and so on. The term comes from a 1972 Monty Python sketch in which every item in the menu of a restaurant contained spam as an ingredient. Among software developers, the term 'ham' is used to refer to non-spam emails. I use this terminology in this book.

For this, we use the remember-formulate-predict method.

First let us **remember**, say, the last 10 emails that we got from Bob. We remember that 4 of them were spam, and the other 6 were ham. From this information, we can **formulate** the following rule:

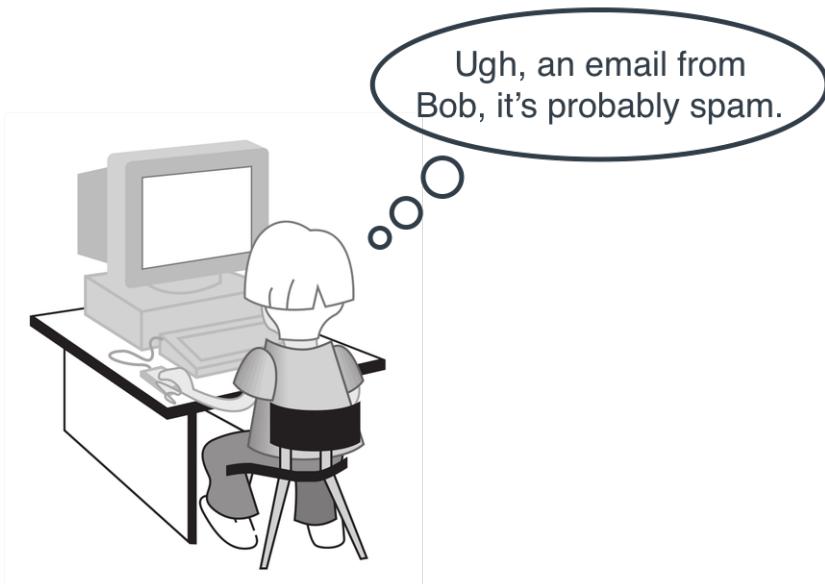
**Rule 1:** 4 out of every 10 emails that Bob sends us are spam.

This rule will be our *model*. Note, this rule does not need to be true. It could be outrageously wrong. But given our data, it is the best that we can come up to, so we'll live with it. Later in this book, we learn how to evaluate models and improve them when needed. But for now, we can live with this.

Now that we have our rule, we can use it to **predict** if the email is spam or not. If 40 out of 10 of the emails that Bob sends us are spam, then we can assume that this new email is 40% likely to be spam, and 60% likely to be ham. Therefore, it's a little safer to think that the email is ham. Therefore, we predict that the email is not spam.

Again, our prediction may be wrong. We may open the email and realize it is spam. But we have made the prediction *to the best of our knowledge*. This is what machine learning is all about.

But you may be thinking, 6 out of 10 is not enough confidence on the email being spam or ham, can we do better? Let's try to analyze the emails a little more. Let's see when Bob sent the emails to see if we find a pattern.



**Figure 1.3.** A very simple machine learning model.

### Example 2: A seasonal annoying email friend

Let us look more carefully at the emails that Bob sent us in the previous month. Let's look at what day he sent them. Here are the emails with dates, and information about being spam or ham:

- Monday: Ham
- Tuesday: Ham
- Saturday: Spam
- Sunday: Spam
- Sunday: Spam
- Wednesday: Ham
- Friday: Ham
- Saturday: Spam
- Tuesday: Ham
- Thursday: Ham

Now things are different. Can you see a pattern? It seems that every email Bob sent during the week, is ham, and every email he sent during the weekend is spam. This makes sense, maybe during the week he sends us work email, whereas during the weekend, he has time to send spam, and decides to roam free. So, we can **formulate** a more educated rule:

**Rule 2:** Every email that Bob sends during the week is ham, and during the weekend is spam.

And now, let's look at what day it is today. If it is Saturday, and we just got an email from him, then we can **predict** with great confidence that the email he sent is spam. So we make this prediction, and without looking, we send the email to the trash can.

Let's give things names, in this case, our prediction was based on a feature. The feature was the day of the week, or more specifically, it being a weekday or a day in the weekend. You can imagine that there are many more features that could indicate if an email is spam or ham. Can you think of some more? In the next paragraphs we'll see a few more features.

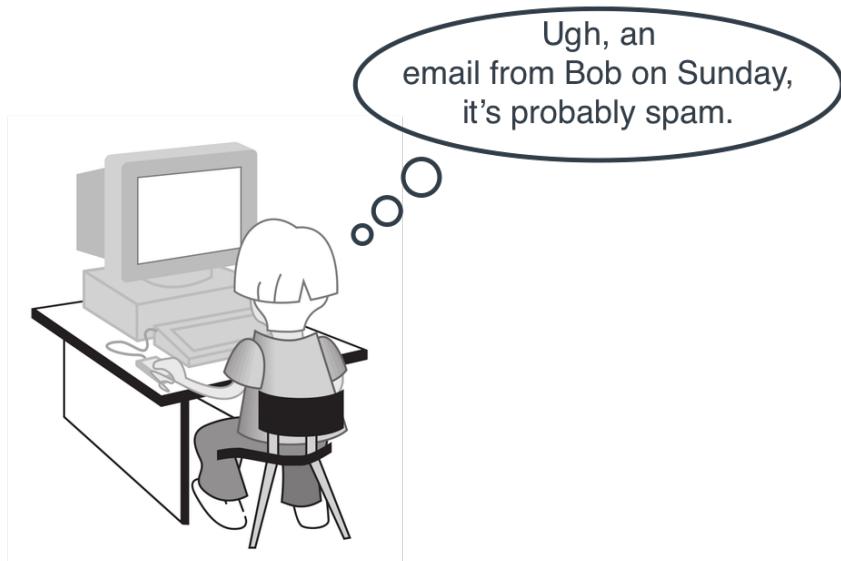


Figure 1.4. A slightly more complex machine learning model, done by a human.

### Example 3: Things are getting complicated!

Now, let's say we continue with this rule, and one day we see Bob in the street, and he says "Why didn't you come to my birthday party?" We have no idea what he is talking about. It turns out last Sunday he sent us an invitation to his birthday party, and we missed it! Why did we miss it, because he sent it on the weekend. It seems that we need a better model. So let's go back to look at Bob's emails, in the following table, this is our **remember** step. Now let's see if you can help me find a pattern.

- 1KB: Ham
- 12KB: Ham
- 16KB: Spam

- 20KB: Spam
- 18KB: Spam
- 3KB: Ham
- 5KB: Ham
- 25KB: Spam
- 1KB: Ham
- 3KB: Ham

What do we see? It seems that the large emails tend to be spam, while the smaller ones tend to not be spam. This makes sense, since maybe the spam ones have a large attachment.

So, we can **formulate** the following rule:

**Rule 3:** Any email larger of size 10KB or more is spam, and any email of size less than 10KB is ham.

So now that we have our rule, we can make a **prediction**. We look at the email we received today, and the size is 19KB. So we conclude that it is spam.

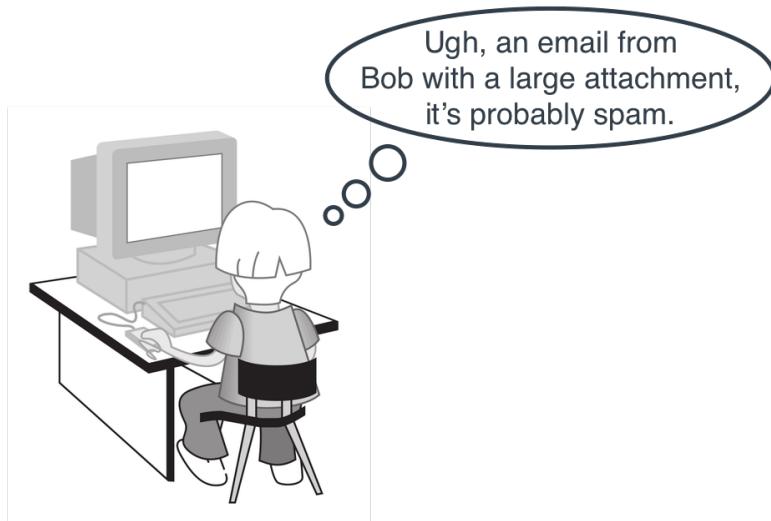


Figure 1.5. Another slightly more complex machine learning model, done by a human.

Is *this* the end of the story? I don't know...

#### Example 4: More?

Our two classifiers were good, since they rule out large emails and emails sent on the weekends. Each one of them uses exactly one of these two features. But what if we wanted a rule that worked with both features? Rules like the following may work:

**Rule 4:** If an email is larger than 10KB or it is sent on the weekend, then it is classified as spam. Otherwise, it is classified as ham.

**Rule 5:** If the email is sent during the week, then it must be larger than 15KB to be classified as spam. If it is sent during the weekend, then it must be larger than 5KB to be classified as spam. Otherwise, it is classified as ham.

Or we can even get much more complicated.

**Rule 6:** Consider the number of the day, where Monday is 0, Tuesday is 1, Wednesday is 2, Thursday is 3, Friday is 4, Saturday is 5, and Sunday is 6. If we add the number of the day and the size of the email (in KB), and the result is 12 or more, then the email is classified as spam. Otherwise, it is classified as ham.

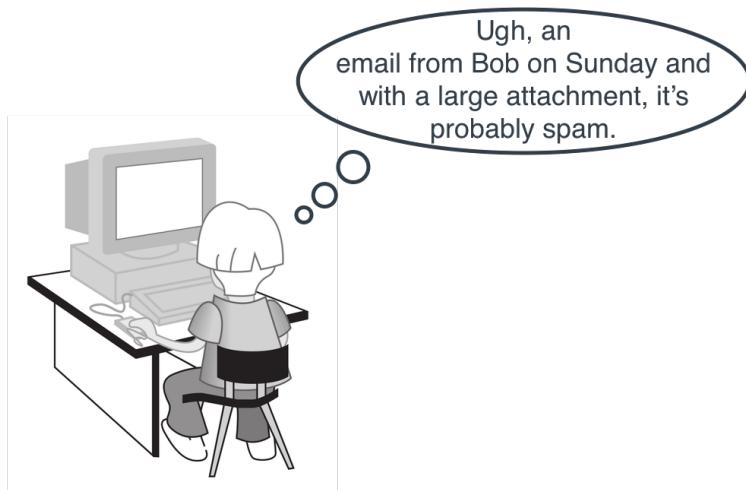


Figure 1.6. An even more complex machine learning model, done by a human.

All of these are valid rules. And we can keep adding layers and layers of complexity. Now the question is, which is the best rule? This is where we may start needing the help of a computer.

### 1.4.2 How do machines think?

The goal is to make the computer think the way we think, namely, use the remember-formulate-predict framework. In a nutshell, here is what the computer does in each of the steps.

**Remember:** Look at a huge table of data.

**Formulate:** Go through many rules and formulas, and check which one fits the data best.

**Predict:** Use the rule to make predictions about future data.

This is not much different than what we did in the previous section. The great advancement here is that the computer can try building rules such as rules 4, 5, or 6, trying

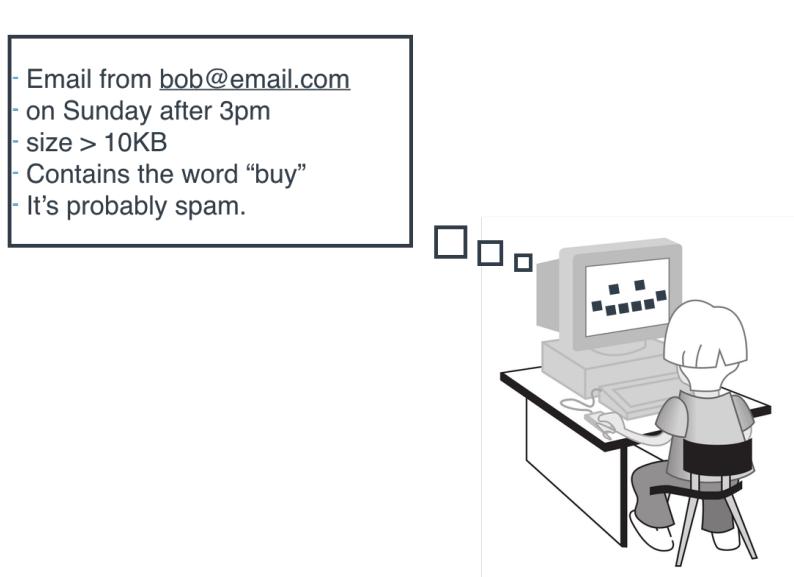
different numbers, different boundaries, and so on, until finding one that works best for the data. It can also do it if we have lots of columns. For example, we can make a spam classifier with features such as the sender, the date and time of day, the number of words, the number of spelling mistakes, the appearances of certain words such as "buy", or similar words. A rule could easily look as follows:

**Rule 7:**

- If the email has two or more spelling mistakes, then it is classified as spam.
  - Otherwise, if it has an attachment larger than 20KB, it is classified as spam.
    - Otherwise, if the sender is not in our contact list, it is classified as spam.
    - Otherwise, if it has the words "buy" and "win", it is classified as spam.
    - Otherwise, it is classified as ham.

Or even more mathematical, such as:

**Rule 8: If**

$$\begin{aligned} & (\text{size}) + 10 \times (\text{number of spelling mistakes}) - (\text{number of appearances of the word 'mom'}) \\ & + 4 \times (\text{number of appearances of the word 'buy'}) > 10, \\ & \text{then we classify the message as spam. Otherwise we do not.} \end{aligned}$$


**Figure 1.7. A much more complex machine learning model, done by a computer.**

Now the question is, which is the best rule? The quick answer is: The one that fits the data best. Although the real answer is: The one that generalizes best to new data. At the end of the day, we may end up with a very complicated rule, but the computer can formulate it and use it to make predictions very quickly. And now the question is: How to build the best model? That is exactly what this book is about.

## 1.5 What is this book about?

Good question. The rules 1-8 above, are examples of machine learning models, or classifiers. As you saw, these are of different types. Some use an equation on the features to make a prediction. Others use a combination of if statements. Others will return the answer as a probability. Others may even return the answer as a number! In this book, we study the main algorithms of what we call predictive machine learning. Each one has its own style, way to interpret the features, and way to make a prediction. In this book, each chapter is dedicated to one different type of model.

This book provides you with a solid framework of predictive machine learning. To get the most out of this book, you should have a visual mind, and a basis of mathematics, such as graphs of lines, equations, and probability. It is very helpful (although not mandatory) if you know how to code, specially in Python, as you will be given the opportunity to implement and apply several models in real datasets throughout the book. After reading this book, you will be able to do the following:

- Describe the most important algorithms in predictive machine learning and how they work, including linear and logistic regression, decision trees, naive Bayes, support vector machines, and neural networks.
- Identify what are their strengths and weaknesses, and what parameters they use.
- Identify how these algorithms are used in the real world, and formulate potential ways to apply machine learning to any particular problem you would like to solve.
- How to optimize these algorithms, compare them, and improve them, in order to build the best machine learning models we can.

If you have a particular dataset or problem in mind, we invite you to think about how to apply each of the algorithms to your particular dataset or problem, and to use this book as a starting point to implement and experiment with your own models.

I am super excited to start this journey with you, and I hope you are as excited!

## 1.6 Summary

- Machine learning is easy! Anyone can do it, regardless of their background, all that is needed is a desire to learn, and great ideas to implement!
- Machine learning is tremendously useful, and it is used in most disciplines. From science to technology to social problems and medicine, machine learning is making an impact, and will continue making it.
- Machine learning is common sense, done by a computer. It mimics the ways humans

think in order to make decisions fast and accurately.

- Just like humans make decisions based on experience, computers can make decisions based on previous data. This is what machine learning is all about.
- Machine learning uses the remember-formulate-predict framework, as follows:
  - **Remember:** Use previous data.
  - **Formulate:** Build a model, or a rule, for this data.
  - **Predict:** Use the model to make predictions about future data.

# 2

## *Types of machine learning*

### **This chapter covers:**

- Three main different types of machine learning.
- The difference between labelled and unlabelled data.
- What supervised learning is and what it's useful for.
- The difference between regression and classification, and what are they useful for.
- What unsupervised learning is and what it's useful for.
- What reinforcement learning is and what it's useful for.

As we learned in Chapter 1, machine learning is common sense, but for a computer. It mimics the process in which humans make decisions based on experience, by making decisions based on previous data. Of course, this is challenging for computers, as all they do is store numbers and do operations on them, so programming them to mimic human level of thought is difficult. Machine learning is divided into several branches, and they all mimic different types of ways in which humans make decisions. In this chapter, we overview some of the most important of these branches.

ML has applications in many many fields. Can you think of some fields in which you can apply machine learning? Here is a list of some of my favorites:

- Predicting housing prices based on their size, number of rooms, location, etc.
- Predicting the stock market based on other factors of the market, and yesterday's price.
- Detecting spam or non-spam e-mails based on the words of the e-mail, the sender, etc.
- Recognizing images as faces, animals, etc., based on the pixels in the image.
- Processing long text documents and outputting a summary.
- Recommending videos or movies to a user (for example YouTube, Netflix, etc.).
- Chatbots that interact with humans and answer questions.

- Self driving cars that are able to navigate a city.
- Diagnosing patients as sick or healthy.
- Segmenting the market into similar groups based on location, acquisitive power, interests, etc.
- Playing games like chess or Go.

Try to imagine how we could use machine learning in each of these fields. Some applications look similar. For example, we can imagine that predicting housing prices and predicting stock prices must use similar techniques. Likewise, predicting if email is spam and predicting if credit card transactions are legitimate or fraudulent may also use similar techniques. What about grouping users of an app based on similarity? That sounds very different than predicting housing prices, but could it be that it is done in a similar way as we group newspaper articles by topic? And what about playing chess? That sounds very different than predicting if an email is spam. But it sounds similar to playing Go.

Machine learning models are grouped into different types, according to the way they operate. The main three families of machine learning models are

- supervised learning,
- unsupervised learning, and
- reinforcement learning.

In this chapter, we overview them all. However, in this book, we only cover supervised learning, as it is the most natural one to start learning, and arguably the most commonly used. We encourage you to look up the other types in the literature and learn about them too, as they are all very interesting and useful!

**(Sidebar) Recommended sources: (not sure how to write this)**

1. Grokking Deep Reinforcement Learning, by Miguel Morales (Manning)
2. UCL course on reinforcement learning, by David Silver (<http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html>)
3. Deep Reinforcement Learning Nanodegree Program, by Udacity. (<https://www.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893>)

## 2.1 What is the difference between labelled and unlabelled data?

### Actually, what is data?

Let's first establish a clear definition of what we mean by data. Data is simply information. Any time we have a table with information, we have data. Normally, each row is a data point. Let's say, for example, that we have a dataset of pets. In this case, each row represents a different pet. Each pet is described then, by certain features.

### Ok. And what are features?

Features are simply the columns of the table. In our pet example, the features may be size, name, type, weight, etc. This is what describes our data. Some features are special, though, and we call them *labels*.

### Labels?

This one is a bit less obvious, and it depends on the context of the problem we are trying to solve. Normally, if we are trying to predict a feature based on the others, that feature is the label. If we are trying to predict the type of pet we have (for example cat or dog), based on information on that pet, then that is the label. If we are trying to predict if the pet is sick or healthy based on symptoms and other information, then that is the label. If we are trying to predict the age of the pet, then the age is the label.

So now we can define two very important things, labeled and unlabeled data.

**Labeled data:** Data that comes with a label.

**Unlabeled data:** Data that comes without a label.

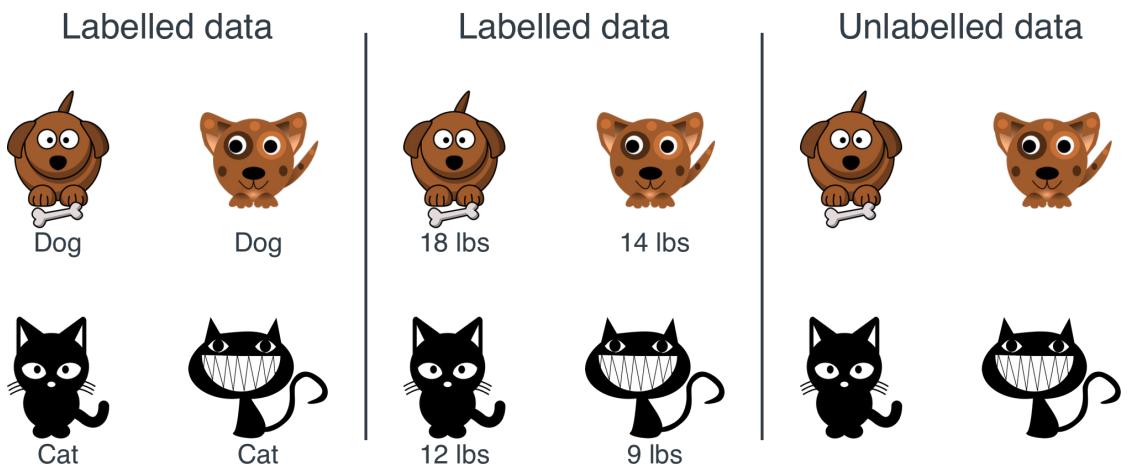


Figure 2.1. Labeled data is data that comes with a tag, like a name, a type, or a number. Unlabeled data is data that comes with no tag.

### So what is then, supervised and unsupervised learning?

Clearly, it is better to have labeled data than unlabeled data. With a labeled dataset, we can do much more. But there are still many things that we can do with an unlabeled dataset.

The set of algorithms in which we use a labeled dataset is called *supervised learning*. The set of algorithms in which we use an unlabeled dataset, is called *unsupervised learning*. This is what we learn next.

## 2.2 What is supervised learning?

Supervised learning is the type of machine learning you find in the most common applications nowadays, including image recognition, various forms of text processing, recommendation systems, and many more. As we stated in the previous section, it is a type of predictive machine learning in which the data comes with labels, where the label is the target we are interested in predicting.

In the example on Figure 2.1, where the dataset is formed by images of dogs and cats, and the labels in the image are 'dog' and 'cat', the machine learning model would simply use previous data in order to predict the label of new data points. This means, if we bring in a new image *without* a label, the model would guess if the image is of a dog or a cat, thus predicting the label of the data point.

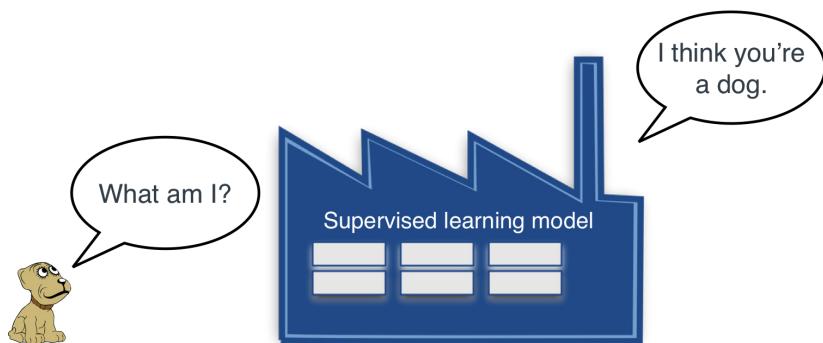


Figure 2.2. A supervised learning model predicts the label of a new data point.

If you recall Chapter 1, the framework we learned for making a decision was Remember-Formulate-Predict. This is precisely how supervised learning works. The model first **remembers** the dataset of dogs and cats, then **formulates** a model, or a rule for what is a dog and what is a cat, and when a new image comes in, the model makes a **prediction** about what the label of the image is, namely, is it a dog or a cat.

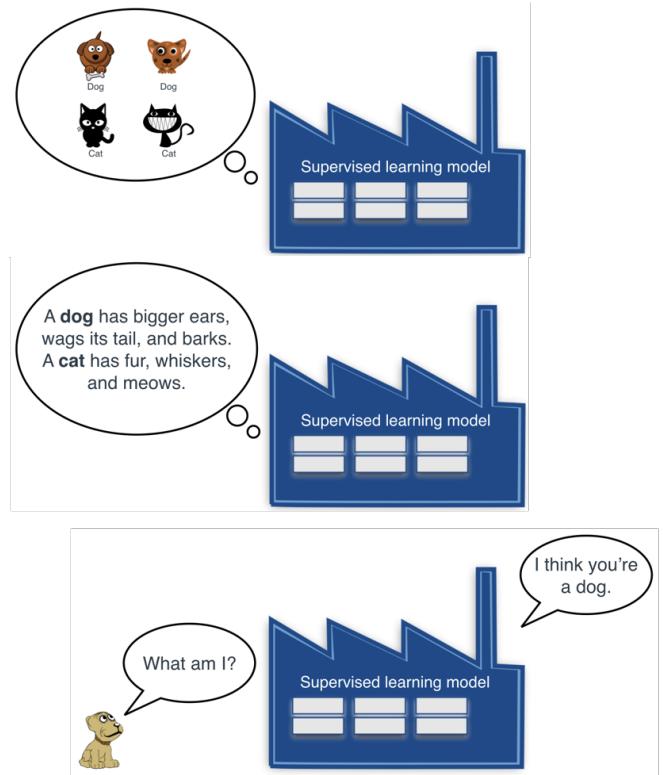


Figure 2.3. Supervised learning follows the Remember-Formulate-Predict framework from Chapter 1.

Now, notice that in Figure 2.1, we have two types of datasets, one in which the labels are numbers (the weight of the animal), and one in which the labels are states, or classes (the type of animal, namely cat or dog). This gives rise to two types of supervised learning models.

**Regression models:** These are the types of models that predict a **number**, such as the weight of the animal.

**Classification models:** These are the types of models that predict a **state**, such as the type of animal (cat or dog).

We call the output of a regression model *continuous*, since the prediction can be any real value, picked from a continuous interval. We call the output of a classification model *discrete*, since the prediction can be a value from a finite list. An interesting fact is that the output can be more than two states. If we had more states, say, a model that predicts if a picture is of a dog, a cat, or a bird, we can still use a discrete model. These models are called multivariate discrete models. There are classifiers with many states, but it must always be a finite number.

Let's look at two examples of supervised learning models, one regression and one classification:

Example 1 (regression), housing prices model: In this model, each data point is a house. The label of each house is its price. Our goal is, when a new house (data point) comes in the market, we would like to predict its label, namely, its price.

Example 2 (classification), email spam detection model: In this model, each data point is an email. The label of each email is either spam or ham. Our goal is, when a new email (data point) comes into our inbox, we would like to predict its label, namely, if it is spam or ham.

You can see the difference between models 1 and 2.

- Example 1, the housing prices model, is a model that can return many numbers, such as \$100, \$250,000, or \$3,125,672. Thus it is a *regression* model.
- Example 2, the spam detection model, on the other hand, can only return two things: spam or ham. Thus it is a *classification* model.

Let's elaborate some more on regression and classification.

### **2.2.1 Regression models predict numbers**

As we mentioned previously, regression models are those that predict a number. This number is predicted from the features. In the housing example, the features can be the size of the house, the number of rooms, the distance to the closest school, the crime rate in the neighborhood, etc.

Other places where one can use regression models are the following:

- Stock market: Predicting the price of a certain stock based on other stock prices, and other market signals.
- Medicine: Predicting the expected lifespan of a patient, or the expected recovery time, based on symptoms and the medical history of the patient.
- Sales: Predicting the expected amount of money a customer will spend, based on the client's demographics and past purchase behavior.
- Video recommendations: Predicting the expected amount of time a user will watch a video, based on the user's demographics and past interaction with the site.

The most common method used for regression is *linear regression*, which is when we use linear functions (basically lines) to make our predictions based on the features. We study linear regression in Chapter 3.

### **2.2.2 Classification models predict a state**

Classification models are those that predict a state, from a finite set of states. The most common ones predict a 'yes' or a 'no', but there are many models which use a larger set of states. The example we saw in Figure 2.3 is of classification, as it predicts the type of the pet, namely, 'cat' or 'dog'.

In the email spam recognition example, the state of the email (namely, is it spam or not) is predicted from the features. In this case, the features of the email are the words on it, the number of spelling mistakes, the sender, and many others.

Another very common example of classification is image recognition. The most popular image recognition models take as an input the pixels in the image, and output a prediction of what the image most likely depicts. Two of the most famous datasets for image recognition are MNIST and CIFAR-10. MNIST is formed by around 70,000 images of handwritten digits, which are classified as the digits 0-9. These images come from a combination of sources, including the American Census Bureau, and handwritten digits taken from American high school students. It can be found in the following link: <http://yann.lecun.com/exdb/mnist/>. CIFAR-10 is made of 60,000 32 by 32 colored images of different things. These are classified as 10 different classes (thus the 10 in the name), namely airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. This database is maintained by the Canadian Institute For Advanced Research (CIFAR), and can be found in the following link: <https://www.cs.toronto.edu/~kriz/cifar.html>.

Other places where one can use classification models are the following:

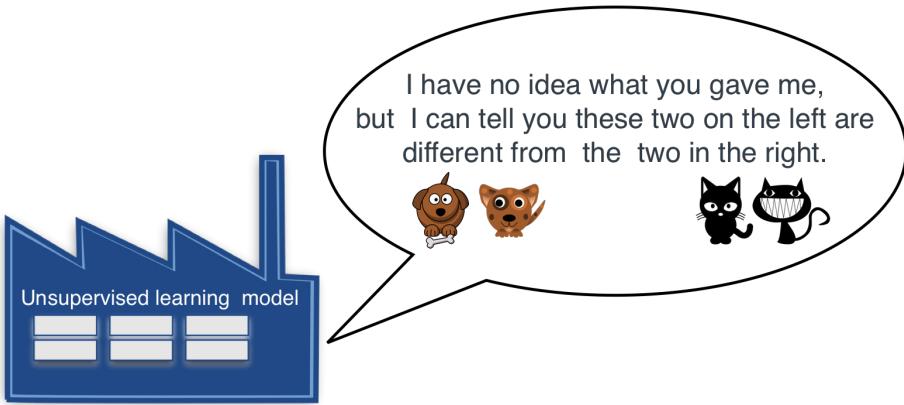
- Sentiment analysis: Predicting if a movie review is positive or negative, based on the words in the review.
- Website traffic: Predicting if a user will click on a link or not, based on the user's demographics and past interaction with the site.
- Social media: Predicting if a user will befriend or interact with another user or not, based on their demographics, history, and friends in common.

The bulk of this book talks about classification models. In chapters 3-x, we talk about classification models in the context of logistic regression, decision trees, naive Bayes, support vector machines, and the most popular classification models nowadays: neural networks.

## 2.3 What is unsupervised learning?

Unsupervised learning is also a very common type of machine learning. It differs from supervised learning in that the data has no labels. What is a dataset with no labels, you ask? Well, it is a dataset with only features, and no target to predict. For example, if our housing dataset had no prices, then it would be an unlabeled dataset. If our emails dataset had no labels, then it would simply be a dataset of emails, where 'spam' and 'no spam' is not specified.

So what could you do with such a dataset? Well, a little less than with a labelled dataset, unfortunately, since the main thing we are aiming to predict is not there. However, we can still extract a lot of information from an unlabelled dataset. Here is an example, let us go back to the cats and dogs example in Figure 2.1. If our dataset has no labels, then we simply have a bunch of pictures of dogs and cats, and we do not know what type of pet each one represents. Our model can still tell us if two pictures of dogs are similar to each other, and different to a picture of a cat. Maybe it can group them in some way by similarity, even without knowing what each group represents.



**Figure 2.4.** An unsupervised learning model can still extract information from data, for example, it can group similar elements together.

And the branch of machine learning that deals with unlabelled datasets is called *unsupervised machine learning*. As a matter of fact, even if the labels are there, we can still use unsupervised learning techniques on our data, in order to preprocess it and apply supervised learning methods much more effectively.

The two main branches of unsupervised learning are clustering and dimensionality reduction. They are defined as follows.

**Clustering:** This is the task of grouping our data into clusters based on similarity. (This is what we saw in Figure 2.4.)

**Dimensionality reduction:** This is the task of simplifying our data and describing it with fewer features, without losing much generality.

Let's study them in more detail.

### 2.3.1 Clustering algorithms split a dataset into similar groups

As we stated previously, clustering algorithms are those that look at a dataset, and split it into similar groups

So let's go back to our two examples. In the first one, we have a dataset with information about houses, but no prices. What could we do? Here is an idea: we could somehow group them into similar houses. We could group them by location, by price, by size, or by a combination of these factors. This is called *clustering*. Clustering is a branch of unsupervised machine learning which consists of grouping the elements in our dataset into clusters that are similar. Could we do that with other datasets?

Let's look at our second example, the dataset of emails. Because the dataset is unlabeled, we don't know if each email is spam or not. However, we can still apply some clustering to our dataset. A clustering algorithm will return our emails split into, say, 4 or 5 different categories, based on different features such as words in the message, sender, attachments, types of links

on them, and more. It is then up to a human (or a supervised learning algorithm) to label categories such as 'Personal', 'Social', 'Promotions', and others.

For example, let's say that we have 9 emails, and we want to cluster them into different types. We have, say, the size of the email, and the number of recipients. And the data looks like this, ordered by number of recipients:

Email	Size	Recipients
1	8	1
2	12	1
3	43	1
4	10	2
5	40	2
6	25	5
7	23	6
8	28	6
9	26	7

**Table 2.5. A table of emails with their size and number of recipients.**

To the naked eye, it looks like we could group them by size, where the emails in one group would have 1 or 2 recipients, and the emails in the other group would have 5 or more recipients. We could also try to group them into three groups by size. But you can imagine that as the data gets larger and larger, eyeballing the groups gets harder and harder. What if we plot the data? Let's plot the emails in a graph, where the horizontal axis records the size, and the vertical axis records the number of recipients. We get the following plot.

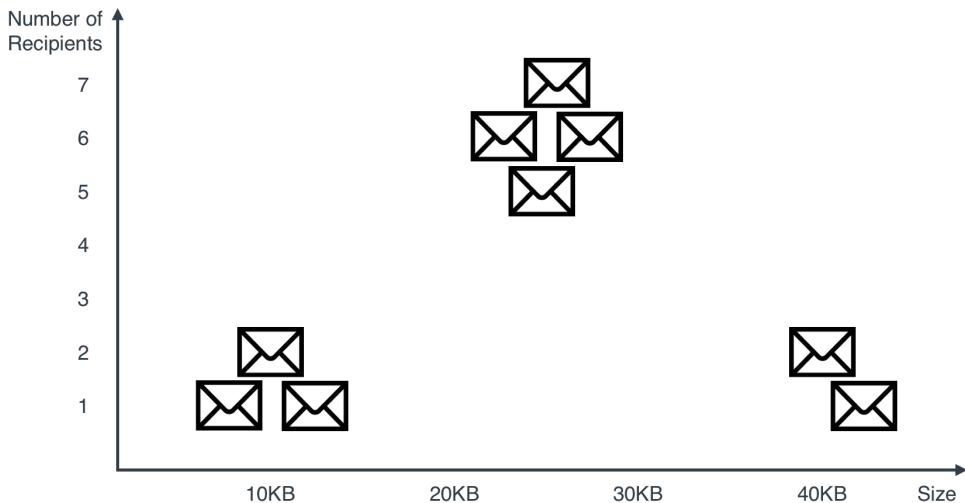


Figure 2.6. A plot of the emails with size on the horizontal axis and number of recipients on the vertical axis. Eyeballing it, it is obvious that there are three distinct types of emails.

In Figure 2.6 we can see three groups, very well defined. We can make each a different category in our inbox. They are the ones we see in Figure 2.7.

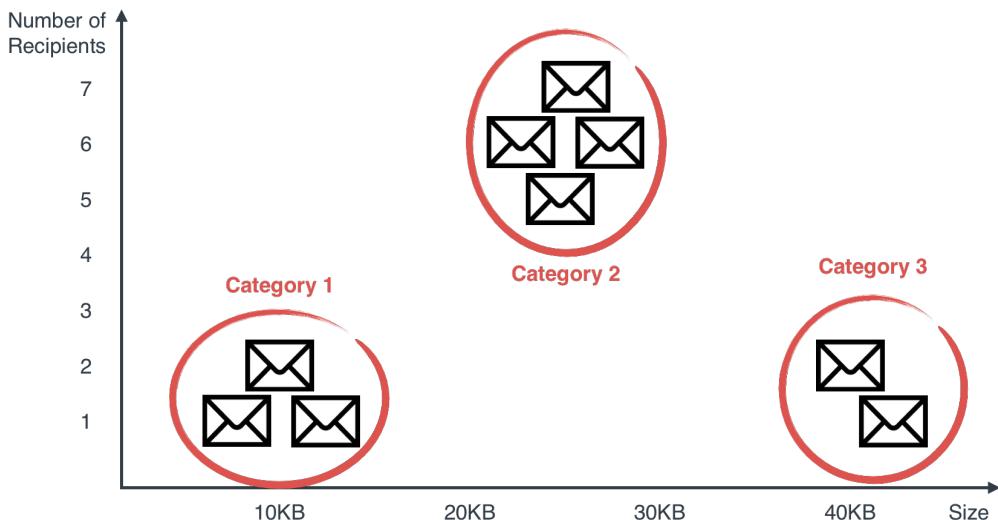


Figure 2.7. Clustering the emails into three categories based on size and number of recipients.

This last step is what clustering is all about. Of course, for us humans, it was very easy to eyeball the three groups once we have the plot. But for a computer, this is not easy. And furthermore, imagine if our data was formed by millions of points, with hundreds or thousands of columns. All of a sudden, we cannot eyeball the data, and clustering becomes hard. Luckily, computers can do these type of clustering for huge datasets with lots of columns.

Other applications of clustering are the following:

- Market segmentation: Dividing customers into groups based on demographic and purchasing (or engagement) behavior, in order to create different marketing strategies for the groups.
- Genetics: Clustering species into groups based on similarity.
- Medical imaging: Splitting an image into different parts in order to study different types of tissue.

### **Unsupervised learning algorithms**

In this book, we don't get to study unsupervised learning. However, I strongly encourage you to study them on your own. Here are some of the most important clustering algorithms out there.

- K-means clustering: This algorithm groups points by picking some random centers of mass, and moving them closer and closer to the points until they are at the right spots.
- Hierarchical clustering: This algorithm starts by grouping the closest points together, and continuing in this fashion, until we have some well defined groups.
- Density-based special clustering (DBSCAN): This algorithm starts grouping points together in points of high density, while leaving the isolated points as noise.

Gaussian mixture models: This algorithm doesn't actually determine if an element belongs to a cluster, but instead gives a breakdown of percentages. For example, if there are three clusters, A, B, and C, then the algorithm could say that a point belongs 60% to group A, 25% to group B, and 15% to group C.

### **2.3.2 Dimensionality reduction simplifies data without losing much information**

Dimensionality reduction is a very useful preprocessing step which we can apply to vastly simplify our data, before applying other techniques. Let's look at the housing example. Let's say that we want to predict the price, and the features are the following:

1. Size.
2. Number of bedrooms.
3. Number of bathrooms.
4. Crime rate in the neighborhood.
5. Distance to the nearest school.

That is five columns of data. What if we wanted a simpler dataset, with fewer columns, but that can portray the information in as faithful a way as possible. Let's do it using common sense. Take a closer look at the five features. Can you see any way to simplify them, maybe to group them into some smaller and more general categories?

After a careful look, maybe you thought the same as I did, which is: The first three features seem similar, and the fourth and fifth also seem similar. The first three are all related to the size of the house, whereas the fourth and fifth are related to the quality of the neighborhood. We could condense them into a big 'size' feature, and a big 'area quality' feature. How do we condense the size features? There are many ways, we could only consider the size, we could add the number of bedrooms and bathrooms, or maybe some linear combination of the three features. How do we condense the neighborhood quality features? Again in many ways, if they are both given by coefficients, we can add them, subtract them, etc. The dimensionality reduction algorithms will find ways that group them, losing as little information as possible, and keeping our data as intact as possible, while managing to simplify it for easier process and storage.

## Dimensionality reduction

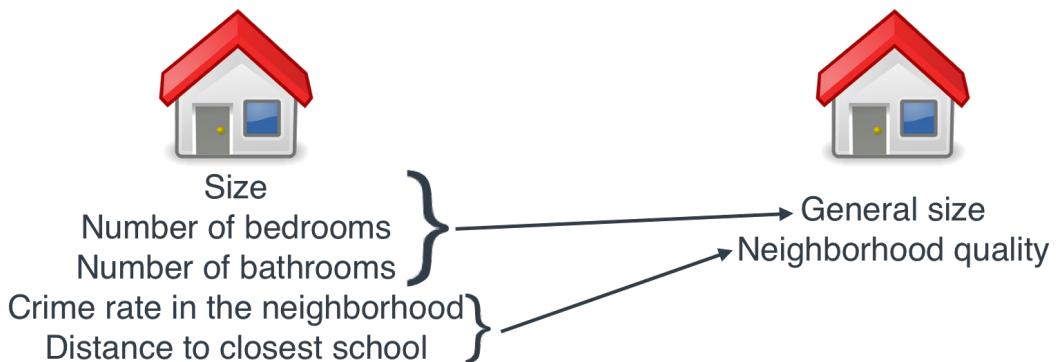


Figure 2.8. Using dimensionality reduction to reduce the number of features in a housing dataset, without losing much information.

Now, why is it called dimensionality reduction, if all we're doing is reducing the number of columns in our data? Well, the fancy word for *number of columns* in data is *dimension*. Think about this, if our data has one column, then each data point is one number. This is the same as if our data set was formed by points in a line, and a line has one dimension. If our data has two columns, then each data point is formed by two numbers. This is like coordinates in a city, where the first number is the street number, and the second number is the avenue. And cities are two dimensional, since they are in a plane (if we imagine that every house has only one floor). Now, what happens when our data has 3 columns? In this case, then each data point is formed by 3 numbers. We can imagine that if every address in our city is a building, then the first and second numbers are the street and avenue, and the third one is the floor in which we live in. This looks like a three-dimensional city. We can keep going. What about four numbers? Well, now we can't really visualize it, but if we could, this would be addresses in a four-

dimensional city, and so on. The best way I can imagine a four dimensional city, is by imagining a table of four columns. And a 100-dimensional city? Simple, a table with 100 columns, in which each person has an address that consists of 100 numbers. The mental picture I have when thinking of higher dimensions is in Figure 2.9.

Therefore, when we went from five dimensions down to two, we reduced our 5-dimensional city into a 2-dimensional city, thus applying dimensionality reduction.

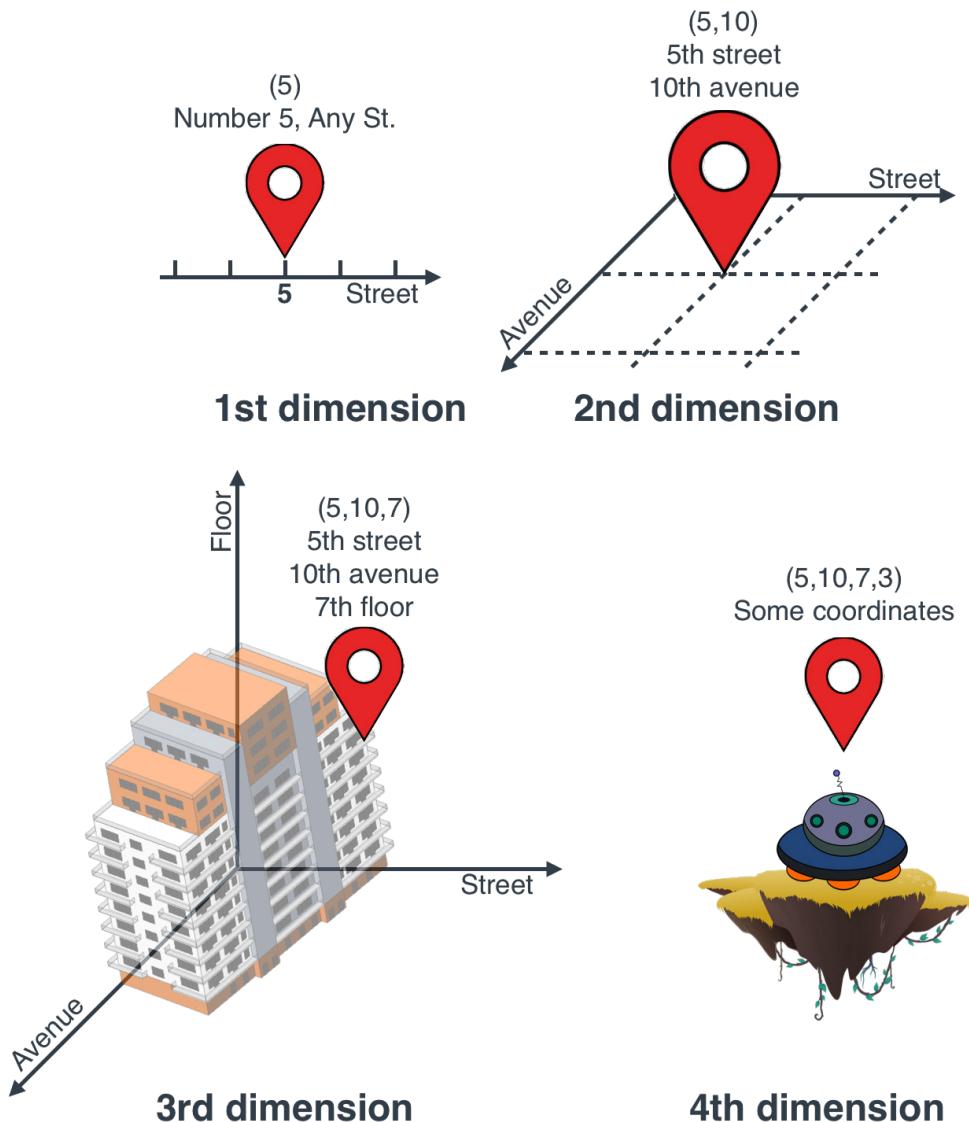


Figure 2.9. How to imagine higher dimensional spaces.

1 dimension is like a street, in which each house only has one number.

2 dimensions is like a flat city, in which each address has two numbers, a street and an avenue.

3 dimensions is like a city with buildings, in which each address has three numbers, a street, an avenue, and a floor.

4 dimensions is like some imaginary place, in which each address has four numbers.

And so on...

### 2.3.3 Matrix factorization and other types of unsupervised learning

It seems that clustering and dimensionality reduction look very different, but in reality they are not so different. If we have a table full of data, each row is a data point, and each column is a feature. Therefore, we can see clustering as a way to group the rows, and dimensionality reduction as a way to group the columns, as figures 2.10 and 2.11 illustrate.

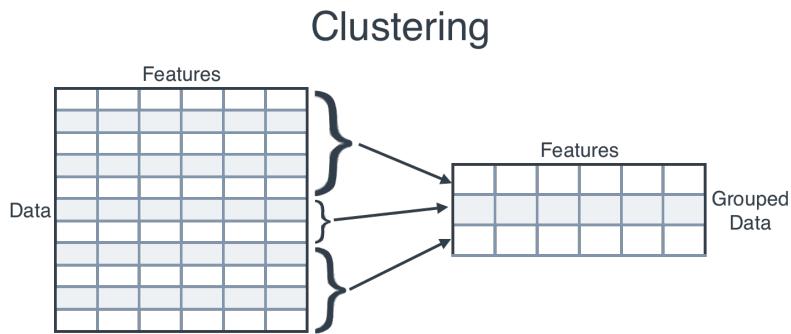
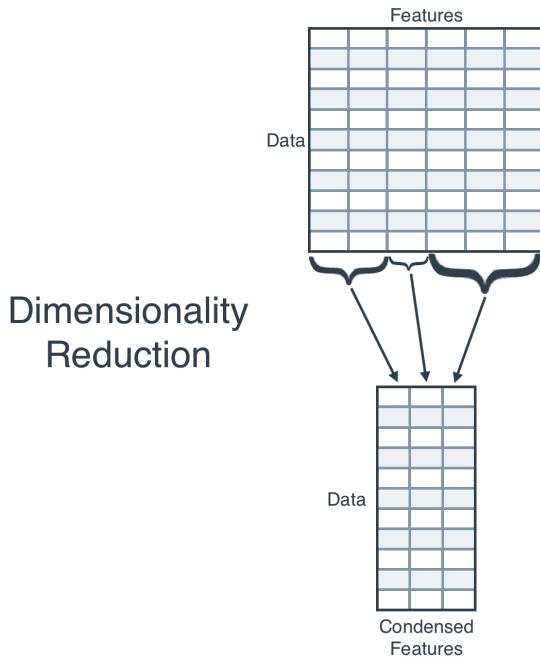


Figure 2.10. Clustering can be seen as a way to simplify our data by reducing the number of rows in our dataset, by grouping some rows into one.



**Figure 2.11.** Dimensionality reduction can be seen as a way to simplify our data by reducing the number of columns in our dataset, by grouping some columns into one.

You may be wondering, is there a way that we can reduce both the rows and the columns at the same time? And the answer is yes! One of the ways to do this is called matrix factorization. Matrix factorization is a way to condense both our rows and our columns. If you are familiar with linear algebra, what we are doing is expressing our big matrix of data into a product of two smaller matrices.

Places like Netflix use matrix factorization extensively to make recommendations. Imagine a large table where each row is a user, each column is a movie, and each entry in the matrix is the rating that the user gave the movie. With matrix factorization, one can extract certain features such as type of movie, actors appearing in the movie, and others, and be able to predict the rating that a user gives a movie, based on these features.

## 2.4 What is reinforcement learning?

Reinforcement learning is a different type of machine learning, in which no data is given, and we must solve a problem. Instead of data, an environment is given, and an agent who is supposed to navigate in this environment. The agent has a goal, or a set of goals. The environment has rewards and punishments, which guide the agent to take the right decisions in order to reach its goal. That all sounded a bit abstract, but let's look at some examples.

### Example 1: Grid world

In Figure 2.10 we see a grid world with a robot on the bottom left corner. That is our agent. The goal is to get to the treasure chest in the top right of the grid. In the grid, we can also see a mountain, which means we cannot go through that square, since the robot cannot climb mountains. We also see a dragon, which will attack the robot, should the robot dare to land in the square of the dragon, so part of our goal is to not land over there. This is the game. And in order to give the robot information about how to proceed, we have a score. The score starts at zero. If we get to the treasure chest, then we gain 100 points. If we reach the dragon, we lose 50 points. And to make things fast, let's say that for every step the robot makes, we lose 1 point, since the robot loses energy.

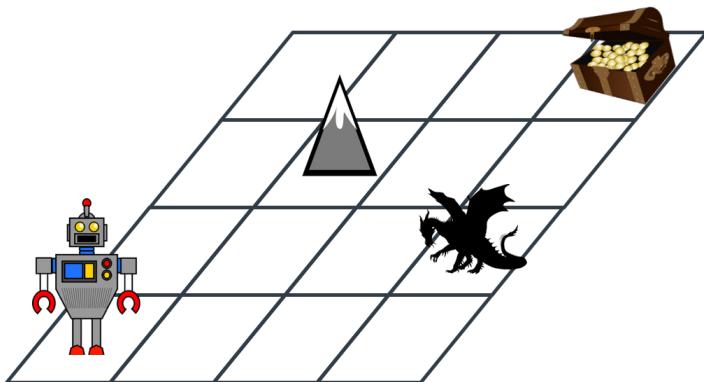
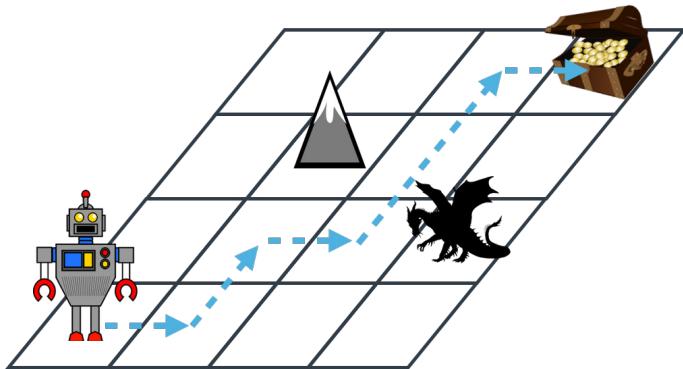


Figure 2.12. A grid world in which our agent is a robot. The goal of the robot is to find the treasure chest, while avoiding the dragon. The mountain represents a place in which the robot can't pass through.

The way to train this algorithm, in very rough terms, is as follows. The robot starts walking around, recording its score, and remembering what steps took it to each decision. After some point, it may meet the dragon, losing many points. Therefore, it learns that the dragon square, and squares close to it, are associated to low scores. At some point it may also hit the treasure chest, and it starts associating that square, and squares close to it, to high scores. Eventually, the robot will have a good idea of how good each square is, and can take the path following the squares all the way to the chest. Figure 2.11 shows a possible path, although this one is not ideal, since it passes close to the dragon. Can you think of a better one?



**Figure 2.13.** Here is a path that the robot could take to find the treasure chest.

Now, of course this was a very brief explanation, and there is a lot more to this. There are many books written only about reinforcement learning. For example, we highly recommend you Miguel Morales's book, called "Grokking Deep Reinforcement Learning". But for the most part, anytime you have an agent navigating an environment, picking up information and learning how to get rewards and avoid punishment, you have reinforcement learning.

Reinforcement learning has numerous cutting edge applications, and here are some of them.

- Games: The recent advances teaching computers how to win at games such as Go or chess, use reinforcement learning. Also, agents have been taught to win at Atari games such as Breakout or Super Mario.
- Robotics: Reinforcement learning is used extensively to help robots do tasks such as picking up boxes, cleaning a room, or any similar actions.
- Self driving cars: For anything from path planning to controlling the car, reinforcement learning techniques are used.

## 2.5 Summary

- There are several types of machine learning, including supervised learning and unsupervised learning.
- Supervised learning is used on labelled data, and it is good for making predictions.
- Unsupervised learning is used on unlabelled data, and it is normally used as a preprocessing step.
- Two very common types of supervised learning algorithms are called regression and classification.
  - Regression models are those in which the answer is any number.
  - Classification models are those in which the answer is of a type yes/no. The answer is normally given as a number between 0 and 1, denoting a probability.

- Two very common types of unsupervised learning algorithms are clustering and dimensionality reduction.
  - Clustering is used to group our data into similar clusters, in order to extract information, or make it easier to handle.
  - Dimensionality reduction is a way to simplify our data, by joining certain similar features and losing as little information as possible.
- Reinforcement learning is a type of machine learning used where an agent has to navigate an environment and reach a goal. It is extensively used in many cutting edge applications.

# 3

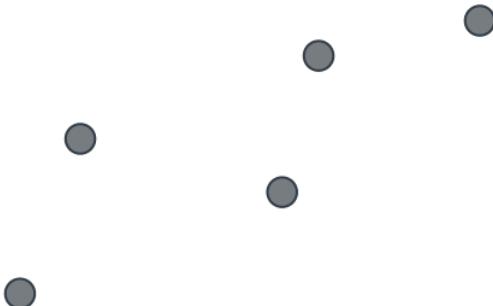
## *Drawing a line close to our points: Linear regression*

### This chapter covers

- What is linear regression?
- How to predict the price of a house based on known prices of other houses
- How to fit a line through a set of data points.
- How to code the linear regression algorithm in Python.
- Examples of linear regression in the real world, such as medical applications and recommender systems.

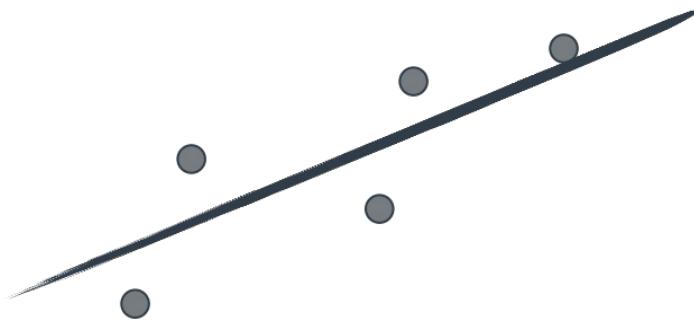
In this chapter we learn linear regression. Linear regression is a very powerful and common method to estimate values, such as the price of a house, the value of a certain stock, the life expectancy of an individual, the amount of time a user will watch a video or spend in a website, etc. In most books you will find linear regression as a plethora of complicated formulas, matrices, derivatives, determinants, etc. Here, this will not happen, I promise. All you need to have is the ability to visualize points and lines in a plane, and visualize the lines moving around to get closer to the points.

The mental picture of linear regression is simple. Let us say we have some points, that roughly look like they are forming a line, like in Figure 3.1.



**Figure 3.1.** Some points that roughly look like forming a line.

The goal of linear regression is to draw the line that passes as close as possible to these points. What line would you draw, that goes close to those points? The one I drew is in Figure 3.2.



**Figure 3.2.** A line that goes close to the points.

The way I like to think of it is as if the points are houses in a town, and we should pass a road through the town. The houses would like to be as close as possible to the road, so we need to find the road that passes the closest to the houses.

Another way I like to imagine it is if all the points are magnets laying bolted to the floor (so they can't move), and we throw a straight metal rod on top of them. The rod will move around, but eventually it would end up in a position of equilibrium as close as possible 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 far from others? This and many other questions should arise in your head. Here are some that I can think of:

- 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 (or the computer) find such a line?
- Why is this useful in the real world?

- Why is this machine learning?

In this chapter, we answer all of these questions. But first, let's start with a very simple example.

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

Let's say we are a real estate agent, and we are 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 which could influence the house, such as size, number of rooms, location, crime rate, school quality, distance to commerce, etc. At the end of the day, what we want is a formula on all these features which gives us the price of the house, or at least an estimate for it.

### 3.2 The solution: Building a regression model for housing prices

Let's go with as simple an example as possible. We'll only look at one of the features, which is the number of rooms. Our house has 4 rooms, and there are 6 houses nearby, with 1, 2, 3, 5, 6, and 7 rooms, respectively. Their prices are in Table 3.3.

**Table 3.1.** A table of houses with their number of rooms, and their price. 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

Take a look at this table. What price would you give to house 4, just from this table? If you said \$300, that is also my guess. You probably saw a pattern, and used it to infer the price of the house. Congratulations! What you did in your head was linear regression. Let's study this pattern more. You probably 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 this simple formula:

```
Price = 100 + 50 * (Number of rooms)
```

What we did here, is come up with a model, which is a formula that gives us a *prediction* of the price of the house, based on the *feature*, which is the number of rooms. These are some very important concepts in machine learning.

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

**LABELS** This is the target, or what we try to predict from the features. In this case, the label is the price of the house.

**MODEL** A machine learning model is simply 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 simply the output of the model. If the classifier says "I think the house with 4 rooms is going to cost \$300", then the prediction is 300. It may or may not be close to the actual price of the house, but from what we know, it is very likely that it will be close.

Ok, 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 formula? In order to illustrate this, let's look at a slightly more complicated example. And since this is a machine learning problem, we will approach it using the remember-formulate-predict framework that we learned in Chapter 2.

### 3.2.1 The remember step: looking at the prices of existing houses

Let's look at a slightly more complicated dataset, like the one in Table 3.3.

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 one is very 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 following one. Can you see the pattern, and more than that, can you find some good predictions for the price of the house with four rooms?

For now, let's start by doing what I do when I am trying to figure things out: I start plotting things. If we plot 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, the graph looks like Figure 3.3.

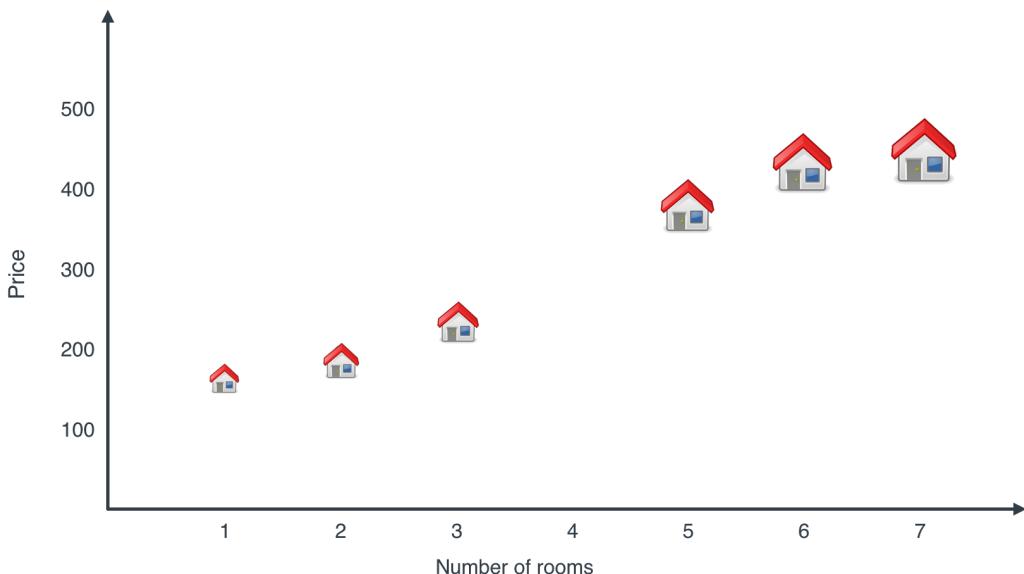


Figure 3.3. The plot of our slightly more complicated dataset of houses with their number of rooms and their price.

### 3.2.2 The formulate step: formulating a rule that estimates the price of the house

The dataset in Table 3.3 is close enough to the one in Table 3.2, so we should feel safe to use 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 the formula as follows:

Price = 100 + 50\*(Number of rooms) + (Small error)

But the point is that if we were to predict prices, we can use this formula, and even though we are not sure we'll get the actual value, we know it's very likely that we are close to it.

Now the question is, how did we find this equation? And most importantly, 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. If the words slope and y-intercept ring a bell, this line is precisely the line with slope 50, and y-intercept 100. The y-intercept being 100 means that our estimate for the price of a (hypothetical) house with zero rooms, would be the base price of \$100. The slope being 50 means that each time we add one room to the house, we estimate that the price of the house will go up by \$50. This line is drawn in Figure 3.4.

**SLOPE** The slope of a line is a measure of how steep the line 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 is constant over the whole line. In a machine learning model, 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.

**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 tells us what the label would be in a datapoint where the feature is 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  $(x,y)$  where the  $y$  satisfies the equation of the line. In a machine learning model, we insert the feature into the variable  $x$ , and the prediction for the label is what comes out of the  $y$ . The  $m$  and the  $b$  are the parameters of the model.

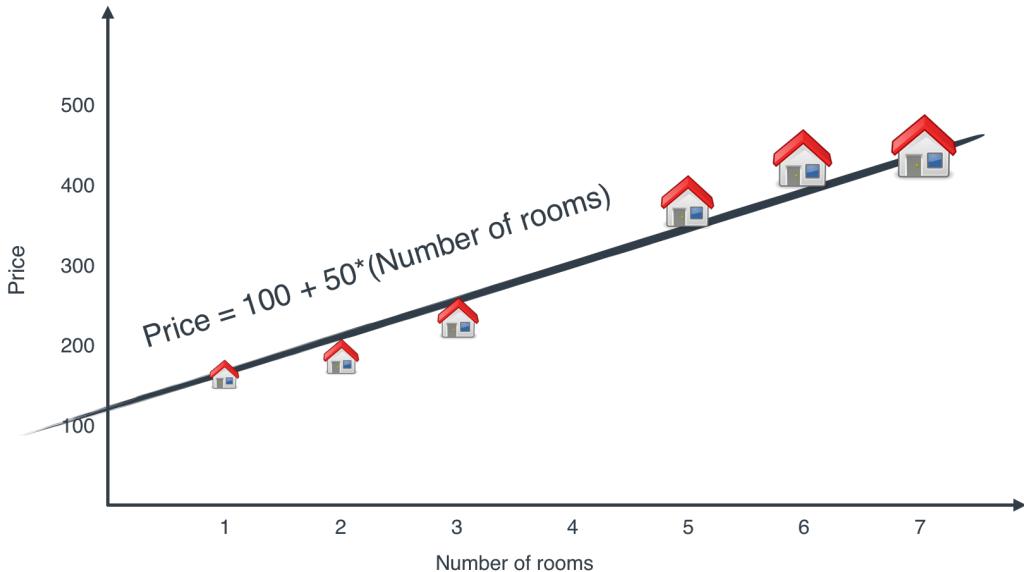


Figure 3.4. The model we formulate is the line that goes as close as possible to all the houses.

Now, of all the possible lines (each one with its own equation), why did we pick this one in particular? The reason is, because that one passes by the points pretty closely. There may be a better one, but at least we know this one is very good, as opposed to one that goes nowhere near the points. Now we are back into the original problem, where we have a set of houses, and we want to pass a road as closely as possible for 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 magic ball that given a bunch of points, finds the line that passes the closest to them.

### 3.2.3 The predict step: what do we do when a new house comes in the market?

Now, on to using our model to predict the price of the house with 4 rooms. This is very simple, all we do is plug the number 4 into our formula, to get the following:

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

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

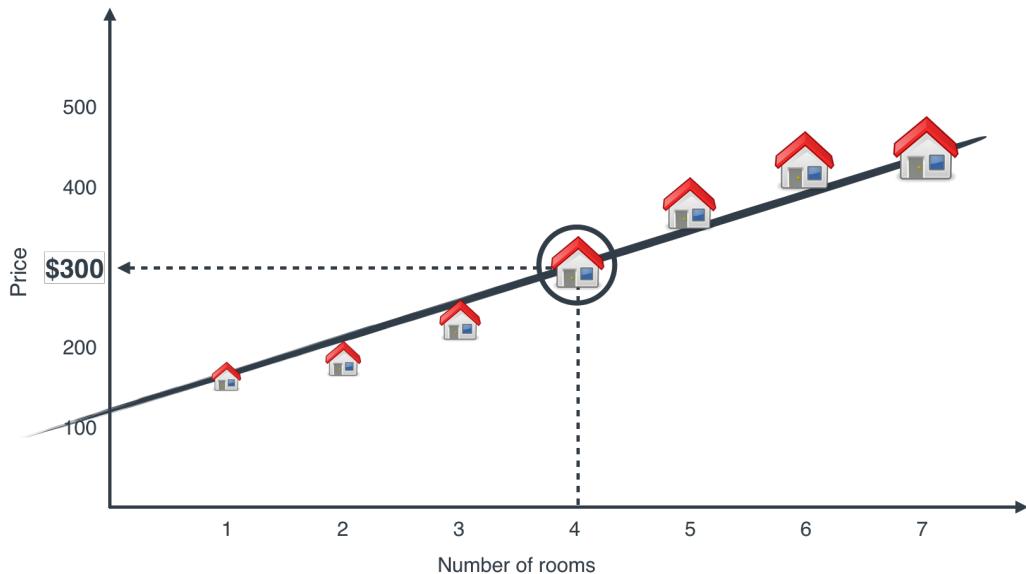


Figure 3.5. Our task is now to predict the price of the house with 4 rooms. Using the model (line), we deduce that the predicted price of this house is \$300.

### 3.2.4 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 on earth did you come up with the scoring system? And what would we do if instead of 6 houses, we have thousands of them?
3. Why did you only use the number of rooms to predict the price of the house? I can think of other features like size, number of floors, number of parking spots, crime rate in the area, school quality, distance to a highway, etc.
4. What if we've built this prediction mode, and then new houses start appearing in the market, is there a way to update the model with new information?

This chapter answers all of these questions, but let me give you some quick answers:

1. **What happens if the model makes a mistake?**

Since the model is estimating the price of a house, we expect it to make a small mistake pretty much all the time, since it is very hard to actually hit the exact price. However, we want to build our model so that we expect it to make small errors for most of the houses, so that we can use it effectively.

2. **How on earth did you come up with the scoring system? And what would we do if instead of 6 houses, we have thousands of them?**

Yes, this is the main question we address on this chapter! When we have 6 houses, the problem of drawing a line that goes close to them is simple, but if we have thousands of houses, this gets hard. What we do in this chapter is to device an algorithm, or a procedure, for the computer to find a good line.

**3. Why did you only use the number of rooms to predict the price of the house? I can think of other features like size, number of floors, number of parking spots, crime rate in the area, school quality, distance to a highway, etc.**

Absolutely. The price of a house doesn't just depend on the number of houses, the location matters, the size matters, etc. All of these, and many more, are valid features. In this chapter we use only one feature so that we can plot the prices and draw a line through them. But in real life, we would be predicting the price using lots of features, and the formulas we get will be very similar, except with a lot more variables.

**4. What if we've built this prediction mode, 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 comes into the model. 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.

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

Now we get to the main question on this chapter. How do we get a computer to draw this line. Before we get into the math, let's think. How would we find the perfect model? Actually, there is a simple question we can ask. The question is, imagine if we already *had* a model. How do we make our existing model *a little bit* better? Even if it is just a little bit. If we have, say, a model that says that the price is  $\$50 + \$40 * (\text{Number of rooms})$ , and a house with 2 rooms which costs \$150, then what's wrong with our model? Well, the model predicts that the house will cost  $\$50 + \$40 * 2 = \$130$ . But the house is \$150. That's not bad, but we can do better. We can imagine that if the model thought the house would be cheaper than it is, then a new model that assigns a slightly higher weight to the base price and a slightly higher weight to the price per room, will do a bit better for this house, correct? So let's tweak the model a little bit. Let's make a model that says that the price is  $\$51 + \$41 * (\text{Number of rooms})$ . This model assigns the house a price of  $\$51 + \$41 * 2 = \$133$ , which is closer to \$150. It's not perfect, but it is *a little* better. What if instead we had a house with two rooms that costs \$80? Then we'll decrease the base price and the price per room a little bit.

You can imagine that this process is not exact, and we may increase and decrease the weights unnecessarily. But Imagine doing this many many times, picking a random house, and using it to make the model a little better. Since computers can run a loop many times at great speed, it

is imaginable that this technique works. And it turns out to work very well. Therefore, here is our algorithm to build a price model for our data set of houses:

1. Start with random values for the base price and the price per room.
2. Repeat many times:
  - a) Pick a random house, and use it to increase or decrease the base price and price per room by a small amount, so the model will predict its price better.
3. Enjoy your model!

Of course, this requires a lot more details, and some math needs to be worked out. For now, let's start looking at what this means geometrically. We will start by drawing a random line, and that is our first model. This line is probably not a great fit. That's ok, we'll make it better gradually in the simplest possible way, repeating the following two simple steps many times.

1. Picking a random point.
2. Moving the line towards this point, by a very small amount.

You may think, what if I have a point that is really far away, and it messes up my procedure? That is ok, if we only move the line by small amounts, this point can't harm us very much.

Now the question is, how do we move a line closer to a point? In the next few sections, we will learn some very simple tricks to do this. But first, a very quick recap on slope and y-intercept.

### **3.3.1 Crash course on slope and y-intercept**

When we think of a line, we can think of it graphically, or as an equation. If we think of it as an equation, then it has two components:

- The slope.
- The y-intercept.

The slope simply tells us how steep the line is, and the y-intercept tells us where the line is located. The slope is defined as raise divided by run, and the y-intercept tells us where the line crosses the y-axis (the vertical one). In the Figure 3.6 we can see both of them in an example. This line has the following equation:

$$y = 0.5x + 2.$$

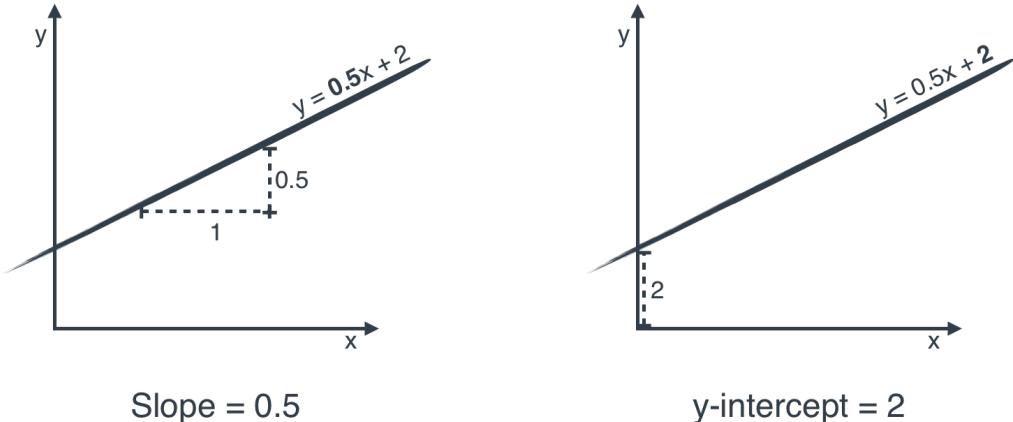


Figure 3.6. The line with equation  $y = 0.5x + 2$  has slope 0.5 (left) and y-intercept 2 (right).

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

The slope being 0.5 means that when we walk along this line, for every unit that we move towards 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. However, many lines can have the same slope. I can draw a parallel line to this one, and it would also rise 0.5 units for every unit it moves towards the right. This is where the y-intercept comes in. The y-intercept tells us where the line cuts the y-axis. This particular line cuts the line at height 2, so that is the y-intercept.

In other words, the slope of the line tells us the *direction* that the line is pointing towards, and the y-intercept tells us the *location* of the line. Notice that by specifying the slope and the y-intercept, the line gets completely specified. Also, think of what would happen if we change the slope and the y-intercept by small amounts. The line would move a little, right? In which directions? We'll see this next.

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

Now we get to our problem, which is: we have a point, and a line, and we need to move the line closer to the point.



Figure 3.7. Our goal: To move the line closer to the point.

Well, the way to do this is to change the slope and the y-intercept by a small amount. We have to be careful here, since we have to figure out if we want to add or subtract a small amount. Notice that from the way we defined the slope, the higher the slope, the higher the amount the line rises. Therefore, steeper lines have larger slopes. The y-intercept tells us where the line is located, so a line with high y-intercept will meet the y-axis at a higher position than one with a low y-intercept. In Figure 3.8 we can see this in some examples of slopes and y-intercepts.

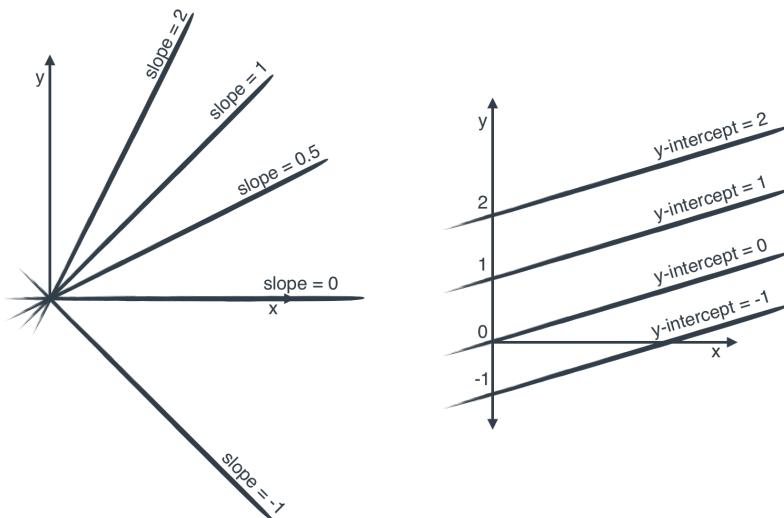


Figure 3.8. Some examples of slope and y-intercept. In the left, we can see several lines with the same intercept and different slopes, and notice that the higher the slope, the steeper the line is. In the right, we can see several lines with the same slope and different y-intercepts, and notice that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

From these, we can conclude the following, which is summarized in Figure 3.9:

### 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 will translate upwards.
- If we decrease the y-intercept of a line, the line will translate downwards.

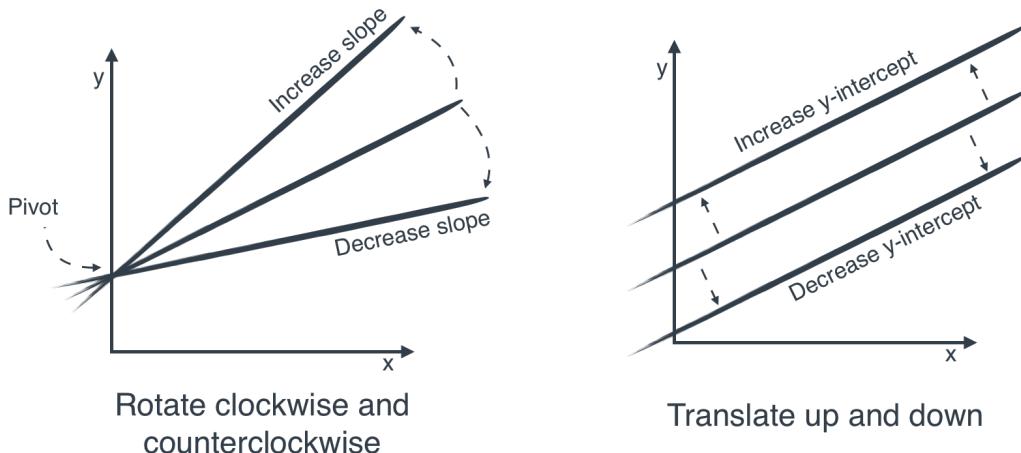


Figure 3.9. Left: Increasing the slope rotates the line counterclockwise, whereas decreasing it rotates it clockwise. Right: Increasing the y-intercept translates the line upwards, whereas decreasing it translates it downwards.

So now we are ready to start moving our lines around!

The trick to move the line correctly towards 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 since the pivot is the point of intersection of the line and the y-axis, then 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 3.10.

**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 upwards.

**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 upwards.

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

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

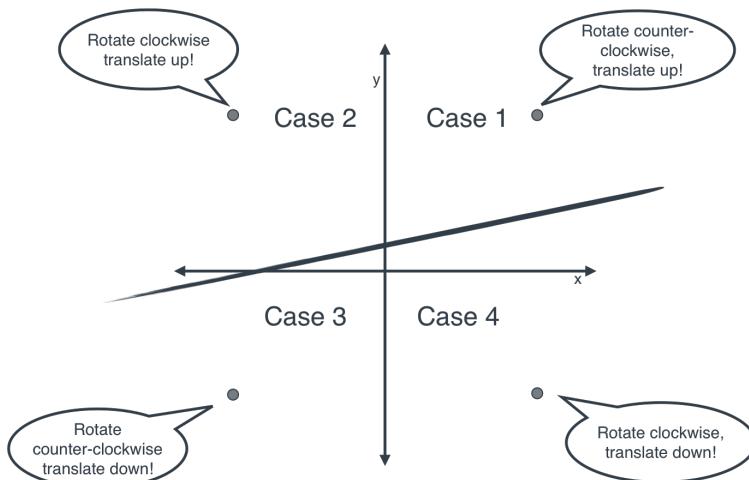


Figure 3.10. The four scenarios. In each of these we have to rotate the line and translate it in a different way in order to move the line closer to the point.

Ok, we are finally ready for our algorithm! All we need to do is a quick check of where the point is, and then we move the line. We simply do the following, we look at where the point is, and add or subtract small amounts to the slope and the y-intercept depending on if we need to increase them or decrease them. The following four cases summarize our procedure:

**Case 1:** If the point is above the line and to the right of the y-axis:

- Add a small amount to the slope.
- Add a small amount to the y-intercept.

**Case 2:** If the point is above the line and to the left of the y-axis

- Subtract a small amount to the slope.
- Add a small amount to the y-intercept.

**Case 3:** If the point is below the line and to the right of the y-axis

- Add a small amount to the slope.
- Subtract a small amount to the y-intercept.

**Case 4:** If the point is below the line and to the left of the y-axis

- Subtract a small amount to the slope.
- Subtract a small amount to the y-intercept.

What each of these cases do, is they make sure that the line rotates and translates a small amount, in order to get closer to the point.

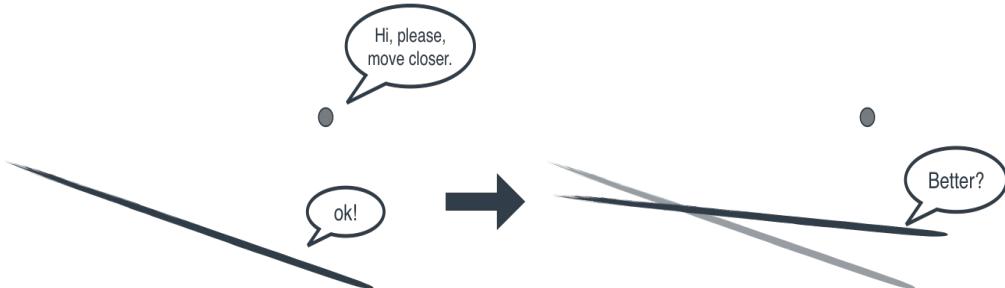


Figure 3.11. The line rotates and translates a small amount in order to get closer to the point.

The question is what is this small amount? Well, in machine learning we always want to make very small steps. So let's pick a very small number for this step size, say, 0.01. This number is very important in machine learning, it is called the *learning rate*.

**LEARNING RATE** A very small number that we pick before training our model, and that will help us make sure our models change in very small amounts while training. The learning rate works in different ways for different models. In the linear regression case, it is a number that we add or subtract to the slope and the y-intercept in order to move the line towards a point very slowly.

We are almost ready to code the algorithm. But before, let's remember that the slope is the price we add per room, and the y-intercept is the base price of the house. The linear equation for the price is the base price plus the price per room times the number of rooms, which makes sense. Since the point on the line is the price that the model predicted, and the point we are approximating is the price of the house, then our four cases from above can be described as follows:

#### PSEUDOCODE FOR THE SIMPLE TRICK

**Case 1:** If the price of the house is higher than the price the model predicted, and the number of rooms is positive:

- Add 1 cent to the price per room
- Add 1 cent to the base price.

**Case 2:** If the price of the house is higher than the price the model predicted, and the number of rooms is negative:

- Subtract 1 cent to the price per room
- Add 1 cent to the base price.

**Case 3:** If the price of the house is lower than the price the model predicted, and the number of rooms is positive:

- Add 1 cent to the price per room
- Subtract 1 cent to the base price.

**Case 4:** If the price of the house is lower than the price the model predicted, and the number of rooms is negative:

- Subtract 1 cent to the price per room
- Subtract 1 cent to the base price.

Since the number of rooms is always positive, then only cases 1 and 3 make sense, but in cases where variables can have negative values, we need all four cases.

Notice too, that cases 1 and 3 make sense, for the following reason. If we predicted a price lower than the price of the house, we would be inclined to think that our base price and our price per room are too small. Therefore, it makes sense to add 1 cent to each one. This would make our model just a little better, but remember, we can loop over this many many times, until we get a good model.

We are ready to code this algorithm in Python! All this code appears in our public repository at <https://www.github.com/luisguiserrano/manning>.

```
def simple_trick(base_price, price_per_room, num_rooms, price, learning_rate): #A
    predicted_price = base_price + price_per_room*num_rooms #B
    if price > predicted_price and x > 0: #C
        price_per_room += learning_rate #D
        base_price += learning_rate #E
    if price > predicted_price and x < 0:
        price_per_room -= learning_rate
        base_price += learning_rate
    if price < predicted_price and x > 0:
        price_per_room += learning_rate
        base_price -= learning_rate
    if price < predicted_price and x < 0:
        price_per_room -= learning_rate
        base_price -= learning_rate
    return price_per_room, base_price
```

#A Recall that base\_price is the y-intercept, and price\_per\_room is the slope.

#B Calculating the prediction.

#C Checking where the point is with respect to the line.

#D Translating the line.

#E Rotating the line.

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

The simple trick works ok, but we can do better. What if we try to bring the four **if** statements in the simple trick down to 1? Let's simplify the problem, what if we added a slightly different amount, but that still manages to be positive or negative, depending if we want to add it or subtract it in the simple trick? Well, here is an idea, first let's keep in mind the following observation.

**OBSERVATION 1** In the simple trick, when the point is above the line, we add a small amount to the y-intercept, and when the point is below the line, we subtract a small amount to the y-intercept.

Wouldn't it be nice if we had some quantity that took care of this adding or subtracting? Luckily we do. Let's take a look at this particular quantity: The actual price of the house minus the price that the model predicted. This is the same as the label minus the predicted label. Notice the following.

**OBSERVATION 2** If a point is to the above the line, the difference of price minus predicted price is positive. If it is below the line, then this difference is negative.

Figures 13 illustrates this observation. Notice that if a point is above the line, then the price is higher than the predicted price (the point over the line), so the price minus the predicted price is positive. If the point is under the line, then the predicted price is smaller than the actual price, so this difference is negative.

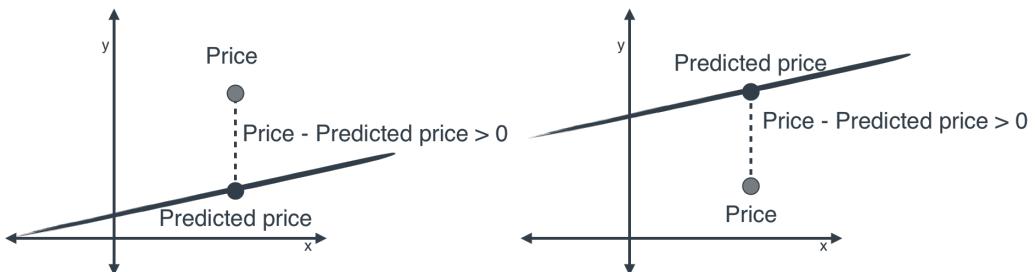


Figure 3.12.

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.

So why don't we change the simple trick, and instead of adding or subtracting a small amount (the learning rate), we now simply add the learning rate times this difference? This will

completely take care of it, since the difference is positive when the point is above the line, and negative when it is below the line! We are adding a different amount, but that is ok, we are still taking a step in the correct direction, and the learning rate makes sure that the step we take is very small.

Now, wouldn't it be nice if we can do the same thing for the slope? Luckily, we can again! First, let's remember what the rule is for updating the slope:

**OBSERVATION 3** In the simple trick, when the point is either above the line and to the right of the y-axis, or below the line and to the left of the y-axis, we add a small amount to the slope. Otherwise, we subtract a small amount to the slope.

This case is a bit more complicated, because we need to look at two quantities. The first one is the same one as before. The second one is the number of rooms in the house. Now of course, this number is always positive, since it is counting the rooms in a house. But if we are considering different datasets of different things, we could envision a quantity that is negative. How would this look in the plots? It would simply be a point that's to the left of the y-axis instead of to the right. That will be our second quantity.

**OBSERVATION 4** If the point is to the right of the y-axis, then the number of rooms is positive. If the point is to the left of the y-axis, then this quantity is negative.

Figure 3.13 illustrates the second quantity. When the x-coordinate, in this case the number of rooms, is positive, then the point is to the right of the y-axis. If it is negative, then the point is to the left of the y-axis.

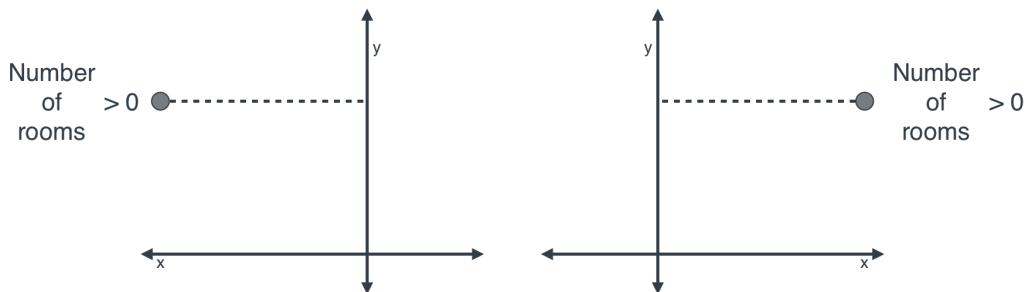


Figure 3.13. Left: When the point is to the left of the y-axis, the number of rooms is negative. Right: When the point is to the right of the y-axis, the number of rooms is positive.

Now, this quantity is not enough to help us out, but if we take the *product* of it times the previous quantity, we get what we want. In other words, we will consider the following product:

$$(\text{Price} - \text{Predicted price}) * (\text{Number of rooms})$$

That product has the correct sign, so that if we add it to the slope, we will always move it in the correct direction. Let's double check this.

**Case 1:** If the price of the house is higher than the price the model predicted, and the number of rooms is positive:

- (Price - Predicted price) \* (Number of rooms) is positive.

**Case 2:** If the price of the house is higher than the price the model predicted, and the number of rooms is negative:

- (Price - Predicted price) \* (Number of rooms) is negative.

**Case 3:** If the price of the house is lower than the price the model predicted, and the number of rooms is positive:

- (Price - Predicted price) \* (Number of rooms) is negative.

**Case 4:** If the price of the house is lower than the price the model predicted, and the number of rooms is negative:

- (Price - Predicted price) \* (Number of rooms) is positive.

Therefore, the pseudocode for our square trick is the following:

#### PSEUDOCODE FOR THE SQUARE TRICK

**All cases:**

- Add the learning rate \* (Price - Predicted price) \* (Number of rooms) to the price per room.
- Add the learning rate \* (Price - Predicted price) to the base price.

And here is the code:

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

#A Calculating the prediction.

#B Translating the line.

#C Rotating the line.

#### 3.3.4 The linear regression algorithm: Repeating the square trick many times

Now we are ready for our linear regression algorithm! This algorithm takes as input a bunch of points, and it returns a line that fits them well. We've already done all the hard work, all we need is to start with random values for our slope and our y-intercept, and then repeat the procedure of updating them many many times. Here is the pseudocode:

### PSEUDOCODE FOR FITTING A LINE THROUGH A SET OF POINTS (LINEAR REGRESSION)

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

The simple trick was used mostly for illustration, in real life, we use the square trick, which works a lot better. Therefore, we'll use this one. Here is the actual code. Note that we have used the random package to generate random numbers for our initial values (slope and y-intercept), and for selecting our points inside the loop.

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

#A Importing the random package to generate (pseudo) random numbers.

#B Generating random values for the slope and the y-intercept.

#C Repeating the update step many times.

#D Picking a random point on our dataset.

#E Applying the square trick to move the line closer to our point.

Let's quickly write some plotting functions, and then we are ready to test this algorithm on our dataset!

#### 3.3.5 Plotting dots and lines

Throughout this chapter, we make plots of our data and models using Matplotlib and Numpy, two very useful Python packages. In this book we show you the plots, but not the code for the plots, as it is out of scope. However, you are more than invited to look at it in detail at our public Github repository at <https://www.github.com/luisquiserrano/manning>.

The first plot we'll show is Figure 3.14, which is the plot of the points in our small housing dataset from Table 3.3. Notice that the points do appear close to forming a line.

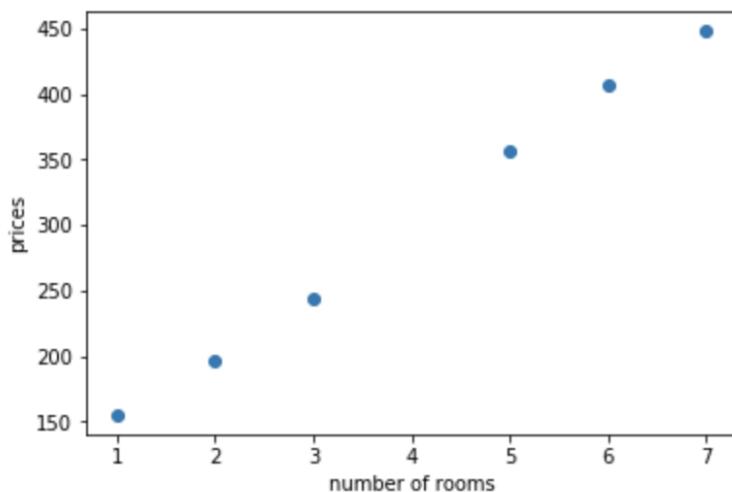


Figure 3.14. The plot of the points on Table 3.3.

### 3.3.6 Using the linear regression algorithm in our dataset

Now, let's apply the algorithm to our dataset! The following line of code runs the algorithm (using the square trick) with the features, the labels, the learning rate equal to 0.01, and the number of epochs equal to 10000. The result is the plot in Figure 3.15.

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

Price per room: \$51.07296115119787  
 Base price: \$99.47510567502614

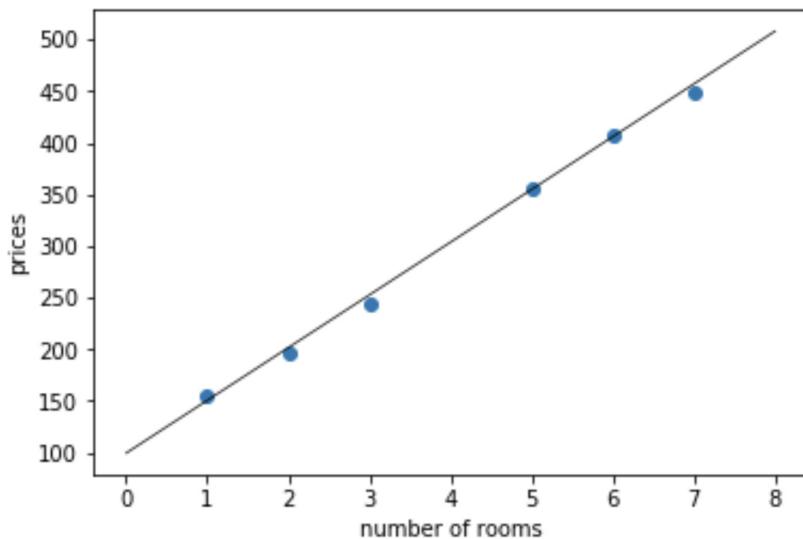


Figure 3.15. The plot of the points on Table 3.3, and the line that we obtained with the linear regression algorithm.

Figure 3.14 shows the line where the price per room is \$51.07, and the base price is \$99.48. This is not far from the \$50 and \$100 we eyeballed earlier in the chapter.

But let's look at the progression a bit more, let's draw a few of the intermediate lines (again, for code on how we plot them, please see the Github repo). The result is in Figure 3.16, where we can see that the line starts far from the points, and moves slowly to fit better and better every time. Notice that at first (in the first 10 epochs), the line moves quickly towards a good solution. After epoch 50, the line is good, but it still doesn't fit the points very well. Then it takes 950 more epochs to go from a good fit to a great fit.

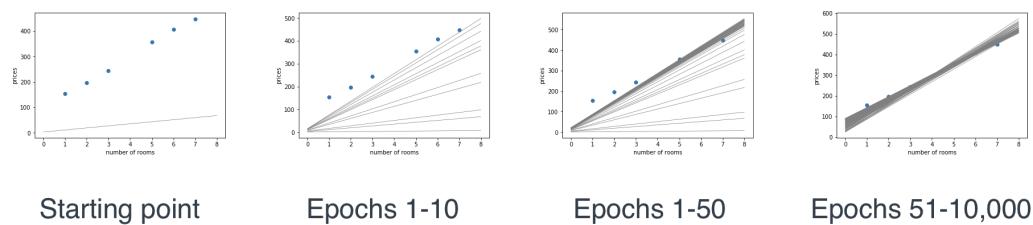


Figure 3.16. Drawing some of the lines in our algorithm, as we approach a better solution. The first graphic

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 10000 (the last epoch).

### 3.4 How do we measure our results? The error function

We have seen a direct approach to finding the best line fit, but many times this is a difficult thing to do. A more indirect, yet more mechanical way to do this is using *error functions*. An error function is simply a metric of how our model is doing. For example, take a look at the two models in Figure 3.17. The one in the left is very bad, while the one in the right is good. The error function will simply assign a large value to the one in the left (the bad model) and a small value to the one in the right (the good model). Error functions are many times also called *loss functions*.

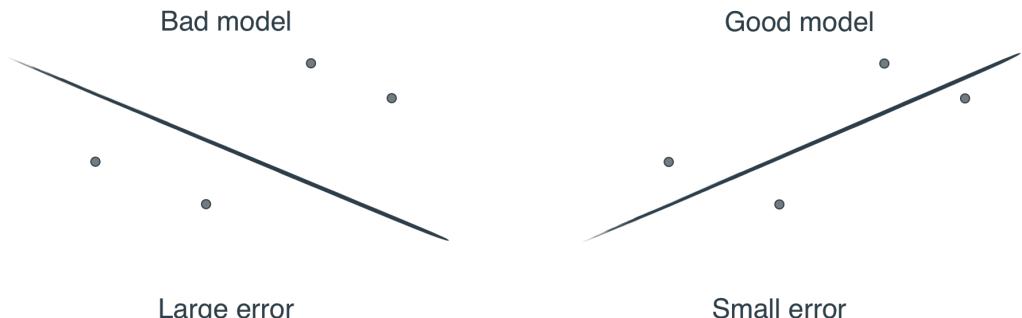


Figure 3.17. Two models, a bad one (at the left) and a good one (at the right). The bad one is assigned a large error, and the good one is assigned a small error.

Now the question is, how do we define a good error function for linear regression models? There are two main ways to do this, namely the *absolute error* and the *square error*.

#### 3.4.1 The absolute error

By definition, a good linear regression model is one where the line passes by close to the points. What does close mean in this case? This is a subjective question, since a line that is close to some of the points may be far from others. In that case, do we rather find a line that is very close to some of the points and far from some of the others? Or do we try to find one that is somewhat close to all the points? The absolute error will help us make this decision. What we do is, for a line, we calculate the sum of vertical distances from each of the points to the line. That measure is precisely our absolute error, and it is illustrated in Figure 3.18. The error is calculated as the sum of the lengths of the red segments.

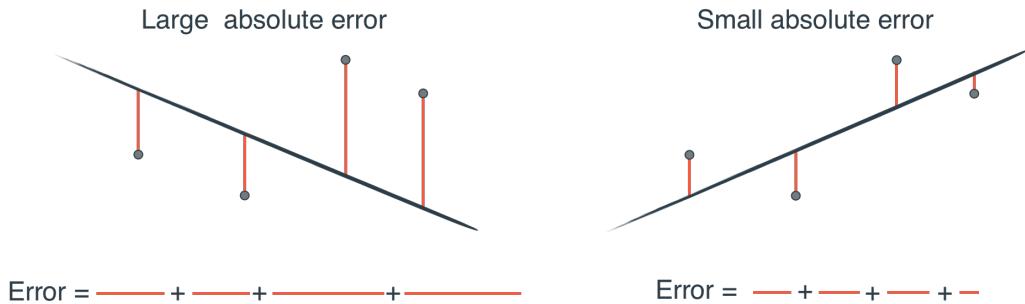


Figure 3.18. The absolute error is the sum of the vertical distances from the points to the line. Note that the bad model in the leftt has a large absolute error, while the good model in the right has a small absolute error.

Why is it called the *absolute error*? The reason is because in order to calculate each of the distances, one takes the difference between the label and the predicted label. This is the height of the point minus the height of the line at that point. If the point is above the line, this difference is positive, but if the point is below the line, this difference is negative. Thus, we take the absolute value of the difference, and this explains the name.

### 3.4.2 The square error

The square error is very similar to the absolute error, except that instead of adding the lengths of the segments, we add the squares of these lengths. The process is illustrated in Figure 3.19, and you can see how the bad model in the left has a large square error, while the good model in the right has a small square error.

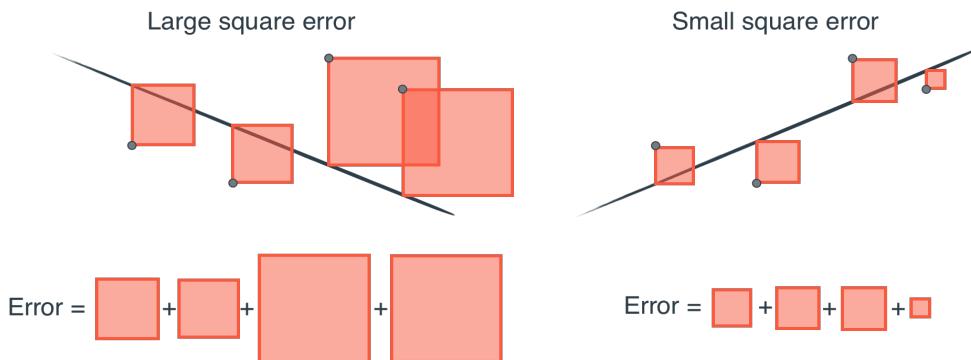


Figure 3.19. The absolute error is the sum of the vertical distances from the points to the line. Note that the bad model in the left has a large absolute error, while the good model in the right has a small absolute error.

The square error is used more commonly in the practice than the absolute error. Why? One reason is that a square has a much nicer derivative than an absolute value, and since derivatives are used in the training process, then we prefer to use the square function.

### 3.4.3 Gradient descent

Derivatives? Yes, derivatives. But here's the great news, we already used them in the training process above. Every time I say "move a small amount in this direction", what we are doing in the background is calculating a derivative of the error function, and using it to give us a direction in which to move our line.

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 best line that 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 are finding the best line fit. Math is all about reducing numbers, only we call it *minimizing functions*. That is, finding the smallest possible value that a function can take. And that is where gradient descent comes in: it is a great way to minimize functions.

In this case, the function we are trying to minimize is the error (absolute or square). A small caveat is that gradient descent doesn't find the very minimum value of the function, but it finds something close to it. The good news is that gradient descent is fast and effective at finding minimums.

Now, how does gradient descent work? Gradient descent is the equivalent of descending from a mountain. Let's say you find yourself on top of a tall mountain called Mount Errorest. You wish to descend, but it is very foggy and you can only see about 1 meter away from you. What do you do? A good method is to look around you, and figure out at what direction you can take one single step, in a way that you descend the most (Figure 3.20).

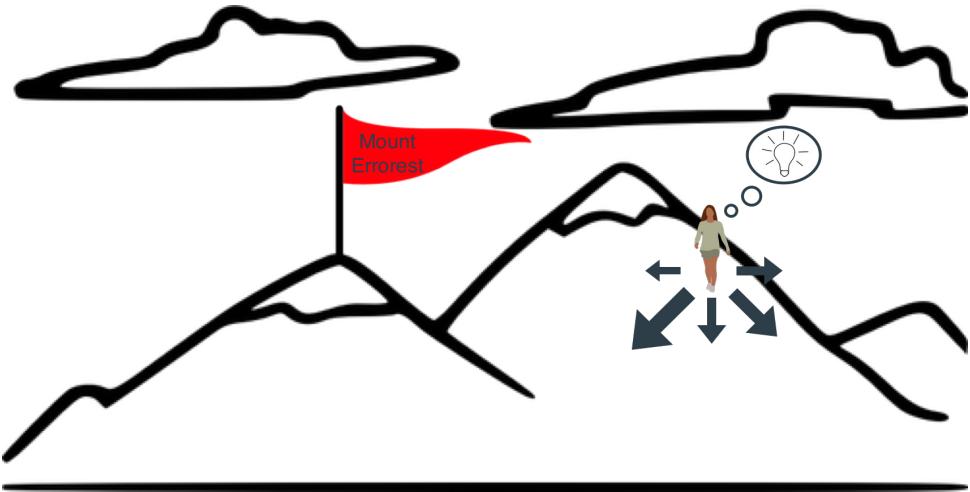


Figure 3.20. You are on top of Mount Errorest and you wish to get to the bottom, but you can't see very far. A way to go down is to look at all the directions in which you can take one step, and figure out which one helps you descend the most. Then you are one step closer to the bottom.

When you find this direction, you take one step, and then you're the closest from the bottom that you can be, with one step. You are now in the same position than before, except a little better. Thus, all you have to do is repeat this process many many times until you reach the bottom! (Hopefully!). This process is illustrated in Figure 3.21.

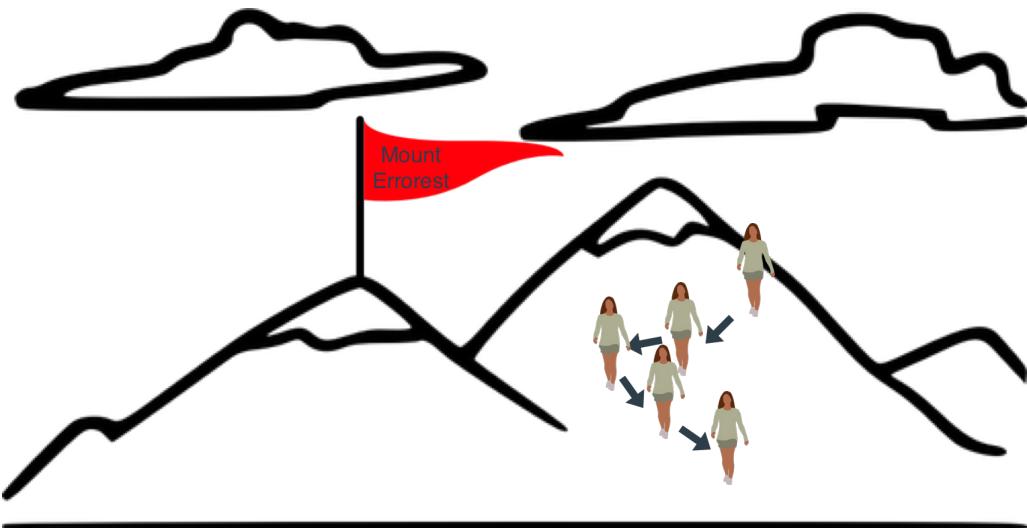


Figure 3.21. The way to descend from the mountain is to take that one step that makes you descend the most,

and continue doing this for a long time.

Why did I say hopefully? Well, there are many caveats to this process. You could reach the bottom, or you could also reach a valley and then you have nowhere to go. We won't deal with that now, but there are techniques to reduce the probability of this happening.

There is a lot of math here that I am sweeping under the rug, which I explain in more detail in the appendix. But what we did in this chapter was exactly gradient descent. How so? This is how gradient descent works:

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

And what did we do to find the best line fit? The following:

1. Start with any line.
2. Find the best direction to move our line a little bit.
3. Move the line in this direction.
4. Repeat steps 2-3 many times.

The way I picture it in my head is like in Figure 3.22. Each point in Mount Errorest corresponds to some model (line) that tries to fit our data. The height of the point is the error given by that line. 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 a 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!).

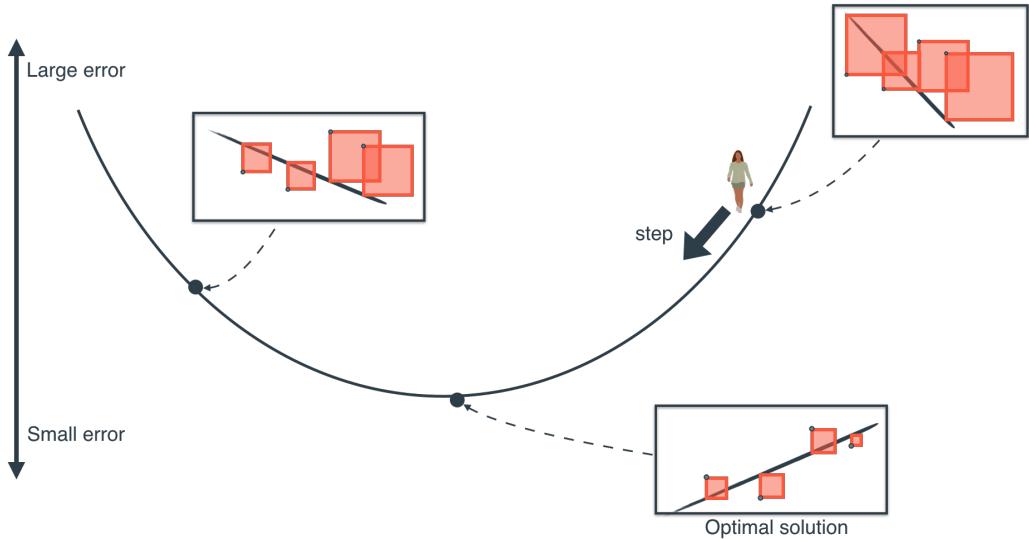


Figure 3.22. The way to descend from the mountain is to take that one step that makes you descend the most, and continue doing this for a long time.

### 3.4.4 Plotting the error function

In the repo, we have also written and plotted the square error function. This is the function, which we call *rmse*, for *root mean square error*. Notice that we didn't just take the sum of squares of the distances, we actually took the average of these, which is simply the sum of squares of distances divided by the total number of data points. Then we took the square root of this number. This is something that is used in convention, so that the units of measurement of our error are the same as the ones used in our dataset. Here is the code for the root mean square error.

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

And the plot of our error is in Figure 3.23. Note that it quickly dropped after about 1000 iterations, and it stayed small for the rest of the process. This tells us that we can run the same algorithm for only 1000 iterations instead of 10,000, and still get similar results!

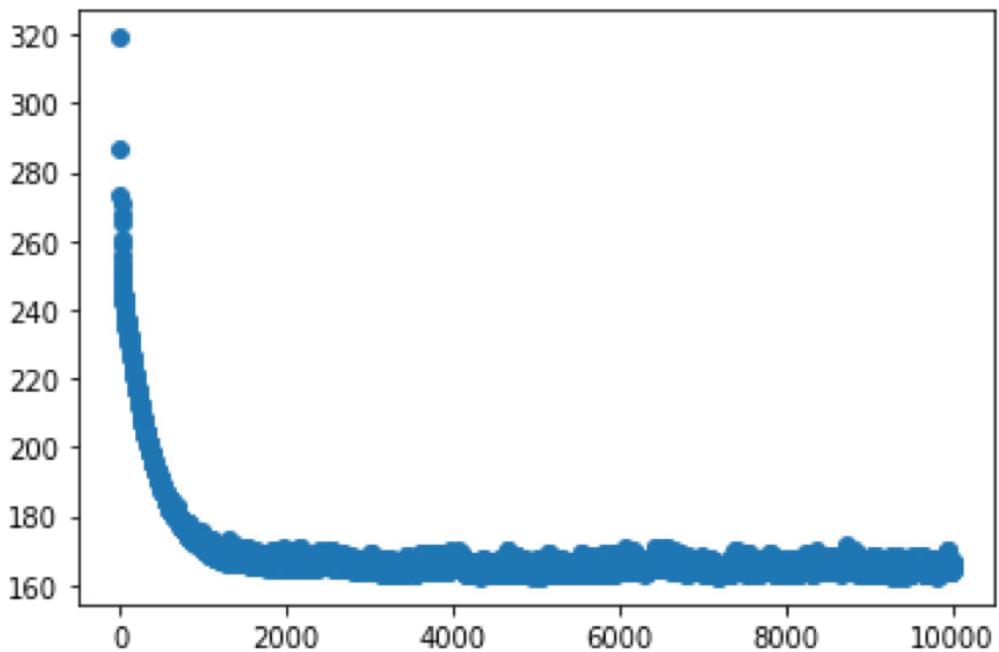


Figure 3.23. The plot of the root mean squared error for our running example. Notice how the algorithm succeeded in reducing this error after a little over 1000 iterations.

## 3.5 Applications

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 the linear regression in real life. In each of the examples, I will outline the problem, show you some features to solve it, and then let the perceptron algorithm do its magic!

### 3.5.1 Applications of linear regression

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. Since regression predicts a quantity, then all one has to do to generate good recommendations is figure out what quantity is the best at indicating user interaction or user satisfaction. Here are some examples.

#### 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, one 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, gender, etc., but they can also be behavioral, such as other videos or songs they have clicked on or interacted with.

### **PRODUCT RECOMMENDATIONS**

Stores and e-commerce websites also use linear regression to predict their sales. One way to do this is to predict how much will a customer spend in the store. This can be done using linear regression, using the amount spent as a label, and the features can be demographic and behavioral, in a similar way as with video and music recommendations.

### **HEALTHCARE**

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

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

## **3.6 Summary**

- Regression is a very important part of machine learning. It consists of training an algorithm with labelled data, and using it to make predictions on future (unlabelled) data.
- Labelled 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, distance to the highway, etc.
- 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, in order to attempt to classify them correctly.
- Linear regression has numerous applications, including recommendation systems, e-commerce, and healthcare.

# 4

## *Using lines to split our points: The perceptron algorithm*

### This chapter covers

- What is classification?
- Sentiment analysis: How to tell if a sentence is happy or sad using machine learning.
- What are perceptrons, or discrete logistic classifiers.
- What is the perceptron algorithm?
- How to draw a line that separates points of two different colors.
- How to measure the error of a line that separates points of two different colors.
- How to code the perceptron algorithm in Python.

Here is the problem we'll tackle in this chapter. How do we teach a computer to determine if a sentence is happy or sad? The way we tackle this is by introducing a branch of machine learning called *classification*. Much like regression, classification consists of predicting labels according to the features of our data. However, in regression we predicted numbers, such as the price of a house, and in classification, we predict categories, such as sad/happy, yes/no, or dog/cat/bird.

There are many classification models and many algorithms which will help us find them, and we'll see some of the most important ones in the next few chapters. In this chapter we cover the perceptron. The perceptron is a type of model, or classifier, which takes as input the features, and returns a 1 or a 0, which can be interpreted as the answer to a yes/no question. A way to build a good perceptron that fits our data, is via the *perceptron algorithm*. The perceptron algorithm is a beautiful procedure which, in a similar way to the linear regression algorithm, will iterate many times until it finds a good classifier. These two algorithms (and many more that we learn in this book) are very useful for working out patterns in data.

The use case that we use in this chapter is *sentiment analysis*. Sentiment analysis is the branch of machine learning dedicated to predicting the sentiment of a certain piece of text. For example, being able to tell that the sentence "I feel wonderful today!" is a happy sentence, or that the sentence "I am so sad, this is terrible." is a sad sentence.

Sentiment analysis is used in many practical applications. Here are some examples:

- Companies analyzing the conversations between customers and technical support, to see if the service is good.
- Companies analyzing their online reviews and tweets, to see if their products or marketing strategies are well received.
- Twitter analyzing the overall mood of the population after a certain event.
- Stock brokers analyzing the overall sentiment of users towards a company, to decide if they buy or sell their stock.

Let's stop and think for a moment, how would we build a classifier for sentiment analysis? I invite you to put the book down for a couple of minutes, and think of how you would go about building this type of model.

Here's an idea, let's look at the words in the sentence. For simplicity, let's forget the structure of the sentence, and only think of the words in it. We'd imagine that if a sentence has words such as 'wonderful' and 'great', it may be a happy sentence. In contrast, if it has words like 'sad' or 'despair', it may be a sad sentence. How do we inject this logic into a computer? One way to do it is to attach a score to every word, in a way that the happy words have positive scores, and the sad words have negative scores. Furthermore, let's make it so that the happier (or sadder) the word, the higher (or lower) its score is, so that the word 'exhilarating' has a higher score than good, and 'awful' has a lower score than 'bad'. Here are some words and scores off the top of my head:

- Wonderful: 4 points
- Delighted: 3 points
- Happy: 2 points
- Sad: -2 points
- Awful: -4 points
- Every other word: 0 points.

Now we calculate score every sentence by adding the scores of all its words. If the score is positive, then we predict that the sentence is happy, and if the score is negative, we predict that it is sad. For example, let's calculate it the sentences "I feel awful today" and "Everything is wonderful, I am delighted!". We can see that according to our model, the first sentence is sad, and the second one is happy.

**Sentence 1:** I feel awful today

Score: +0 +0 -4 +0 = -4

Prediction: The sentence is sad.

<b>Sentence 2:</b>	Everything	is	wonderful	I	am	delighted
Score:	+0	+0	+4	+0	+0	+3
	= +7					

Prediction: The sentence is happy.

That seems to work. All we have to do now is score all the words in the English language, and we are done. But that is a lot of work. And how do we know we did a good job? These and many other questions may arise in your head, such as the following:

1. How do we know the score of *every* single word?
2. The scores we came up with are based on our perception. How do we know these are the good scores?
3. What if we have to build a sentiment analysis in a language that we don't speak?
4. The sentences "I am not sad, I am happy.", and "I am not happy, I am sad." have completely different sentiments, yet they have the exact same words, so they would score the same, no matter the classifier. How do we account for this?
5. What happens with punctuation? What about sarcastic comments? What about the tone of voice, time of the day, or the overall situation? Too many things to take into account, aaahhh!

The answer to questions 1, 2, and 3, is the topic of this chapter. We'll introduce the perceptron algorithm and use it to train a classifier over a set of data, which will give us scores for each word. The scores of the words are those that best fit the given data, so we have some confidence in their validity. Furthermore, since we are running an algorithm in a dataset, we don't need to speak the language. All we need is to know what words tend to appear more in happy or sad sentences. Therefore, we can build sentiment analysis classifiers in every language, as long as our data consists in a set of sentences with the sentiment specified. What I mean by this is that our dataset has many sentences, and each sentence has a label attached which says 'happy' or 'sad'.

The answer to questions 4 and 5 is more complicated, and we won't deal with it in this chapter. Our classifier will not take into account the order of words, punctuation, or any other external features, and thus, it will make some mistakes. This is ok, the goal is to have a classifier that is correct most of the time, and it is nearly impossible to have a classifier that is *always* correct. And believe it or not, simple classifiers like this tend to do a good job in general.

There are classifiers that take into account the order of the words, and they have great performance. Of course, they require lots of data and more computational power. Although they are out of the scope of this book, I recommend you to look at *recurrent neural networks*, *long short-term memory networks (LSTM)*, and *hidden Markov models*.

But before we delve into looking at data and building our classifiers, let me give you an idea of how the perceptron algorithm works. The idea is to iterate many times and improve our weights. Let's start by assigning random scores to each word. And let's say that our

classifier is terrible. For example, it gives the word ‘awful’ a score of +10. Then we bump into the sentence “Awful!”, which is classified as sad. Our model classifies it as happy, since it assigns it a score of +10. This sentence hints to us that the word ‘awful’ has been assigned a score that is much higher than what it should. Therefore, we proceed to reduce the score of the word ‘awful’ to, say, 9.9. It is a small step, but imagining picking a random sentence and adjusting the scores of the words in it millions of times. You’d imagine that the word ‘awful’ appears in lots of sad sentences, and eventually its score will decrease to something negative. It may increase a couple of times, since ‘awful’ may also appear on some happy sentences, but for the most part, it will decrease. If we do this with all the words at the same time, it is imaginable that one day we’ll have a good classifier.

Ok, time for us to get our hands dirty and build a classifier properly. First let’s start with a simple dataset.

## 4.1 The problem: We are in an alien planet, and we don’t know their language!

Imagine the following scenario. We are astronauts and we have just landed on a distant planet, where a race of unknown aliens lives. We would like to interact a bit with the aliens, but for this, we feel like we need a system that will help us determine if any alien we encounter is happy or sad. In other words, we need to build a sentiment analysis classifier. We befriend four aliens, and we start observing their mood and listening to what they say. We observe that two of them are happy, and two of them are sad. They also keep repeating the same sentence over and over. They seem to only have two words in their language: “aack”, and “beep”. This forms our dataset:

### **Dataset:**

- Alien 1
  - Mood: Happy
  - Sentence: “Aack, aack, aack!”
- Alien 2:
  - Mood: Sad
  - Sentence: “Beep beep!”
- Alien 3:
  - Mood: Happy
  - Sentence: “Aack beep aack!”
- Alien 4:
  - Mood: Sad
  - Sentence: “Aack beep beep beep!”

Data	Prediction
 Aack aack aack!	Is this alien happy or sad?
 Beep beep!	
 Aack beep aack!	
 Aack beep beep beep!	 aack beep aack aack!

**Figure 4.1.** Our dataset of aliens. We have recorded their mood (happy or sad) and the sentence they keep repeating. Now a fifth alien comes in, saying a different sentence. Do we predict that this alien is happy or sad?

Now, a fifth alien comes in, and it says “Aack beep aack aack!”. But we have no idea of their mood. What do we predict, that this alien is happy or sad?

If you said happy, then we are on the same page; I also think the alien is happy. The reason is that, although I don’t know the language, it seems that the word ‘aack’ appears more in happy sentences, while the word ‘beep’ appears more in sad sentences. In some way, even if we don’t know the language, we can figure out that ‘aack’ probably means something positive, like ‘joy’, ‘happy’, or ‘wonderful’, and ‘beep’ probably means something negative, like ‘sadness’, ‘despair’, or ‘awful’.

So our first classifier basically checks which word appears most, and decides the mood from that. Let’s put that in formal terms.

### Simple sentiment analysis classifier

Count the number of appearances of ‘aack’ and ‘beep’. If there are more ‘aack’s, then the alien is happy. If there are more ‘beep’s, then the alien is sad.

Notice that we didn’t specify what happens when the number of ‘aack’s is equal to the number of ‘beep’s. This is ok, we can just say that the classifier is undecided. In the practice, with lots of variables and words, it is very unlikely that a classifier is undecided, so we don’t need to worry much about this. For the rest of this chapter, by default, we’ll say that in this case the classifier predicts that the alien is happy.

We can do something a bit more mathematical. Let's assign a score to each word, as follows.

### **Mathematical sentiment analysis classifier**

The classifier is defined by some scores, and a rule, as follows:

#### **Scores:**

- Aack: 1 point
- Beep: -1 points

#### **Rule:**

Add the scores of all the words.

- If the score is positive or zero, predict that the alien is happy.
- If the score is negative, predict that the alien is sad.

In most situations, it is useful to plot our data, since sometimes nice patterns become visible. In Table 4.1 we have our four aliens, as well as the number of times each said the words 'aack' and 'beep' and their mood.

**Table 4.1. Our dataset of aliens, the sentences they said, and their mood. We have broken each sentence down to its number of appearances of the words 'aack' and 'beep'.**

Sentence	Aack	Beep	Mood
Aack aack aack!	3	0	 Happy
Beep beep!	0	2	 Sad
Aack beep aack!	2	1	 Happy
Aack beep beep beep!	1	3	

			Sad
--	--	--	-----

The plot will consist of two axes, the horizontal (x) axis and the vertical (y) axis. In the horizontal axis we record the number of appearances of 'aack', and in the vertical axis the appearances of 'beep'. This plot can be seen in Figure 4.2.

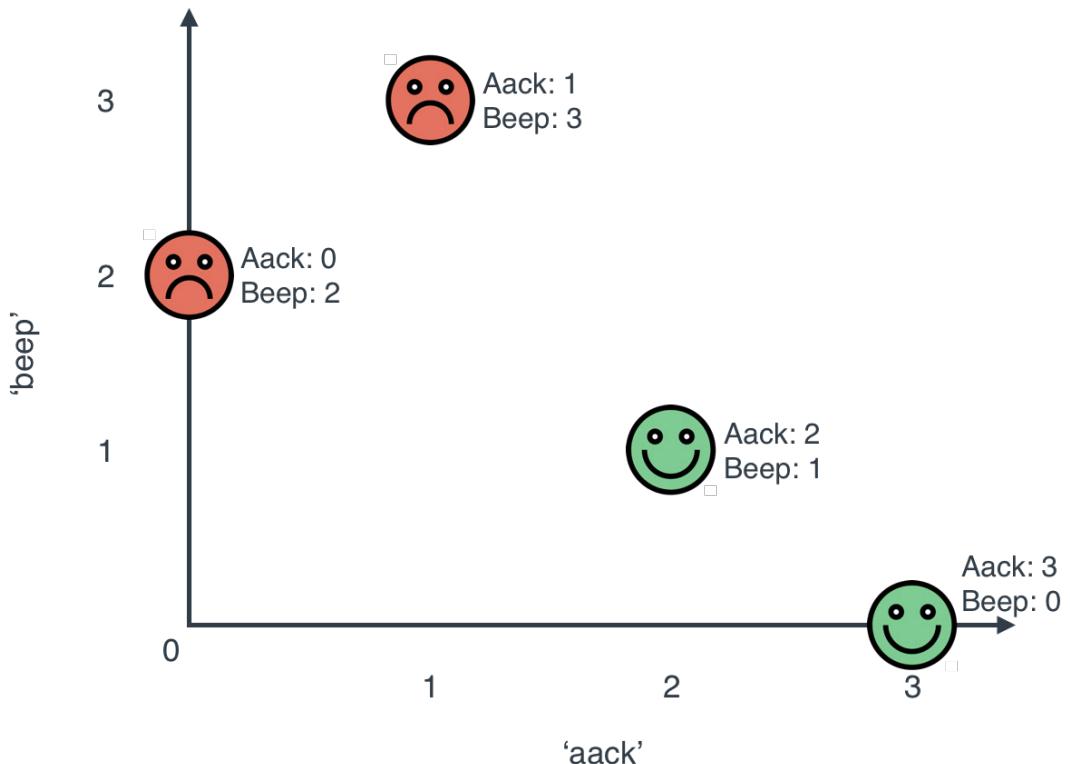


Figure 4.2. A plot of the dataset of aliens. In the horizontal axis we plot the number of appearances of the word 'aack', and in the vertical axis, the appearances of the word 'beep'.

Note that in the plot in Figure 4.2, the happy aliens seem to be located on the bottom right, while the sad aliens are in the top left. This is because the bottom right is the area where the sentences have more aack's than beep's, and the top right area is the opposite. In fact, there is a line dividing these two regions. It is precisely the line formed by all the sentences with the same number of aack's than beep's, namely, the diagonal drawn in Figure 4.3. This diagonal has the equation

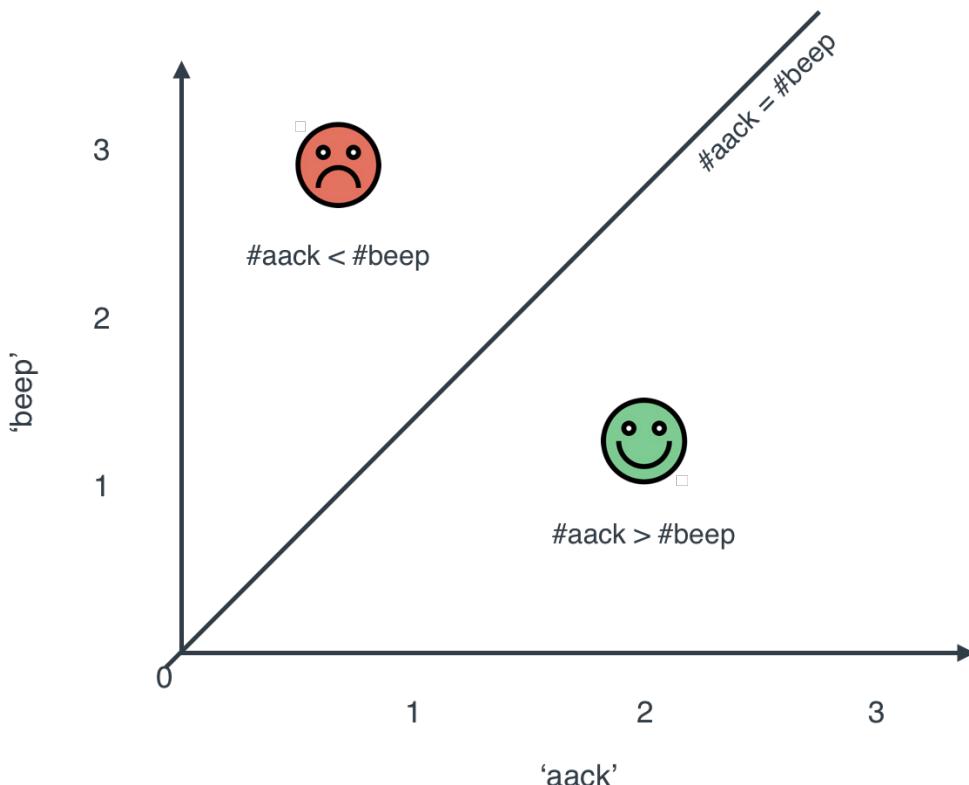
$$\#aack = \#beep.$$

Or equivalently, the equation

$$\#aack - \#beep = 0.$$

If you like equations with variables, this is simply the equation  $y=x$ , or equivalently, the equation  $y-x=0$ . We also have two areas, as follows:

- The positive (happy) area, where the sentences have more aack's than beep's, with equation  $\#aack - \#beep > 0$ .
- The negative (sad) area, where the sentences have more beep's than aack's, with equation  $\#aack - \#beep < 0$ .



**Figure 4.3.** The classifier is the diagonal line that splits the happy and the sad points. The equation of this line is  $\#aack = \#beep$  (or equivalently,  $\#aack - \#beep = 0$ ), since the line corresponds to all the points where the horizontal and the vertical coordinates are equal. The happy zone is the zone in which  $\#aack$  is greater than  $\#beep$ , and the sad zone is the zone in which  $\#aack$  is less than  $\#beep$ .

Thus, we can write our classifier as follows.

### Geometric sentiment analysis classifier

The classifier is defined by a line of equation  $\#aack - \#beep = 0$ , and a rule.

**Rule:**

- If the point is in the positive region of this line, with equation  $\#aack - \#beep > 0$ , or over the line, then the alien is classified as happy.
- If the point is in the negative region of this line, with equation  $\#aack - \#beep < 0$ , then the alien is classified as sad.

It is easy to tell that the three sentiment analysis classifiers are the exact same thing. It is more a matter of preference if you prefer to count words, to add scores for words, or to plot points and see which side of a line they are.

These three classifiers, which at heart are the same one, are examples of a perceptron.

**PERCEPTRON** A perceptron is a classification model which consists of a set of weights, or scores, one for every feature, and a threshold. The perceptron multiplies each weight by its corresponding score, and adds them, obtaining a score. If this score is greater than or equal to the threshold, then the perceptron returns a 'yes', or a 'true', or the value 1. If it is smaller, then it returns a 'no', or a 'false', or the value 0. In our case, the features are the words 'aack' and 'beep', the weights are the scores for each word (+1 and -1), and the threshold is 0.

Now let's look at a slightly more complex example.

#### 4.1.1 A slightly more complicated planet

Let's say we're done with the first planet, and we move to another one. In here, the aliens speak a slightly more complicated language. This one still has two words: 'crack' and 'doink'. The dataset is the following (Table 4.2):

**Dataset:**

- Alien 1
  - Mood: Sad
  - Sentence: "Crack!"
- Alien 2:
  - Mood: Sad
  - Sentence: "Dunk."
- Alien 3:
  - Mood: Sad
  - Sentence: "Crack dunk!"
- Alien 4:
  - Mood: Happy
  - Sentence: "Crack crack dunk dunk."

- Alien 5:
  - Mood: Happy
  - Sentence: "Crack dunk dunk crack crack!"
- Alien 6:
  - Mood: Happy
  - Sentence: "Dunk dunk crack dunk crack!"

**Table 4.2.** The new dataset of aliens. Again, we've recorded each sentence, the number of appearances of each word in that sentence, and the mood of the alien.

Sentence	Crack	Dunk	Mood
Crack!	1	0	Sad
Dunk dunk!	0	2	Sad
Crack dunk!	1	1	Sad
Crack dunk dunk!	1	2	Sad
Crack crack dunk dunk!	1	3	Happy
Crack dunk dunk crack crack!	2	2	Happy

Dunk dunk crack dunk crack!	3	2	 Happy
Crack dunk dunk crack dunk!	2	3	 Happy

This one seems to be less obvious. First of all, would the words 'crack' and 'dunk' have positive or negative scores? Namely, are they happy or sad words? From the first sentences, they sound like sad words, since the aliens who only say 'crack' or 'dunk' are sad. However, if they say words many times, they're happy. It almost seems that the aliens who don't talk much are sad, and those who talk a lot are happy. Could this be the classifier? Let's say that that's our first attempt at a classifier. Let's count the words. The sad aliens said sentences with 1, 1, and 2 words. The happy ones said sentences with 3, 4, and 5 words. There's a rule! If the alien said 4 words or more, then it is happy. If it said 3 words or less, then it is sad. Let's formalize this.

### Simple sentiment analysis classifier

If the number of words spoken by the alien is 4 or more, then the alien is happy. If it is less than 3, the alien is sad.

But let's put that into the same language as the previous classifier. In this case, we can simply attach a score of 1 to each of the words. In our previous classifier, we needed the score to be positive or zero for the alien to be happy, and negative for the alien to be sad. Now, it's a bit more complicated. Now, we can set a threshold of 3.5, and say: If the score of the sentence is 3.5 or more, then the alien is happy. If it's less than 3.5, then the alien is sad. There's nothing special about that 3.5, it could have been 3.8, or 3.1, but we picked the one that is more in the middle. Now we have a more formal classifier.

### Mathematical sentiment analysis classifier

The classifier is defined by the scores, and a rule, as follows:

#### Scores:

- Aack: 1 point
- Beep: 1 points

#### Rule:

Add the scores of all the words.

- If the score is larger than or equal to 3.5, predict that the alien is happy.

- If the score is smaller than 3.5, predict that the alien is sad.

Notice that in the previous classifier, the threshold was 0, and in this one, it is 3.5. In some places, you'll see classifiers defined by scores and a threshold. However, it's more common for the threshold to be equal to zero. In this case, what we do is subtract the threshold from the score. To be more formal, we introduce a score called the bias, which corresponds to no word, it simply gets added to every sentence, and it is equal to the negative of the threshold. In this case, our classifier becomes the following.

### **Modified mathematical sentiment analysis classifier**

The classifier is defined by the scores, a bias, and a rule, as follows:

#### **Scores:**

- Aack: 1 point
- Beep: 1 points
- Bias: -3.5 points

#### **Rule:**

Add the scores of all the words plus the bias.

- If the score is positive or 0, predict that the alien is happy.
- If the score is negative, predict that the alien is sad.

Thus, we can slightly modify our perceptron definition to the following:

**PERCEPTRON** A perceptron is a classification model which consists of a set of weights, or scores, and a bias. The perceptron calculates a score by adding the products of the weights and scores, plus the bias. If this score is greater than or equal to zero, the perceptron returns a 'true' or a value of 1. If the score is negative, the perceptron returns a 'false' or a value of 0.

In the previous example we interpreted our perceptron as a set of scores, and also as a line that split some point in the plane. Let's do this again by plotting the happy and sad aliens just like we did with the first example. We get the plot in Figure 4.4.



**Figure 4.4.** The plot of the new dataset of aliens. Notice that the happy ones tend to be above and to the right, and the sad ones below and to the left.

Again, our classifier will consist of a line that splits the happy and the sad faces. Many lines work, and you may think of some that are equally good or better than mine, but I'll use the one I've drawn in Figure 4.5. This line consists of all the points where the sum of the horizontal coordinate (number of 'crack's) and the vertical coordinate (number of 'dunk's) is equal to 3.5. In other words, the equation of this line is:

$$\#crack + \#dunk = 3.5.$$

For convenience, we'll write the equation as follows:

$$\#crack + \#dunk - 3.5 = 0.$$

In that way, we have our geometric interpretation of the perceptron.

### Geometric sentiment analysis classifier

The classifier is defined by a line of equation  $\#crack + \#dunk - 3.5 = 0$ , and a rule.

#### Rule:

- If the point is in the positive region of this line, with equation  $\#crack - \#dunk - 3.5 > 0$ , then the alien is classified as happy.
- If the point is in the negative region of this line, with equation  $\#crack - \#dunk - 3.5 < 0$ , then the alien is classified as sad.

Again, one can tell that the three sentiment analysis classifiers are the exact same thing.



**Figure 4.5.** The classifier for the new dataset of aliens. It is again a line that splits the happy and the sad aliens.

Notice something interesting in Figure 4.5. The y-intercept, or the point in which the boundary line meets the horizontal (y) axis is precisely 3.5. This is the threshold, or the negative of the bias. This observation also has a mathematical and a logical meaning, as we'll see in the next section.

#### 4.1.2 The bias, the y-intercept, and the inherent mood of a quiet alien

Well are we getting philosophical here? Why not! Let's tackle a question that may have come to your head now. What is the bias? Why would we add a number to the score of every sentence, instead of simply adding the scores of the words? Well, in the second example, we saw that only scoring the words is not enough, and a threshold was needed, which then turned into the bias. But that doesn't really answer the question.

A related question is, what happens when an alien says absolutely nothing. Is that alien happy or sad? In the latest model, if an alien says nothing, then the score of the sentence is 0 (since there are no words) *plus* the bias, which is -3.5. This is a negative number, so the alien is predicted to be sad. Therefore, we can see the bias as the inherent mood of the alien. This mood *plus* the scores of the words, form the happiness score of the alien.

What if a model had a positive bias? Then an alien who says nothing would score positively, and thus, the alien would be predicted as happy.

Can you help me think of two scenarios in sentiment analysis, one where the threshold is positive, and one where it's negative? Here's what I can think, based on my experience, but maybe you can find better examples!

- **Positive threshold (or negative bias):** This is a sentiment analysis model where saying nothing is a sad sentence. I think that if we look at conversations between humans when they bump into each other in the street, this model would work. Imagine bumping into a friend of yours in the street, and they say nothing. I would imagine that they're pretty upset! This would mean that in this model, the threshold is positive, since we need to say a few words and make sure the sentence gets a high score in order to convey happiness.
- **Negative threshold (or positive bias):** This is a sentiment analysis model where saying nothing is a happy sentence. I think of online reviews of products restaurants. Normally when I'm happy with a product or a service, I give the place 5 stars, and I don't find the need to elaborate on a comment. But when I'm very upset, I give it 1 star and I write a long review of why I'm upset. Maybe this is just me, but I feel (and looked at data), and noticed that many people are like that. In this case, if we were to train a model on these type of customer comments, I imagine the threshold in this model would be negative.

As we saw at the end of the previous section, the bias is the negative of the y-intercept. What does this mean? Well, let's think graphically. If the boundary line is above the origin (the point with coordinates (0,0)), then the origin is in the sad region. If it is above, the point is in the happy region. The point with coordinates (0,0) is precisely the point corresponding to the sentence with no words. This is illustrated in Figure 4.6.

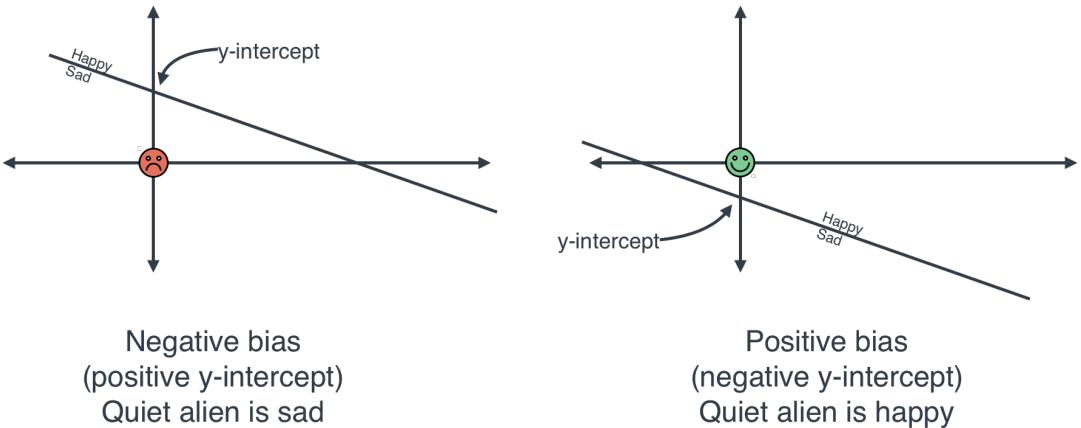


Figure 4.6. Left: the classifier has a negative bias, or a positive threshold (y-intercept). This means that the alien that doesn't say anything falls in the sad zone, and is classified as sad.

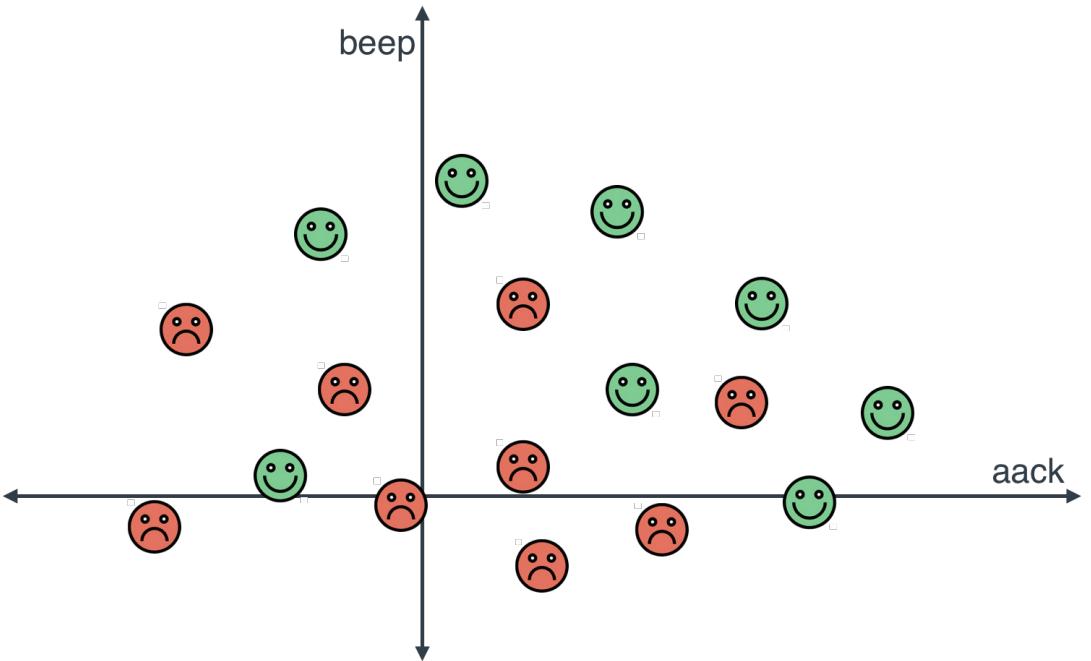
Right: The classifier has a positive bias, or a negative threshold (y-intercept). This means that the alien that doesn't say anything falls in the happy zone, and is classified as happy.

#### 4.1.3 More general cases

We've figured out classifiers look in a language with two words, they basically look like one of the following two:

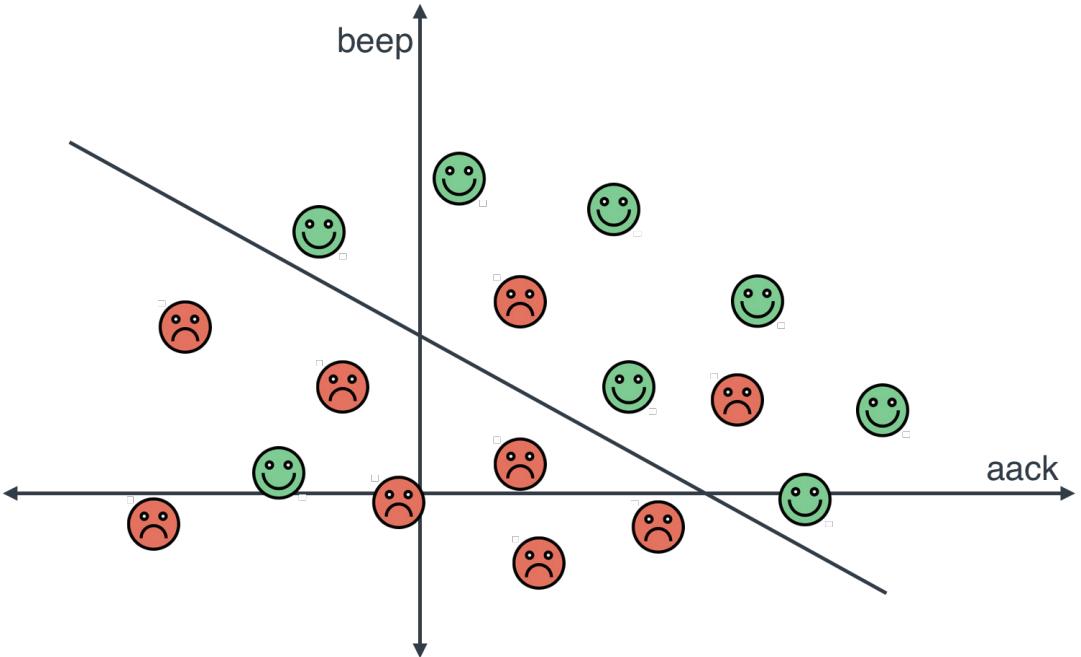
1. **Mathematical classifier:** A score attached to each of the words, plus a bias score that we add to every sentence. If the sentence scores positive (or 0), it is classified as happy, and if it scores negative, it is classified as sad.
2. **Geometric classifier:** A line that splits two kinds of points in the plane. A sentence corresponding to a point above (or over) the line is classified as happy, and one below the line is classified as negative.

A general case will look more complicated than what we've seen so far. We could have positive and negative coordinates, coordinates that are not integers, etc. Even more, it's not very likely that one line will be able to split all the points. We could have an arrangement of points such as Figure 4.7.



**Figure 4.7.** A larger arrangement of points. Note that this one is impossible to split using a line. This is ok, we can find a split that does a good job, namely, not too many mistakes. Can you find some good lines that split this dataset well?

For this dataset, we need to find the line that *best* splits it, even if it makes some mistakes. Figure 4.8 shows a possible solution. This solution only makes three mistakes, since it mistakenly classifies two sad aliens as happy, and one happy alien as sad.



**Figure 4.8.** This line splits the dataset well. Note that it only makes 3 mistakes, 2 on the happy zone, and one on the sad zone.

This line could have any equation. Since the coordinates are the number of appearances of 'aack' and 'beep', the equation could be, for example:

$$1.9 \cdot \text{'aack'} - 3.1 \cdot \text{'beep'} - 2.2 = 0$$

This means our classifier assigns 1.9 points to the first word ('aack'), 3.1 points to the second word ('beep'), and the threshold is 2.2.

Also, as you can imagine, alphabets can have many more than two words. Let's look at what happens then.

### ALPHABETS WITH 3 WORDS

Let's say that we have a classifier with three words, 'aack', 'beep', and 'crack'. Now we can't draw it in 2 dimensions, but we can switch to 3. We have 3 axes, one for each word, and the points will be floating in space. For example, a sentence with the word 'aack' appearing 5 times, the word 'beep' appearing once, and the word 'crack' appearing 8 times, will be at the point with coordinates (5, 1, 8). This is illustrated in Figure 4.9.

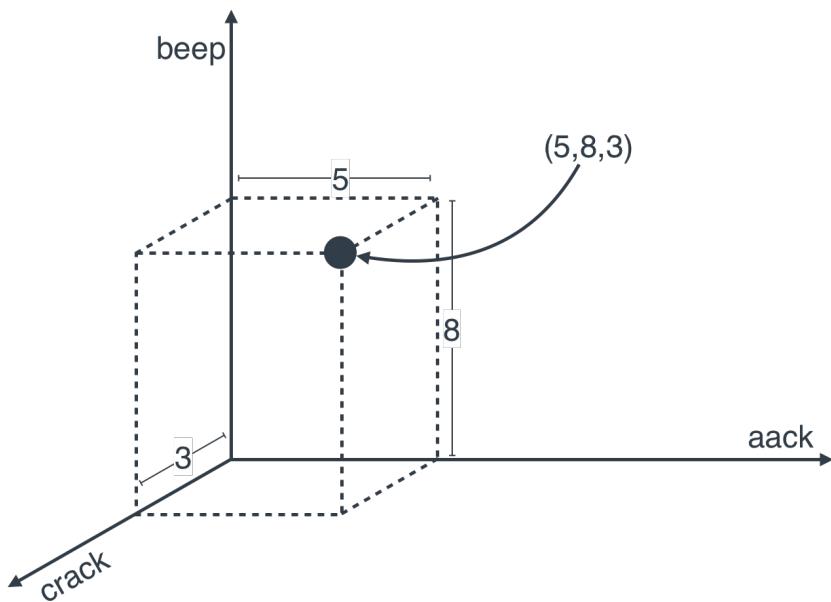


Figure 4.9. A sentence with three words can be plotted as a point in space. In this case, a sentence with the word 'aack' 5 times, 'beep' 8 times, and 'crack' 3 times, is plotted in the point with coordinates (5,8,3).

In this case, a dataset could look like Figure 4.10.

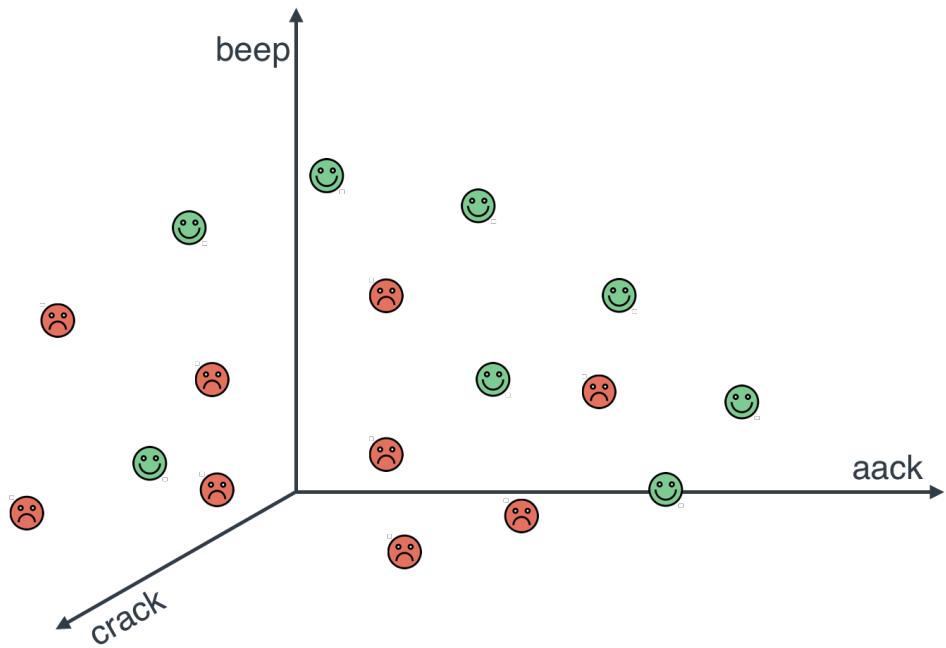
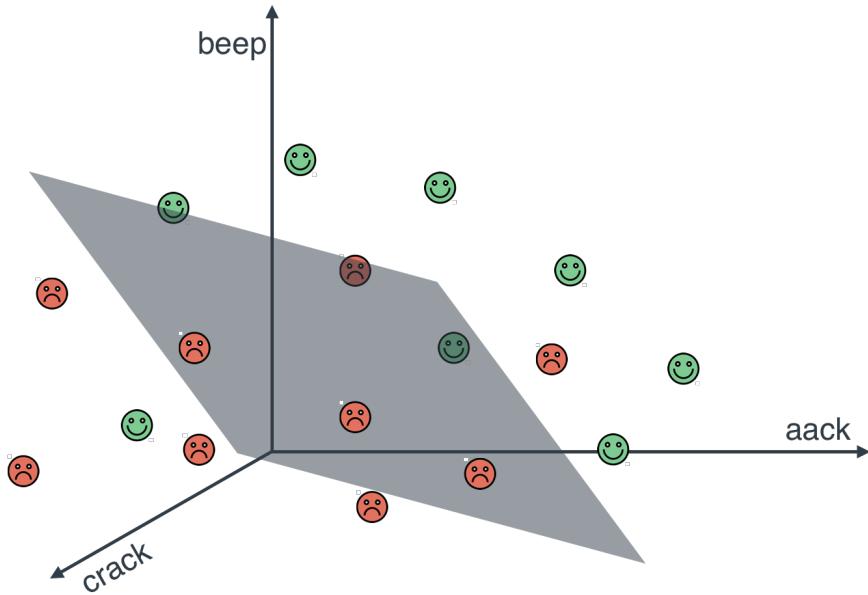


Figure 4.10. A dataset of sentences with three words gets plotted in three dimensions.

How would a classifier look on this dataset? It wouldn't be a line any more, because the points are in space. It would be a plane that cuts the space in two, trying its best to leave all the happy points on one side, and the sad points on the other side. A good separating plane looks like the one in Figure 4.11.



**Figure 4.11.** In the 3-dimensional case, we would need a plane to split our points, instead of a line.

Now, how does the equation of a plane look? If the equation of a line looked like a linear combination of the two words plus a bias

$$2 \cdot \text{'aack'} + 3 \cdot \text{'beep'} - 2.5 = 0,$$

then the equation of a plane looks exactly the same, except with 3 words instead of 2.

$$2 \cdot \text{'aack'} - 3 \cdot \text{'beep'} + 1 \cdot \text{'crack'} - 2.5 = 0.$$

This classifier above assigns the following scores to the words:

- 'Aack': 2 points
- 'Beep': -3 points
- 'Crack': 1 point.
- Threshold: 2.5 points.

What does this mean? Well, for once, we can see that 'aack' is a pretty happy word, 'crack' is a happy word since its weight is 2, (although not as happy as 'aack', with weight 1) and 'beep' is a very sad word with weight -3. Furthermore, the threshold is 2.5, so an alien that is quiet is inherently sad.

## **ALPHABETS WITH MANY WORDS**

What if our alphabet has more than 3 words. What if it has thousands of words? Now we can't visualize it, because unfortunately humans can only see in 3 dimensions (although if you want to imagine happy and sad faces living in a 1000-dimensional universe, that's how it would look!). However, we can still look at the classifier as a set of scores. For example, if we were to run a classifier on the English language, with thousands of words, we could end up with something as follows:

### **Scores:**

- A: 0.1 points
- Aardvark: 1.2 points
- Aargh: -4 points
- ...
- Zygote: 0.4 points.

### **Threshold:**

- 2.3 points.

Ok, now a question is, how do we come up with these scores and this threshold? And even for the simpler classifiers, how do we come up with the scores of 'aack' and 'beep'? In other words, how do we draw a line, or a plane, that splits happy and sad points in the best possible way? This is what I'll explain in the remainder of the chapter. I will show you two very similar ways of doing this called the perceptron algorithm and logistic regression.

## **4.2 How do we determine if a classifier is good or bad? The error function**

The 1 million dollar question here is: How do we find these classifiers? We need to find an algorithm, or a sequence of steps, which receives as input a dataset of sentences with 'happy' and 'sad' labels attached to them, and returns a set of scores for each of the words, plus a threshold. This will be done in a similar way as we did for regression in chapter three, namely:

1. Start with a set of random scores, and a random threshold.
2. Take a step to improve these scores just a little bit.
3. Repeat step 2 many times.
4. Enjoy your classifier!

But before this, we'll come up with a way to quantify how good our classifier is. This way is called an error function. If we have a two classifiers, a good one and a bad one, the error function will assign the first (good) classifier a small number, and the second (bad) classifier a high number.

First, let's start with the error function.

### 4.2.1 How to compare classifiers? The error function

Let me give you two classifiers, one good one and one bad one, and you can help me assign each one an error score. We'll represent them as a line that tries to separate happy and sad points in the plane as best as possible. Since classifiers specifically predict a point as happy or sad, we'll actually specify which side of the line corresponds to either happy or sad points. The two classifiers are in Figure 4.12.

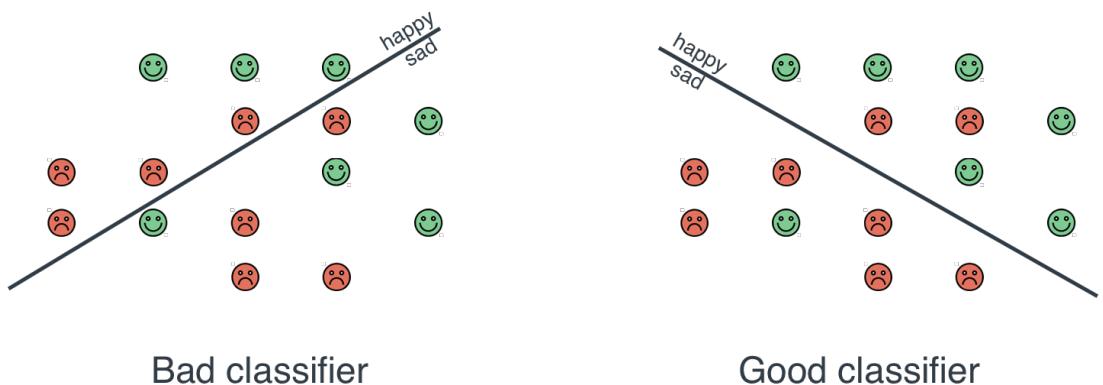


Figure 4.12. Left: On the left, we have a bad classifier, that doesn't really split the points well. On the right we have a good classifier.

#### ERROR FUNCTION 1: NUMBER OF ERRORS

Here's an idea, the error of a classifier is the number of points it classifies incorrectly. In summary, the way we calculate this error is as follows:

- Points that are correctly classified produce an error of 0.
- Points that are misclassified produce an error equal to the distance from that point to the line.

In this case, the bad classifier has an error of 8, since it erroneously predicts 4 happy points as sad, and 4 sad points as happy. The good classifier has an error of 3, since it erroneously predicts 1 happy point as sad, and 2 sad points as happy. This is illustrated in Figure 4.13.

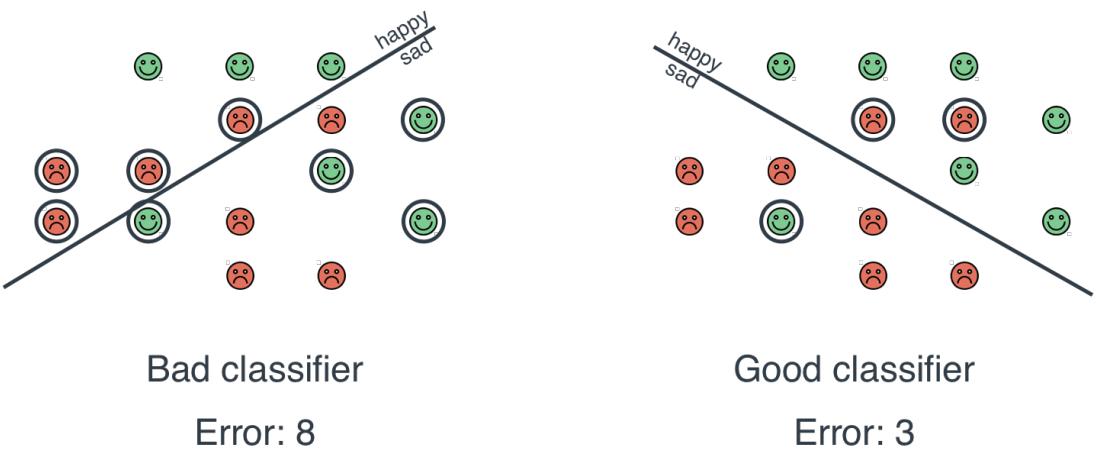
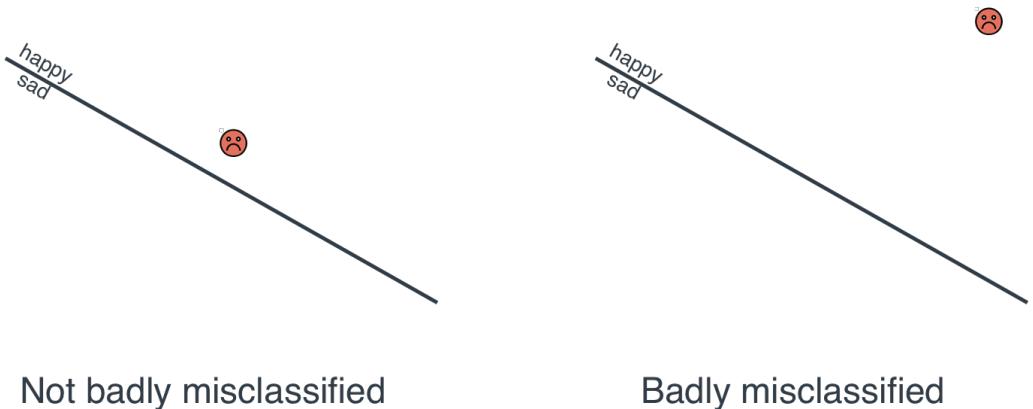


Figure 4.13. The way to tell that the left classifier is bad and the right one is good is by counting the number of mistakes each one makes. The one in the left makes 8 mistakes, while the one in the right only makes 3.

This is a good error function. But it's not a perfect error function. We want one that tells us a lot about the classifier, and counting the number of errors is not descriptive enough. In particular, it treats all errors equally, when normally this is not the case. A misclassified point can be close to being correctly classified, or horribly misclassified, and this error function will treat them equally. Think about this, if a classifier has a threshold of, say, 4, and a sad sentence scores 4.1, then the classifier predicts it as happy, since its score is higher than the threshold. The model made a mistake, but not as bad a mistake as it given the sentence a score of 100. These two mistakes should be treated differently. Graphically, a point that is misclassified, though not very badly, is a point which is on the wrong side of the line, but close to the line. In contrast, a point that is badly misclassified is also on the wrong side of the line, but it is very far from it. Figure 4.14 shows two sad points that are incorrectly classified as happy. The one on the left is not very badly misclassified, while the one on the right is badly misclassified. However, our error function counts both of them as one error.

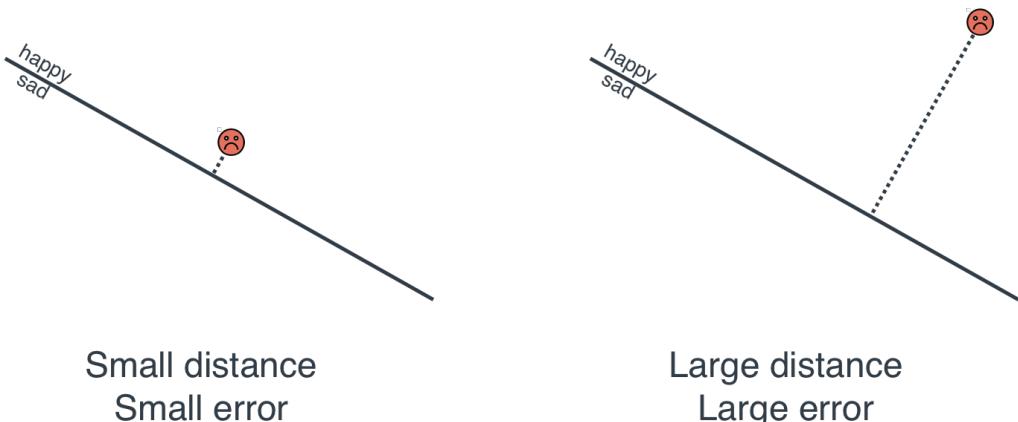


**Figure 4.14.** These two points are misclassified, but at a different degree. The point in the left is not so badly misclassified, since it is close to the boundary line. The one in the right, in contrast, is very badly misclassified point, since it is very far from the boundary line.

We would like to assign a small error to the point in the left, and large error to the point in the right, in Figure 4.14. Any ideas?

#### **ERROR FUNCTION 2: DISTANCE**

In Figure 4.15, what tells the two points apart is that the one in the left is close to the line, while the one in the right is far from the line. Therefore, if we simply consider the distance from the point to the line, we have a very good error function.

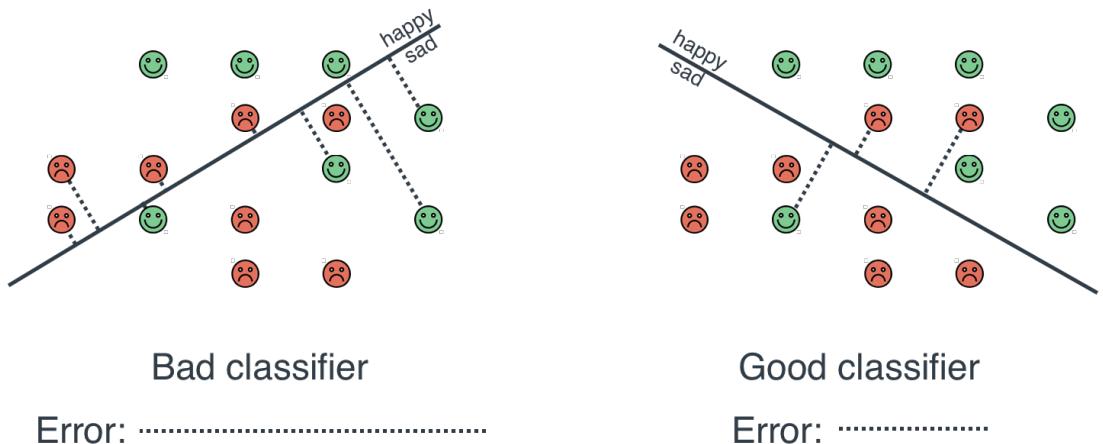


**Figure 4.15.** The way to tell how badly misclassified a point is, is by first making sure that it is in the wrong side of the line (this one is sad, and it lies in the happy zone), and second by checking how far it is from the boundary line. The point in the left is at close to the line, so its error is small, and the point in the right is far from the line, so its error is large.

This is a much better error function. In summary, what this error function does is the following:

- Points that are correctly classified produce an error of 0.
- Points that are misclassified produce an error equal to the distance from that point to the line.

Let's go back to the good and the bad classifier. The way we calculate the errors is by adding the error corresponding to every point. This means that we only look at the misclassified points, and we add the perpendicular distances from these points to the line. Notice that the bad classifier has a large error, while the good classifier has a small error.



**Figure 4.16.** In order to calculate the total error of a classifier, we add up all the errors (distances). Recall that only the misclassified points create an error. When we add the distances in the classifier in the left, we get a large error, while with the one in the right we get a small error. Thus, we conclude that the classifier in the right is better.

So this is a good error function. Is it the one we'll use? Unfortunately no, it misses one thing: it is too complex. Let's think mathematically, how does one calculate a distance? One uses the Pythagorean theorem. The Pythagorean theorem has a square root. Square roots have an ugly derivative. It turns out that in our calculations, having an ugly derivative will complicate things unnecessarily. However, there is a nice fix to this. We'll come up with a similar, yet slightly different error function, which will capture the essence of the distance, but which fits our calculations very well.

### ERROR FUNCTION 3: SCORE

Recall that classifiers can be thought of both geometrically and mathematically. In order to construct this error function, we'll use the mathematical definition. Let's say that our classifier has the following scores and threshold:

#### Scores:

- 'Aack': 1 pt.
- 'Beep': 2 pts.
- **Bias:** -4 pts.

The classifier looks like Figure 4.17., where the score of each sentence is  $\#aack + 2\#\text{beep} - 4$ . The boundary line is where this score is 0, the happy zone is where this score is positive, and the sad zone is where this score is negative.

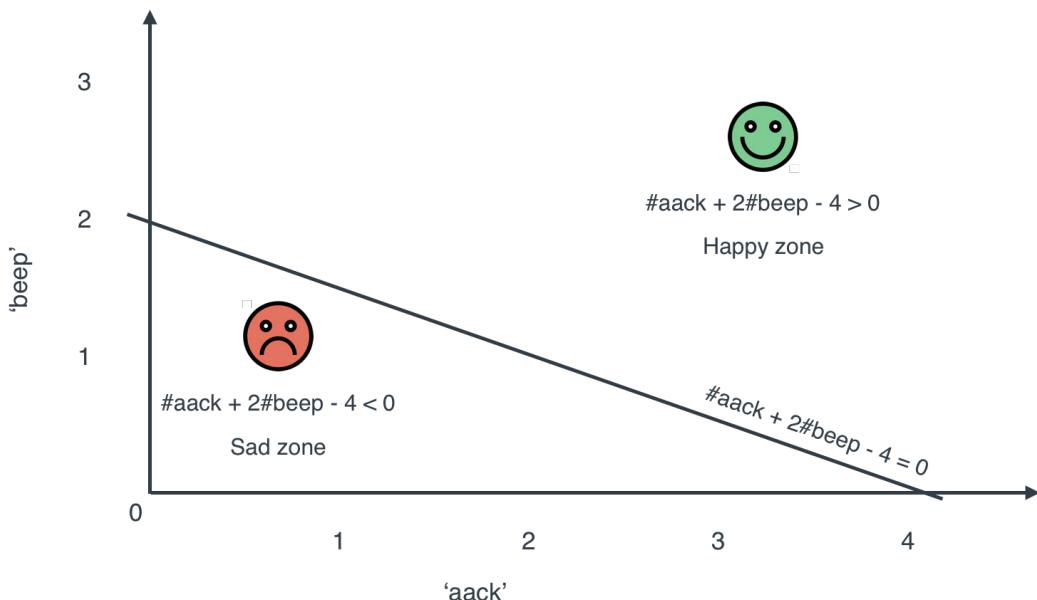
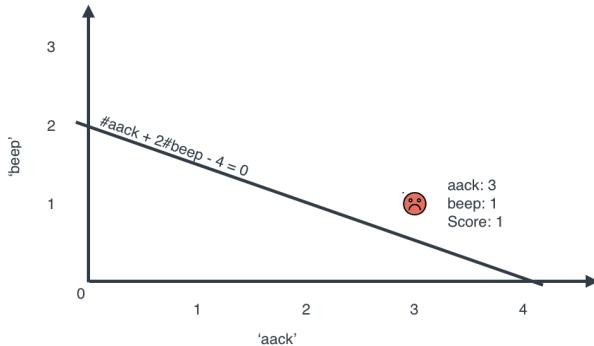


Figure 4.17. Our classifier has equation  $\#aack + 2\#\text{beep} - 4 = 0$ . The happy zone is where this equation returns a positive number, and the sad zone is where it returns a negative number.

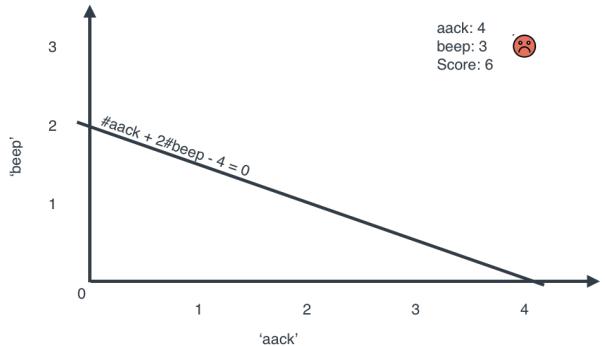
Now, let's say that we have two points misclassified by this classifier. They will be the following two sad sentences:

- Sentence 1: "Aack aack aack beep."
- Sentence 2: "Aack aack aack aack beep beep beep."

Since the horizontal coordinate is the number of appearances of the word 'aack', and the vertical coordinate is the number of appearances of the word 'beep', then Sentence 1 is in position (3,1), and Sentence 2 in position (4,3).



Not very badly classified point



Very badly classified point

**Figure 4.18.** As before, we have two misclassified points. The one on the left is not so badly classified, since it is close to the line, whereas the point in the right is far from the line, so it is badly misclassified. Notice that the score of the point in the left is 1, while the score of the point in the right is 6. Thus, the score is higher if the point is badly misclassified.

Furthermore, the scores are the following:

- Sentence 1, coordinates (3,1):
  - Score  $= \#aack + 2\#\text{beep} - 4 = 1$ .
- Sentence 2, coordinates (4,3):
  - Score  $= \#aack + 2\#\text{beep} - 4 = 6$

Notice that Sentence 1 is badly misclassified, although only by a little bit. However, Sentence 2 is very badly misclassified. Therefore, the score is a good measure of the error for a misclassified point. We'll take that as our error function.

What if we have happy points misclassified, would the story be the same? Almost. Let's say we have the following two sentences, that are happy:

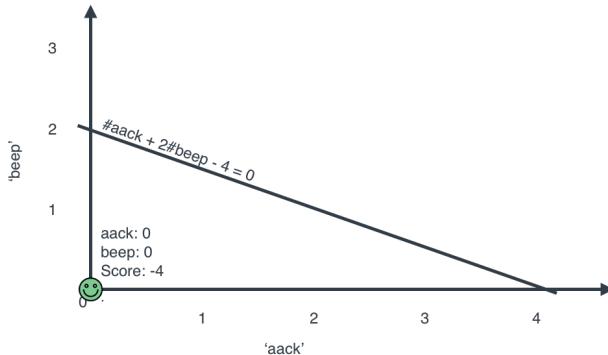
- Sentence 3: ""
- Sentence 4: "Aack aack beep."

Sentence 1 is the empty sentence, it corresponds to an alien who said nothing. Let's evaluate the scores of these sentences.

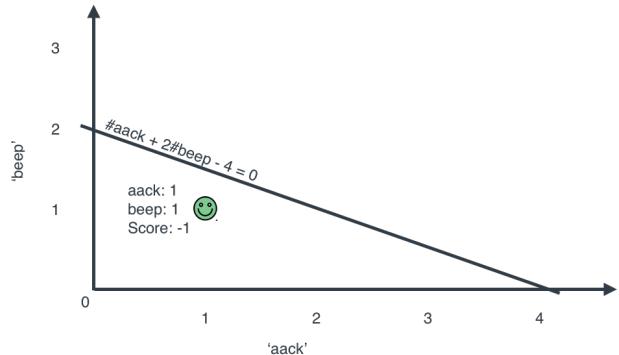
- Sentence 3, coordinates (0,0):
  - Score  $= \#aack + 2\#\text{beep} - 4 = -4$ .

- Sentence 4, coordinates (1,1):
  - Score = #aack + 2#beep - 4 = -1

In this case, as you can see in Figure 4.19, Sentence 3 is very badly misclassified, while Sentence 4 is misclassified, but not as badly, since it is much closer to the boundary line.



Very badly classified point



Not very badly classified point

**Figure 4.19.** In a similar fashion as in Figure 4.18, we now have two misclassified points, except this time they are happy points in the sad zone. Notice that the error of the badly misclassified point is -4, while the error in the not so badly misclassified point is -1. Thus, the point with a higher absolute value error is misclassified worse.

Here, the badly classified point has a score of -4, and the not very badly classified point has a score of -1. Therefore, a sensible error for these points are 4 and 1, respectively. This means, if the point is labelled 'happy' and it is misclassified, we should take its error to be the negative of the score.

In other words, the error is defined as follows:

- If the point is correctly classified:
  - Error = 0
- Else, if the point is incorrectly classified:
  - If the point has a positive label, and the prediction is negative:
    - Error = -Score
  - Else, if the point has a negative label, and the prediction is positive:
    - Error = Score.

But this can be made even simpler. Notice that if a point has a positive label, but it's predicted negative, then its score must be negative. Likewise, if a point has a negative label, but it's predicted positive, then its score must be positive. In both cases, we end up with the error

being the absolute value of the score, since an error should always be positive or zero. Therefore, the definition of error is the following:

### Perceptron error

- If the point is correctly classified:
  - Error = 0
- Else, if the point is incorrectly classified:
  - Error = |Score|.

### Example

Let's do an example of two classifiers with different errors. Our dataset is the following four sentences:

- **Sentence 1** (sad): "Aack."
- **Sentence 2** (happy): "Beep."
- **Sentence 3** (happy): "Aack beep beep beep."
- **Sentence 4** (sad): "Aack beep beep aack aack."

And we'll compare the following two classifiers:

**Classifier 1:**  $\#aack + 2\#beep - 4 = 0$

**Classifier 2:**  $-\#aack + \#beep = 0$

The points and the classifiers can be seen in Figure 4.20.

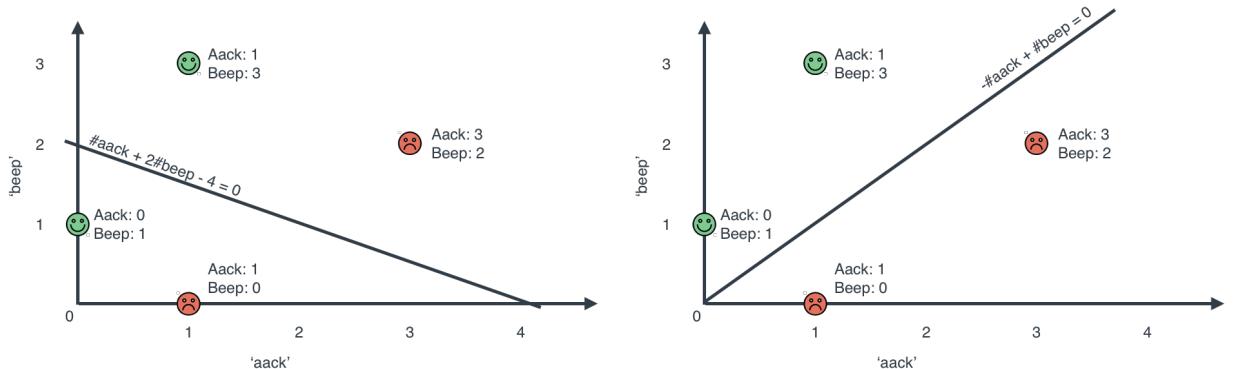


Figure 4.20. In the left we have a Classifier 1, and in the right we have Classifier 2.

### Calculation of error for Classifier 1:

Notice that Classifier 1 classifies them as follows:

- Sentence 1:

- Coordinates = (1,0)
- Score = #aack + 2#beep - 4 = -3
- Prediction: Sad
- Sentence 2:
  - Coordinates = (0,2)
  - Score = #aack + 2#beep - 4 = -2
  - Prediction: Sad
- Sentence 3:
  - Coordinates = (1,3)
  - Score = #aack + 2#beep - 4 = 3
  - Prediction: Happy
- Sentence 4:
  - Coordinates = (3,2)
  - Score = #aack + 2#beep - 4 = 3
  - Prediction: Happy

Now on to calculate the errors. Since sentences 1 and 3 are correctly classified, they produce 0 error. Sentences 2 and 4 are incorrectly classified, so they produce an error. Sentence 2 has a score of -2, so its error is 2. Sentence 4 has a score of 3, so its error is 3. In summary:

- Sentence 1:
  - Correctly classified
  - Score = -3
  - **Error = 0**
- Sentence 2:
  - Incorrectly classified
  - Score = -2
  - **Error = 2**
- Sentence 3:
  - Correctly classified
  - Score = 3
  - **Error = 0**
- Sentence 4:
  - Incorrectly classified
  - Score = 3
  - **Error = 3**

The total error for Classifier 1 is the sum of errors, which is  $0+2+0+3 = 5$ .

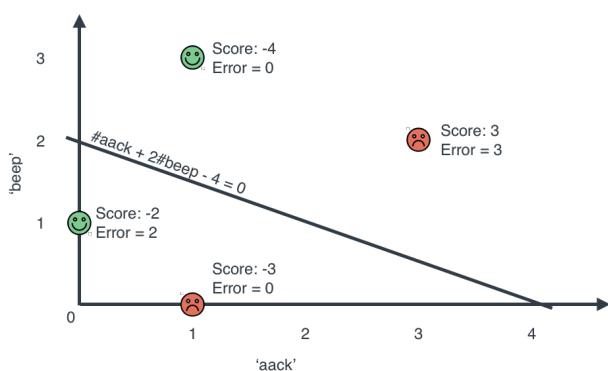
#### **Calculation of error for Classifier 1:**

This one is much easier. Notice that Classifier 2 classifies every point correctly. To verify this, notice that:

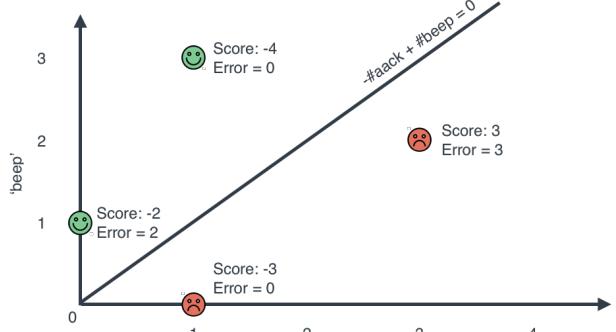
- Sentence 1 (sad):
  - Coordinates = (1,0)
  - Score =  $-\#aack + \#beep = -1$
  - Prediction: Sad
- Sentence 2 (happy):
  - Coordinates = (0,2)
  - Score =  $-\#aack + \#beep = 2$
  - Prediction: Happy
- Sentence 3 (happy):
  - Coordinates = (1,3)
  - Score =  $-\#aack + \#beep = 2$
  - Prediction: Happy
- Sentence 4 (sad):
  - Coordinates = (3,2)
  - Score =  $-\#aack + \#beep = -1$
  - Prediction: Sad

Therefore, all points, being correctly classified, produce an error of zero. Thus, the total error is zero.

We then conclude that Classifier 2 is better than Classifier 1. The summary of these calculations is in Figure 4.21.



$$\text{Error} = 0 + 2 + 0 + 3 = 5$$



$$\text{Error} = 0 + 0 + 0 + 0 = 0$$

**Figure 4.21.** Classifier 1 has an error of 5, while Classifier 2 has an error of 0. Thus, we conclude that Classifier 2 is better than Classifier 1.

Now that we know how to compare classifiers, let's move on to finding the best one of them, or at least, a pretty good one.

## 4.3 How to find a good classifier? The perceptron algorithm

In order to build a good classifier, we'll follow a very similar approach as we followed with linear regression on Chapter 3. We start with a random classifier, and slowly improve it until we have a good one. We'll also use the error to tell us when it's okay to stop improving it. Here is some pseudocode of how the algorithm looks. You'll find it very similar to the linear regression algorithm in Chapter 3.

### Pseudocode for the perceptron algorithm:

- Begin with a random classifier
- Loop many times:
  - Improve the algorithm a small amount
- Output a good classifier.

By now, your head may be full of questions, such as the following:

1. What do we mean by making the classifier a little better?
2. How do we know this algorithm gives us a good classifier?
3. How many times do we run the loop?

The answers to all these questions are in the next few sections. Let's begin with question 1: What do we mean by making the classifier a little better? This is what we call the perceptron trick.

### 4.3.1 The perceptron trick

The perceptron trick is a tiny step that helps us go from a classifier, to a slightly better classifier. We'll actually take a less ambitious step. Like in Chapter 3, we'll only focus on one point, and ask the question: "How can I make this classifier *a little bit better* for this one point?

We'll take a slightly arbitrary approach (if you don't like it and have ideas to improve it, you'll really enjoy the later parts of this chapter!). The arbitrary step is the following:

- If the point is correctly classified, we won't touch the classifier. It's good as it is.
- If the point is misclassified, we will move the line a slight amount towards the point. Why? Because if the point is in the wrong side of the line, then we'd actually like the line to move over the point, putting it in the correct side. Since we only want to make small steps, moving the line slightly towards the point is a good start.

Figure 4.22 summarizes our arbitrary step. We'll call this the perceptron trick.

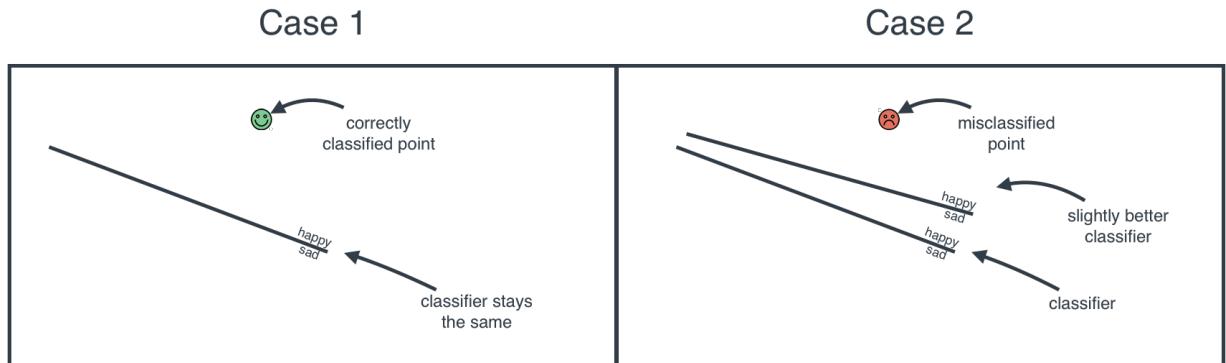


Figure 4.22. Case 1 (left): A point that is correctly classified tells the line to stay where it is.

Case 2 (right): A point that is misclassified tells the line to move closer towards it.

Now the question is, how much do we move the line? Like all we do in machine learning we'll move it a very tiny amount. Since we repeat this algorithm many times, the line will eventually make significant moves. The way we control the size of the step is, like in Chapter 3, with a *learning rate*. We pick a very small number for our learning rate. For the duration of this chapter, let's pick 0.01.

Let's focus on the case when a point is misclassified, and see how much we need to change the weights in order to move the line. We look at an example. Our classifier is the same as before, in which the scores of the words 'aack' and 'beep' are both 1, and the bias is -4 (which means the threshold is 4). Therefore, the equation for the score of a sentence is  $1 \cdot \#aack + 1 \cdot \#beep - 4$ . When the score is positive or zero, the classifier predicts that the sentence is happy, and when it's negative it predicts that the sentence is sad.

In other words, our classifier, called the 'OK Classifier' is defined by the following scores:

### OK Classifier

- 'Aack': 1 point
- 'Beep': 1 point
- Bias: -4 points.

Now, let's pick a misclassified point. Let's pick the following sentence:

- **Sentence 1 (sad):** 'Aack beep beep beep aack beep beep!'.

Why is this a misclassified point? To start, the sentence comes with a sad label. But it contains the word 'aack' 2 times and 'beep' 5 times, so its score is  $2+5-4$ , which is 3. Since 3 is positive, the sentence gets classified as happy. Since it is labelled sad, then the OK Classifier misclassifies it. Furthermore, the error, as we've defined it above, is equal to the score, which is 3.

Let's try to build a classifier called "Better Classifier", which may still misclassify the point, but with a smaller error. We'll do the following (again slightly arbitrary) step. We want to decrease the error, which is equal to the score of the sentence. Therefore, we need to reduce the score of the sentence by a small amount. The score is the sum of the scores of the words plus the bias, so we simply reduce all these a small amount. We'll do the following, remembering that the learning rate we're using in this chapter is 0.01:

- Since the word 'aack' appears twice, we'll reduce its score by two times the learning rate, or 0.02.
- Since the word 'beep' appears five times, we'll reduce its score by five times the learning rate, or 0.05.
- Since the bias only adds to the score once, we reduce the bias by the learning rate, or 0.01.

We obtain the following classifier.

### Better Classifier 1

- 'Aack': 0.98 points
- 'Beep': 0.95 points
- Bias: -4.01 points.

Now, what is the score that the Better Classifier gives this sentence? It is

$$0.98 \cdot \#aack + 0.95 \cdot \#beep - 4.01 = 0.98 \cdot 2 + 0.95 \cdot 5 - 4.01 = 2.7.$$

We did it! The Better Classifier has a smaller score on this sentence than the OK Classifier. It hasn't improved by much, but imagine doing this step many times, always picking some point in our set. Some steps may mislead us, but the majority will take us closer to at least some misclassified point, thus improving our classifier on average.

But why the logic of decreasing the score of 'aack' only by 0.02 and the score of 'beep' by 0.05? Well, since beep appears much more than 'aack' in the sentence, we can assume that the score of 'beep' is more crucial to our score, and needs to be decreased more. In reality, these numbers come out of calculating a derivative in a step called gradient descent. If you're curious and want to look at the calculus, it is all in the Appendix at the end of the book.

Now, that was for a sad sentence that got classified as happy. What about the opposite? Let's say we again have the Ok Classifier, and the following sentence:

- **Sentence 2 (happy):** 'Aack.'

This is again a misclassified point, and let's see why. The sentence has a happy label. However, its score is  $1 \cdot 1 + 1 \cdot 0 - 4 = -3$ , as it contains the word 'aack' once and the word 'beep' zero times. Since the score is negative, the classifier predicts that the sentence is sad.

In this case, the logic is the following. In order for the classifier to be more correct about this sentence, it needs to give it a larger score. Therefore, the word 'aack' should have a

higher score, and the bias should also be higher. The word ‘beep’ is irrelevant here. Since the word ‘aack’ appears once, we’ll increase it by one times the learning rate. We’ll also increase the bias by the learning rate. We get the following classifier:

### Better Classifier 2

- ‘Aack’: 1.01 points
- ‘Beep’: 1 point
- Bias: -3.99 points.

Let’s summarize these two cases and obtain the pseudocode for the perceptron trick.

### Perceptron trick (pseudocode):

Input:

- A classifier with the following scores:
  - Score of ‘aack’:  $a$ .
  - Score of ‘beep’:  $b$ .
  - Bias:  $c$ .
- A point with coordinates  $(x, y)$  (where  $x$  is the number of appearances of the word ‘aack’, and  $y$  of the word ‘beep’).
- A learning rate  $\eta$ .

Procedure:

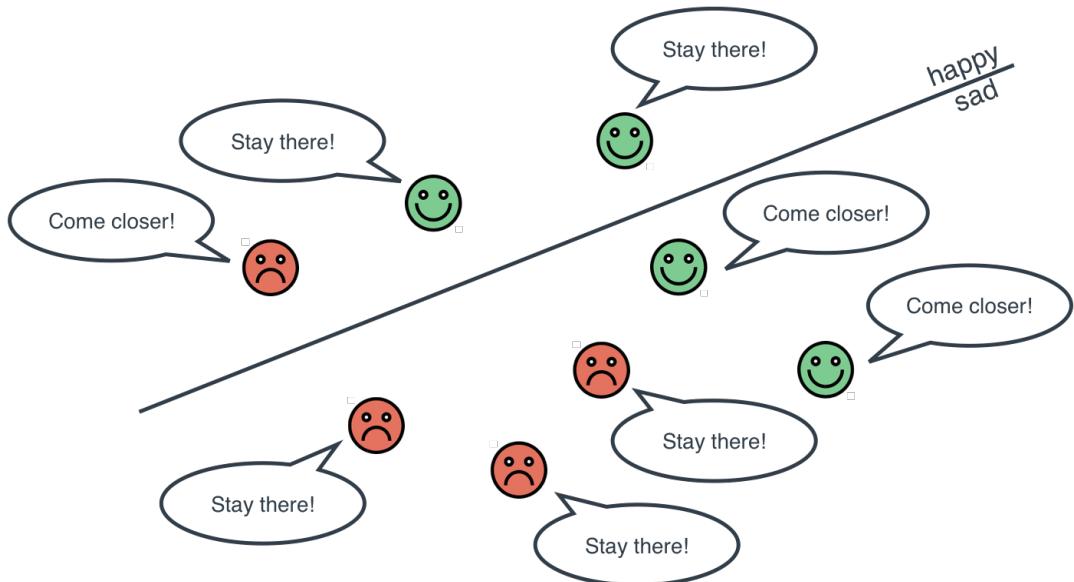
- If the point is correctly classified:
  - Output the exact same classifier.
- Else, if the point has a negative label but is misclassified as positive:
  - Output a classifier with the following scores:
    - Score of ‘aack’:  $a - \eta x$ .
    - Score of beep:  $b - \eta y$ .
    - Bias:  $c - \eta$ .
- Else, if the point has a negative label but is misclassified as positive:
  - Output a classifier with the following scores:
    - Score of ‘aack’:  $a + \eta x$ .
    - Score of beep:  $b + \eta y$ .
    - Bias:  $c + \eta$ .

## 4.4 Repeating the perceptron trick many times: The perceptron algorithm

The perceptron trick takes one point, one line, and modifies the line a little bit (or not at all), in order for that line to be a better fit for the point. Now we need to use this trick to find the

best line that will separate two classes of points. How do we do this? Simple, by repeating the perceptron trick many times. Allow me to elaborate.

Let's begin with a random line, that may not be the best fit for our points. If each point had its own way, it would tell the line what to do to be equally or better classified. Points that are correctly classified tell the line to stay where it is, and points that are misclassified tell the line to move a little closer. This is illustrated in Figure 4.23.



**Figure 4.23.** Each point tells the classifier what to do to make life better for itself. The points that are correctly classified tell the line to stay still. The points that are misclassified tell the line to move slightly towards them.

Now, the line will simply pick one of them at random, and do what the point says. If it picks a correctly classified point, then nothing changes, but if it picks an incorrectly classified one, then the line moves in order to provide a better fit for that point (Figure 4.24). It may become a worse fit for other points, but that doesn't matter.

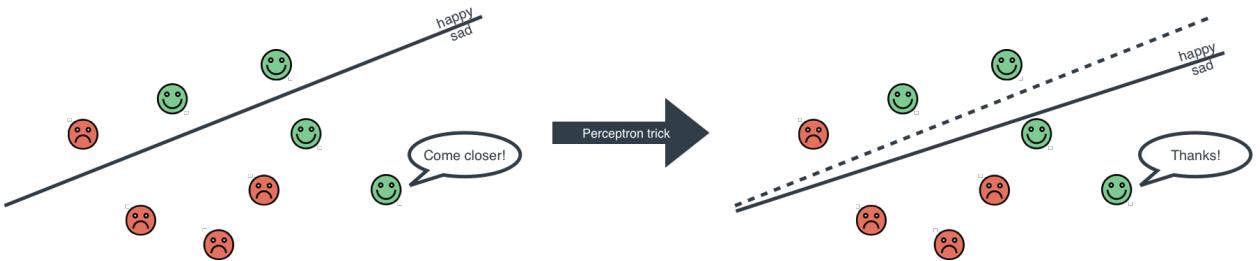


Figure 4.24. If we apply the perceptron trick to a classifier and a misclassified point, the classifier moves slightly towards the point.

It is imaginable that if we were to repeat this process many times, eventually we will get to a good solution. If you are a bit reluctant to believe this, you are right to do so, this procedure doesn't always get us to the best solution. But in a majority of the situations, it gets us to a very good solution, and this has been verified in practice. Since we can run algorithms many times and compare solutions, then an algorithm that works pretty well most of the time, will actually do the job. We call this the *perceptron algorithm*.

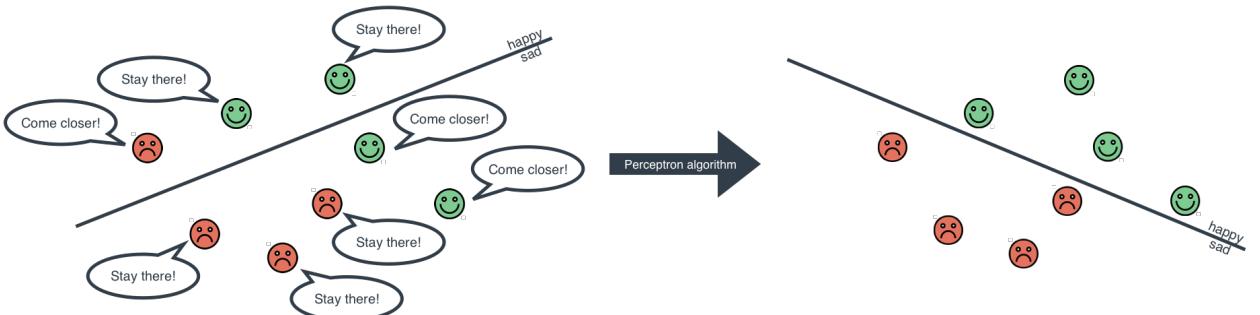


Figure 4.25. If we apply the perceptron trick many times, each time picking a random point, one can imagine that we'll obtain a classifier that classifies most points correctly.

The number of times we run the algorithm is the number of epochs. Therefore, this algorithm has two hyperparameters: The number of epochs, and the learning rate. The following is the pseudocode of the perceptron algorithm.

#### **Perceptron algorithm (pseudocode):**

Input:

- A dataset of points, where every point has a positive or negative label.
- A number of epochs,  $n$ .
- A learning rate  $\eta$ .

Procedure:

- Start with a random line. In other words, start with random values for the score of each word, and the bias.
- Repeat the following procedure n times:
  - Pick a random point.
  - Apply the perceptron trick to the point and the line. In other words, if the point is well classified, do nothing, and if it is misclassified, move the line a little bit closer to the point.
- Enjoy your well fitted line!

Since the correctly classified points want the line to stay where it is, and the incorrectly classified points want it to move closer to them, we can imagine the points yelling at the line as in the left part of Figure 4.25. The line will simply pick one of them at random, and do what this point says. This process gets repeated many times until we have a good line that separates the points well.

Now the question, how many times should we run this procedure of picking a random point and adjusting the line to fit this point better? There are several different criteria to figure out how many times to run it, such as the following:

- Run the algorithm a fixed number of times, which could be based on our computing power, or the amount of time we have.
- Run the algorithm until the error is lower than a certain threshold we set beforehand.
- Run the algorithm until the error doesn't change significantly for a certain amount of time.

Normally, if we have the computing power, it's ok to run it many more times than needed, since once we have a well-fitted line, it won't move very much. In the following section we'll code the perceptron algorithm, and analyze it by measuring the error in each step, so we'll get a better idea of when to stop running it.

## 4.5 Coding the perceptron algorithm

Ok, ready to get our hands dirty and code the perceptron algorithm? Here we go. The code is available on the public Github repo for this book, at <http://github.com/luisquiserrano/manning>. We'll test it on the following dataset.

**Table 4.3. A dataset of aliens, the times they said each of the words, and their mood.**

Aack	Beep	Happy/Sad
1	0	0

0	2	0
1	1	0
1	2	0
1	3	1
2	2	1
3	2	1
2	3	1

Let's begin by defining our dataset as a Pandas DataFrame (Table 4.3). We call X our dataset of sentences, where each sentence is encoded by the number of appearances of 'aack' and 'beep'. The set of labels is called y, where we denote the happy sentences by 1 and the sad ones by 0.

```
import pandas as pd
X = pd.DataFrame([[1,0],[0,2],[1,1],[1,2],[1,3],[2,2],[3,2],[2,3]])
y = pd.Series([0,0,0,1,1,1,1])
```

This gives us the plot on Figure 3.x. In this figure, the happy sentences are triangles and the sad ones are squares

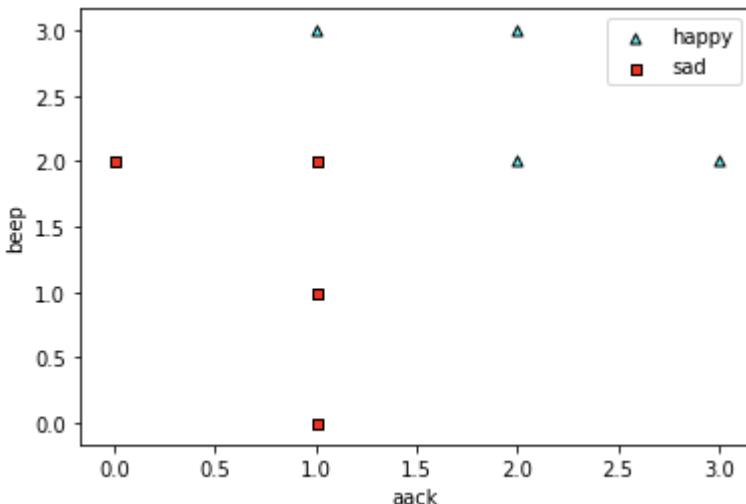


Figure 4.26. The plot of our dataset. Triangles are happy aliens, and squares are sad aliens.

#### 4.5.1 Coding the perceptron trick

In order to code the perceptron trick, let's define some notation that can be used in general, with languages of as many words as we want. Say we have a language of  $n$  words. The features are the number of times each word appears, and they are denoted as follows.

- Features:  $x_1, x_2, \dots, x_n$

The scores for the words are called the *weights*. They also include the bias, and they are denoted as follows.

- Weights:  $w_1, w_2, \dots, w_n$ .
- Bias:  $b$

The score for a particular sentence is, just as before, the sum of the number of times each word appears ( $x_i$ ) times the weight of that word ( $w_i$ ), plus the bias ( $b$ ).

$$\text{Score} = w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b = \sum_{i=1}^n w_i x_i + b.$$

This formula above is known as the *dot product*. We say that the dot product between the vectors  $(x_1, x_2, \dots, x_n)$  and  $(w_1, w_2, \dots, w_n)$  is the sum of the products of their entries, namely  $w_1 x_1 + w_2 x_2 + \dots + w_n x_n$  (if the word vector is not familiar, just think of it as an ordered set of numbers).

Finally, the prediction is simply 1 (happy) if the score is greater than or equal to zero, or 0 (sad) if the score is less than zero.

Let's begin by coding the score function and the prediction. Both receive as input, the weights of the model and the bias, and the features of one data point. The score function

returns the score that the model gives to that data point, and the prediction function returns a 1 if the score is greater than or equal to zero, and a 0 if the score is less than zero.

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

#A This function calculates the dot product between the weights and the features, and adds the bias.

```
def prediction(weights, bias, features):
    if score(weights, bias, features) >= 0:      #B
        return 1
    else:
        return 0
```

#B This function looks at the score, and if it is positive or zero, it returns 1. If it is negative, it returns 0.

It's also useful to code the error function for one point. Recall that the error is zero if the point is correctly classified, and the absolute value of the score if the point is misclassified. This function takes as input the weights and bias of the model, and the features and label of one data point.

```
def error(weights, bias, features, label):
    pred = prediction(weights, bias, features)
    if pred == label:      #C
        return 0
    else:                  #D
        return np.abs(score(weights, bias, features))
```

#C If the prediction is equal to the label, then the point is well classified, which means the error is zero.

#D If the prediction is different than the label, then the point is misclassified, which means the error is equal to the absolute value of the score.

We now write a function that adds the errors of all the points in our dataset.

```
def total_error(weights, bias, X, y):
    total_error = 0
    for i in range(len(X)):          #E
        total_error += error(weights, bias, X.loc[i], y[i])
    return total_error
```

#E We loop through our data, and for each point, we add the error at that point. We then return this error.

Now that we have the error function, we can go ahead and code the perceptron trick. Recall that the perceptron trick checks if the point is correctly classified. If it is, then it does nothing. If it is misclassified, then it adds

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

```

pred = prediction(weights, bias, features)
if pred == label:                      #F
    return weights, bias
else:
    if label==1 and pred==0:   #G
        for i in range(len(weights)):
            weights[i] += features[i]*learning_rate
            bias += learning_rate
    elif label==0 and pred==1: #H
        for i in range(len(weights)):
            weights[i] -= features[i]*learning_rate
            bias -= learning_rate
return weights, bias

```

#F If the prediction is equal to the label, the point is correctly classified and we don't need to update the model.  
#G If the prediction is negative and the label is positive, then we increase the weights and label in the model.  
#H If the prediction is positive and the label is negative, then we decrease the weights and label in the model.

There is a small shortcut we can use here. Notice that the difference between the label and the prediction (namely label-pred) is 0 when they're equal, +1 when the label is positive and the prediction is negative, and -1 when the label is negative and the prediction is positive. This helps simplify our code in the perceptron trick by using this quantity to remove the if statement as follows.

```

def perceptron_trick_clever(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

```

The functions `perceptron_trick` and `perceptron_trick_clever` do the exact same job, so it doesn't matter which one we use. However, later in this chapter when we study logistic regression, the function `perceptron_trick_clever` will be much more useful.

Now, recall that for the perceptron algorithm all we need to do is repeat the perceptron trick many times (as many as the number of epochs). For convenience, we'll also keep track of the errors at each epoch. As inputs, we not only have the data (features and labels), we also have the learning rate which we default to 0.01, and the number of epochs which we default to 200. Here is the code for the perceptron algorithm.

```

def perceptron_algorithm(X, y, learning_rate = 0.01, epochs = 200):
    weights = [1.0 for i in range(len(X.loc[0]))]      #I
    bias = 0.0
    errors = []                                         #J
    for i in range(epochs):                            #K
        errors.append(total_error(weights, bias, X, y))  #L
        j = random.randint(0, len(features)-1)           #M
        weights, bias = perceptron_trick(weights, bias, X.loc[j], y[j])  #N
    return weights, bias, errors

```

```
#I Initialize the weights to 1 and the bias to 0. Feel free to initialize them to small random numbers if you prefer.
#J An array to store the errors.
#K Repeat the process as many times as the number of epochs.
#L Calculate the error and store it.
#M Pick a random point in our dataset.
#N Apply the perceptron algorithm to update the weights and the bias of our model based on that point.
```

Now, let's run the algorithm on our dataset! Below I show you the plots of the data. I have removed the plotting lines from here, but you can find them in the repo.

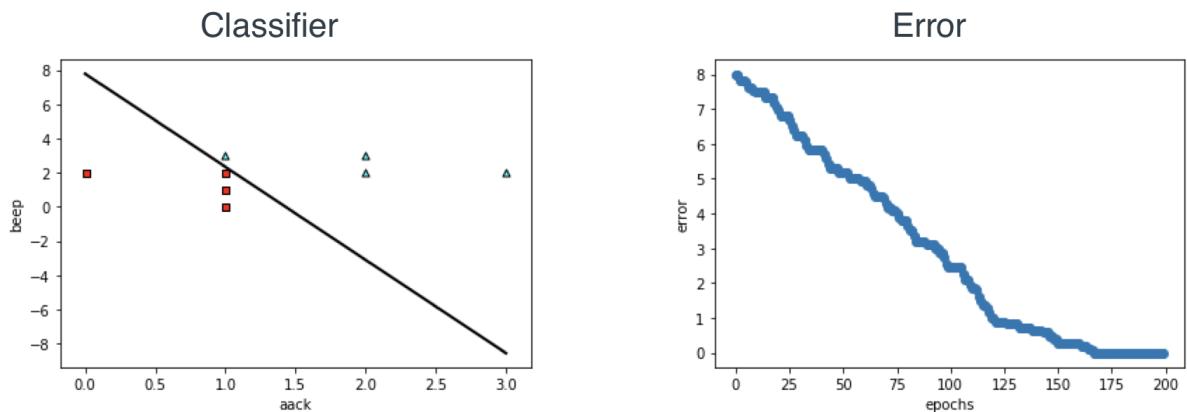
```
perceptron_algorithm(X, y)
```

The answer I got for the weights and bias was the following:

- Weight of 'aack': 0.52
- Weight of 'beep': 0.05
- Bias: -0.66

If you got a different answer than me, this is ok, since there was randomness in our choice of points inside the algorithm. In the repo I have set the random seed to zero, so feel free to look at that if you'd like to obtain the same answer as I did.

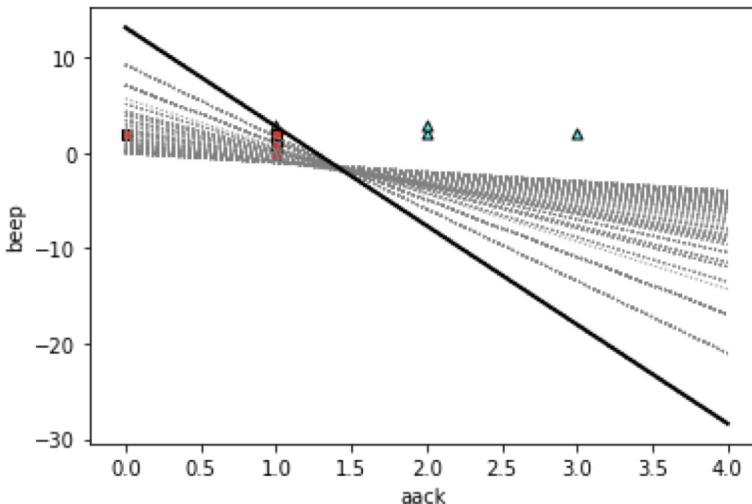
Figure 4.27 has two plots. At the left is the line fit, and at the right is the error function. Notice that the line fits the data pretty well, and the error decreases (mostly) as we increase the number of epochs.



**Figure 4.27. Left:** The plot of our resulting classifier. Notice that it classifies each point correctly.

**Right:** The error plot. Notice that the more epochs we run the perceptron algorithm, the lower the error gets.

Let's see how the classifier improved in each epoch. Figure 4.28 has a plot of each of the 200 classifiers obtained, one per epoch. Notice how we start with a bad line fit, but each iteration gets us closer and closer to separating the points well, until we get to a good classifier.



**Figure 4.28.** A plot of all the intermediate steps of the perceptron algorithm. Notice that we start with a bad classifier, and slowly move towards a good one (the thick line).

That's it, that's the perceptron classifier! In the next section we learn a slight variation of it called logistic regression, in which the predictions are not as drastic as happy/sad.

## 4.6 Applications

Like linear regression, the perceptron algorithm has many applications in real life. Basically any time we try to answer a question with yes or no, where the answer is predicted from previous data, the perceptron algorithm can help us. Here are some examples of real life applications of the perceptron algorithm.

### 4.6.1 Applications of the perceptron algorithm

#### SPAM EMAIL FILTERS

In a very similar way as we predicted if a sentence is happy or sad based on the words in the sentence, we can predict if an email is spam or not spam based on the words in the email. We can also use other features, for example,

- length of the email,
- size of attachments,

- number of senders,
- if any of our contacts is a sender (categorical variable),
- and many others.

Currently, the perceptron algorithm (and its more advanced counterparts, logistic regression and neural networks) and other classification models are used as a part of spam classification pipelines by most of the biggest email providers, with great results.

As you may imagine, any other categorization of emails can also be done with classification algorithms like the perceptron algorithm. Classifying your email into personal, subscriptions, promotions, etc., is the exact same problem. Even coming up with potential responses to an email is also a classification problem, except now the labels that we train the algorithm with, are responses to emails.

### **RECOMMENDATION SYSTEMS**

In many recommendation systems, recommending a video/movie/song/product to a user boils down to a yes/no as to answer. In these cases, the question can be any of the following:

- Will the user click on the video/movie we're recommending?
- Will the user finish the video/movie we're recommending?
- Will the user listen to the song we're recommending?
- Will the user buy the product we're recommending?

The features can be anything, going from demographic features (age, gender, location of the user), to behavioral (what videos did the user watch, what songs did they hear, what products did they buy?). You can imagine that the user vector would be a really long one! For this, large computing power and very clever implementations of the algorithms are needed.

Companies such as Netflix, YouTube, or Amazon, among many others, use the perceptron algorithm or similar more advanced classification models in their recommendation systems.

### **HEALTHCARE**

Many medical models also use classification algorithms such as the perceptron algorithm to answer questions such as the following:

- Does the patient suffer from a particular illness?
- Will a certain treatment work for a patient?

The features for these models will normally be the medical history of the patient, as well as their family medical history. You can imagine that for these types of algorithms, one needs a great deal of accuracy! It's a different thing to recommend a video that a user won't watch, than to recommend the wrong treatment for a patient.

Can you think of other applications of the perceptron algorithm?

## COMPUTER VISION

Classification algorithms such as the perceptron algorithm are widely used in computer vision, more specifically, in image recognition. Imagine if you have a picture and you want to teach the computer to tell if the picture is of a dog or not. You can use classification. The features of the model are simply the pixels of the image, and the label is precisely a variable that tells if the image is a dog or not.

The perceptron algorithm has decent performance in curated datasets such as MNIST, which is a dataset of handwritten digits. However, for more complicated images, it doesn't do very well. For these, one uses a more advanced counterpart called multi-layer perceptrons, also called neural networks. We will learn them later in the book.

Can you think of other applications of the perceptron algorithm?

## 4.7 Some drawbacks of the perceptron algorithm, which will be addressed very soon!

Perceptrons are very useful for answering questions such as yes/no, which is good when our labels are one of two categories such as happy/sad, spam/ham, or dog/no-dog. However, what if we have more categories? Say we have an image dataset with dogs, cats, and birds. What would we do? In general, the approach is to use three perceptrons, one for deciding if the image is a dog or not a dog, one for cat, and one for bird. If the dog classifier says 'yes', and the other two say 'no', then we classify the image as a dog. But what would happen if all three classifiers say 'yes'?

This hints at another problem that perceptron classifiers have, which is that their answers are too drastic, and offer very little information. For example, the sentence "I'm happy", and the sentence "I'm thrilled, this is the best day of my life!", are both happy sentences. However, the second one is much happier than the other one. A perceptron classifier will classify them both as 'happy'. How would we tweak the classifier to tell us that the second sentence is happier?

The solution to the previous two problems is *continuous logistic classifiers*, or *continuous perceptrons*, which instead of answering a question with a 'yes' or 'no', they return a score between 0 and 1. A sentence with a low score such as 0.1 gets classified as sad, and a sentence with a high score such as 0.9 gets classified as happy. For the dog/cat/bird problem, three continuous perceptrons will return three different scores, and the prediction will be given by the classifier that gives the image the highest score. We'll learn continuous logistic classifiers later in this book.

## 4.8 Summary

- Classification is a very important part of machine learning. It is similar to regression in that it consists of training an algorithm with labelled data, and using it to make predictions on future (unlabelled) data, except this time the predictions are categories, such as yes/no, spam/ham, etc.

- Logistic classifiers work by assigning a weight to each of the features, and a threshold. The score of a data point is then calculated as the sum of products of weights times features. If the score greater than or equal the threshold, the classifier returns a 'yes'. Otherwise, it returns a 'no'.
- One can tweak the classifier so that the threshold is always 0. For this, one introduces a bias, which is a number added to the score. The bias is equal to the negative of the previous threshold.
- For sentiment analysis, a logistic classifier consists of a score for each of the words in the dictionary, together with a threshold. Happy words will normally end up with a positive score, and sad words with a negative score. Neutral words like 'the' will likely end up with a score close to zero.
- The bias helps us decide if the empty sentence is happy or sad. If the bias is positive, then the empty sentence is happy, and if it is negative, then the empty sentence is sad.
- Graphically, we can see a logistic classifier as line trying to separate two classes of points, which can be seen as points of two different colors.
- In higher dimensions, a logistic classifier will be a high dimensional plane separating points. This is hard to imagine for dimensions higher than 3, but in 3 dimensions, one can imagine points of two colors flying in space, and a plane separating the points.
- The way the perceptron algorithm works is by starting with a random line, and then slowly moving it to separate the points well. In every iteration it picks a random point. If the point is correctly classified, the line doesn't move. If it is misclassified, then the line moves a little bit closer to the point, in an attempt to pass over it and classify it correctly.
- The perceptron algorithm has numerous applications, including spam email detection, recommendation systems, e-commerce, and healthcare.

# 5

## *A continuous approach to splitting points: Logistic regression*

### This chapter covers

- The difference between hard assignments and soft assignments.
- Activation functions such as the step function vs the sigmoid function.
- Discrete perceptrons vs continuous perceptrons.
- The logistic regression algorithm for classifying data.
- Coding the logistic regression algorithm in Python.
- Using the softmax function to build classifiers for more than two classes.

In the previous chapter, we built a classifier that determined if a sentence was happy or sad. But as you can imagine, there are sentences that 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 one is much happier than the first one. Wouldn’t it be nice to have a classifier that not only predicts if sentences are happy or sad, but that actually gives us 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 section we will define the *logistic regression 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 is assigned.

In a nutshell, logistic regression is a type of model which works just like a perceptron, except instead of returning a yes/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 chapter relies on chapter 4, as the algorithms we develop here are very similar, except for some technical differences. I recommend you make sure you understand chapter 4

well before you read this chapter. In chapter four we described the perceptron algorithm by the means of an error function which tells us how good a perceptron classifier is, and an iterative step which moves us from a classifier to a slightly better classifier. In this chapter we learn the logistic regression algorithm which works in a similar way. The only difference is that the new error function changes (it is now based on a probability), and the new iterative step also changes slightly to fit this new error function.

## 5.1 Logistic Regression (or continuous perceptrons)

In chapter 4, 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*, since it returns an answer from a discrete set. In this chapter we learn *continuous perceptrons*, which are called this because they return an answer that can be any number in the interval between 0 and 1. This answer can be interpreted as a probability in the sense that sentences with a higher score are more likely to be happy sentences, and viceversa. The way I visualize continuous perceptrons is similar to how I visualize 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, while the continuous perceptron assigns a value from 0 to 1 to all the points based on their position. 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, the sentence "Today is Tuesday" is neither happy or sad, so the model would assign it a score close to 0.5. Points in the positive region get scores larger than 0.5, where the points even further away from the 0.5 line in the positive direction get values closer to 1. Points in the negative region get scores smaller than 0.5, where again the farther points from the line get values closer to 0. No point gets a value of 1 or 0 (unless we consider points at infinity).

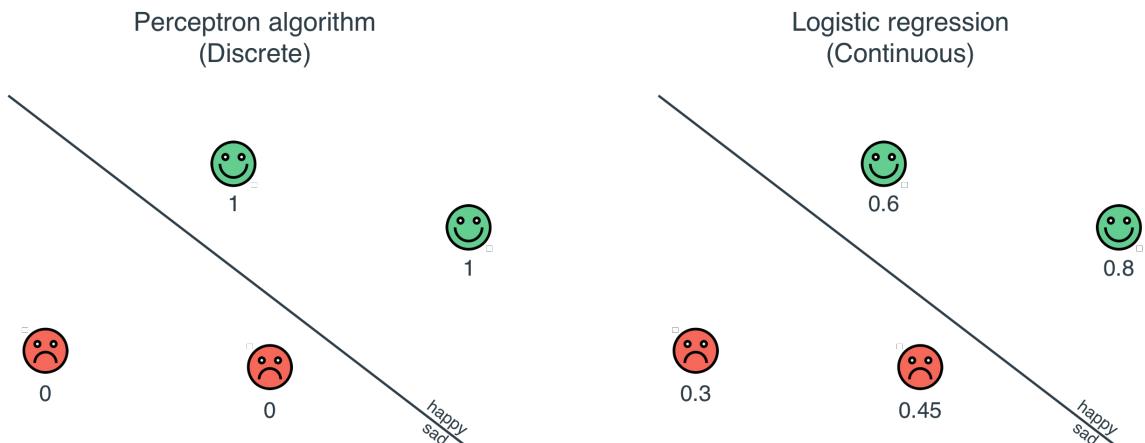


Figure 5.1. Left: The perceptron algorithm trains a discrete perceptron, where the predictions are 0 (happy) and

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.

### 5.1.1 A probability approach to classification - The sigmoid function

The question now is, how do we slightly modify the perceptron models from the previous section in order to get a score for each sentence, as opposed to a simple 'happy' or 'sad'? Let's 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 a function on the score. The function returns a 1 if the score was positive, and a 0 if it was negative. This function is called the *step function*.

Now we'll do something similar. We'll 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 5.2.

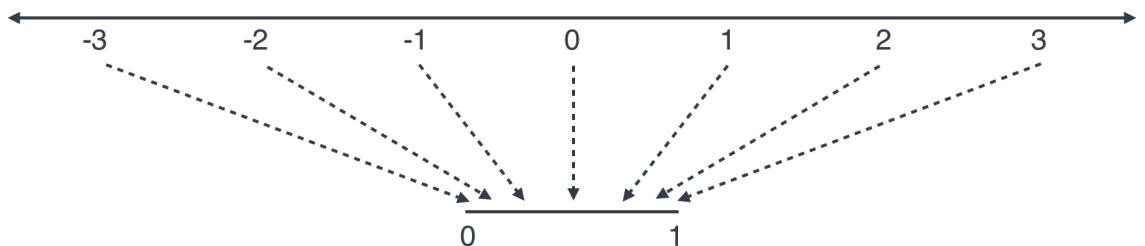
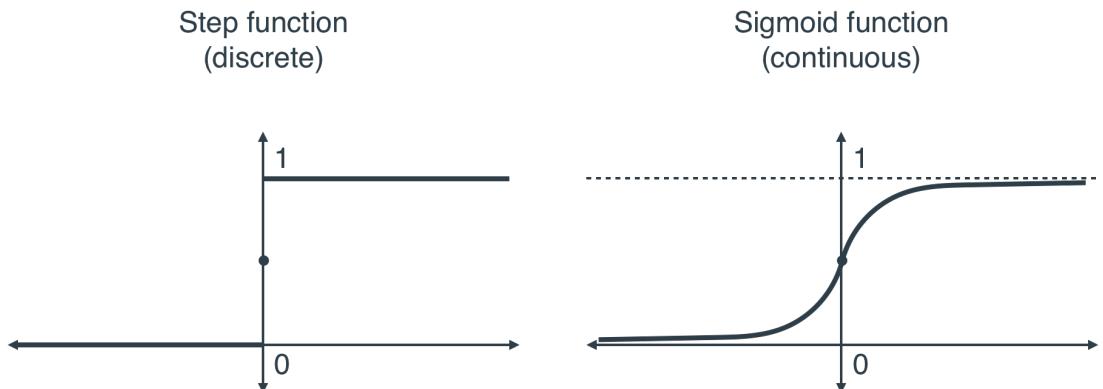


Figure 5.2. INSERT DESCRIPTION

Many functions can help us here, and in this case, we'll use one called the *sigmoid*, denoted with the greek letter  $\sigma$ . The formula for the sigmoid is the following:

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

Don't worry much about this formula. What really matters is what the function does, which is crunch all the real number line into the interval  $(0,1)$ . In Figure 5.3 we can see a comparison of the graphs of the step and the sigmoid functions.



**Figure 5.3.** 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. One reason is that having continuous predictions gives us more information than discrete predictions. Another is that when we do calculus, the sigmoid function has a much nicer derivative than the step function, whose derivative is always zero except for the origin, where it is undefined.

The code for the sigmoid function in Python is very simple; we make use of the numpy function `exp`, which takes any real number as an input, and returns e to the power of that number.

```
import numpy as np
def sigmoid(x):
    return 1/(1+np.exp(-x))
```

In Table 5.1 we use the previous code to calculate some values of the sigmoid function, to make sure the function does what we want it to.

**Table 5.1.** Some inputs and their outputs under the sigmoid function. Notice that large negative values have an output close to 0, and large positive values have an output close to 1. The value 0 has an output of 0.5.

x	$\sigma(x)$
-5	0.0067
-1	0.269

0	0.5
1	0.731
5	0.9933

Now we are ready to define a prediction. The prediction is obtained by applying the sigmoid function to the score, and it returns a number between 0 and 1 which, as I mentioned before, can be interpreted in our example as the probability that the sentence is happy. Here's the code for the prediction function (where the `score` formula is the same as the one in Chapter 4).

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

In the previous chapter we defined an error function for a prediction, and we used it to build a slightly better 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 great similarities.

### 5.1.2 The error functions - Absolute, square, and log loss

In this section we cook up some error functions for a continuous perceptron classifier. But first let's stop and think, what properties would we like a good error function to have? Here are some I can think of:

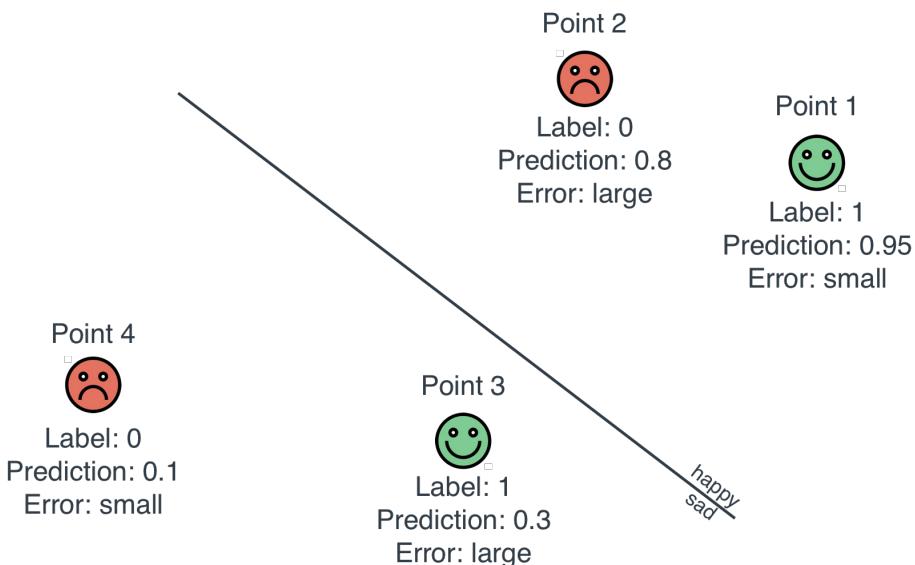
- If a point is correctly classified, the error is a small number.
- If a point is incorrectly classified, the error is a large number.
- The error of a classifier at a set of points is the sum of errors at each point.

Many functions satisfy these properties, and I will show you three of them; the absolute loss, the square loss, and the log loss. Let's look at the scenario in Table 5.2 and Figure 5.4. This dataset consists of two happy points and two sad points. For convenience, we'll assign a label of 1 to the happy points, and of 0 to the sad points. The point of the classifier is to assign predictions for these points that are as close as possible to the labels. We also have a continuous perceptron classifier represented by the line. The classifier then makes a prediction between 0 and 1 for each point in the plane, as follows:

- The points on the line are given a prediction of 0.5.
- Points that are up and to the right of the line 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 down and to the left of the line 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 5.2.** Four points, two happy and two sad with their predictions, as illustrated in Figure 5.4. Notice that points 1 and 4 are correctly classified, while 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	True label	Predicted label	Error?
1	1 (Happy)	0.95	Should be small
2	0 (Sad)	0.8	Should be large
3	1 (Happy)	0.3	Should be large
4	0 (Sad)	0.1	Should be small



**Figure 5.4.** The plot of Table 5.2. Notice that points 1 and 4 are correctly classified while points 2 and 3 are misclassified. We expect points 1 and 4 to have a small error, and points 2 and 3 to have a large error.

Notice that in Table 5.2, points 1 and 4 get a prediction that is close to the label, so they should have small errors. In contrast, points 2 and 3 get a prediction that is far from the label, so they should have large errors. Here are the three error functions.

### **ERROR FUNCTION 1: ABSOLUTE ERROR**

The absolute error is very similar to the absolute error we defined for linear regression in Chapter 3. It is the absolute value of the difference of 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 5.3. Notice that points 1 and 4 (correctly classified) have small errors, while points 2 and 3 (incorrectly classified) have larger errors.

**Table 5.3.** We have attached the absolute error and the square error for the points in Table 5.2. Notice that as we desired, points 1 and 4 have small errors while points 2 and 3 have larger errors.

Point	True label	Predicted label	Absolute Error	Square Error
1	1 (Happy)	0.95	0.05	0.0025
2	0 (Sad)	0.8	0.8	0.64
3	1 (Happy)	0.3	0.7	0.49
4	0 (Sad)	0.1	0.1	0.01

The absolute and the square loss 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 I'll show you. Why is it more popular? One reason is that the math (derivatives) work much nicer with the next function. Another one is that if you notice, these errors are all pretty small. In fact, they are all smaller than 1, no matter how poorly classified the point is. I'll elaborate more in the next section but for now I present to you (drumroll...) the log loss.

### **ERROR FUNCTION 3: LOG LOSS**

Notice that with the absolute and square error functions, points that are vastly misclassified have large errors, but never too large. Let's look at an example: a point with label 1 but that the classifier has assigned a prediction of 0.01. This point is vastly misclassified, since we would hope that the classifier assigns it a prediction close to 1. The absolute error for this point is the difference between 1 and 0.01, or 0.99. The square error is this difference

squared, or 0.9801. But this is a small error for a point that is so vastly misclassified. We'd like an error function that gives us a higher error for this point.

For this purpose, let's look at probabilities. The classifier assigns a probability to every data point, and that is the probability that the point is happy. Let's instead consider the probability that the point is its label. Thus, for the points with label 1 (happy points), we look at the probability that the point is happy (the label), and for the points with label 0, we look at the probability that the point is sad (the opposite label). The prediction that a point is sad is simply 1 minus the prediction that the point is happy. Now, note that a point that is well classified is assigned a high probability of being its label (whether happy or sad), and a point that is poorly classified is assigned a low probability of being its label. If that sounds a bit confusing, let's look at our four points.

- Point 1:
  - Label = 1 (happy)
  - Prediction = 0.95
  - Probability of being happy: 0.95
- Point 2:
  - Label = 1 (sad)
  - Prediction = 0.8
  - Probability of being sad:  $1 - 0.8 = 0.2$
- Point 3:
  - Label = 1 (happy)
  - Prediction = 0.3
  - Probability of being happy: 0.3
- Point 4:
  - Label = 1 (sad)
  - Prediction = 0.1
  - Probability of being happy:  $1 - 0.1 = 0.9$

As we can see from these points, a point being correctly classified is equivalent to a point that has been assigned a high probability of being its own label. All we need to do now is to turn this probability into an error. Ideally, we'd like a function that gives us very high values for points that are close to 0, and very low values for points that are close to 1. We'd like a function where the graph looks like Figure 5.5.

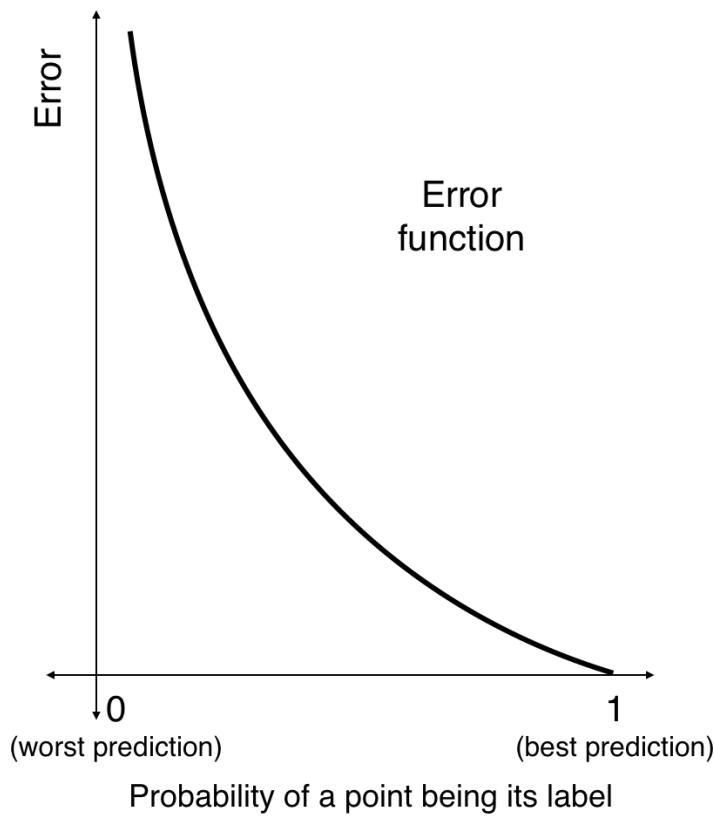


Figure 5.5. In the horizontal axis we see the probability of a point being predicted its label (whether it's happy or sad). In the vertical axis we see the error. Notice that well classified points lie towards the right, as their probability that they are their label is high, whereas poorly classified points lie towards the left. We would like an error function that looks like this graph, namely, that assigns high values to the poorly classified points in the left, and low values to the correctly classified points in the right.

All we need to do is find a formula for that function. Can you help me think of a function that may work? Here is an idea, let's look at the graph of the natural logarithm function, in the interval between 0 and 1 in Figure 5.6.

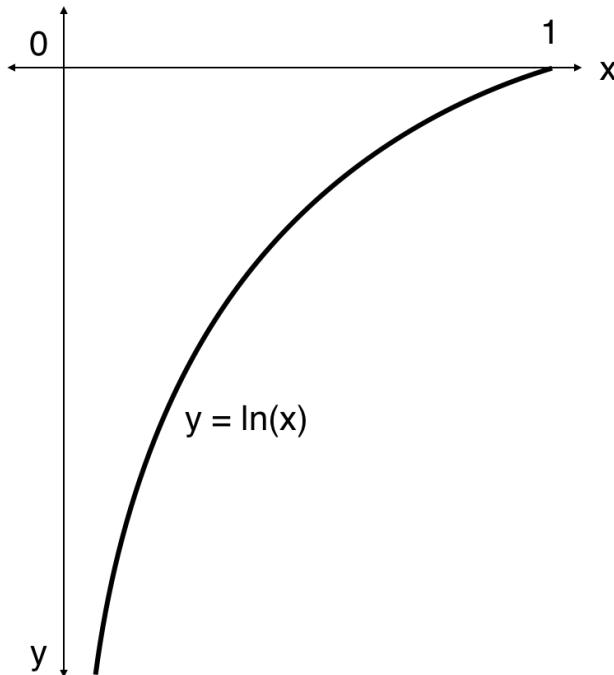


Figure 5.6. Quite conveniently, the plot of natural logarithm looks a lot like the one we desire in Figure 5.5, only inverted!

They look familiar, don't they? All we need to do is flip the function by multiplying it by -1. Thus, we obtain our log loss error function. It is precisely the negative natural logarithm of the probability of the point being its label.

$$\text{log loss} = -\ln(\text{Probability of a point being its label}).$$

Now let's calculate it for our four data points.

Table 5.4. We add the log loss to our table of values. Notice that we now allow larger values for the error, and as you can imagine, the more poorly classified the point, the higher the log loss.

Point	True label	Predicted label	Probability of being its label	log loss
1	1 (Happy)	0.95	0.95	$-\ln(0.95) = 0.051$
2	0 (Sad)	0.8	0.2	$-\ln(0.2) = 1.386$

3	1 (Happy)	0.3	0.3	$-\ln(0.4) = 0.916$
4	0 (Sad)	0.1	0.9	$-\ln(0.9) = 0.105$

We can write the log loss as following:

- If the label is 0:
  - $\text{log loss} = -\ln(1 - \text{prediction})$
- If the label is 1:
  - $\text{log loss} = -\ln(\text{prediction})$ .

As a matter of fact, we can make the formula even nicer. Check this out:

$$\text{log loss} = -\text{label} \times \ln(\text{prediction}) - (1 - \text{label}) \times \ln(1 - \text{prediction})$$

Notice that this works, since if the label is 0, only the second summand survives, and if the label is 1, only the first summand survives, thus giving us exactly what we want.

### COMPARING THE ERROR FUNCTIONS

In table 5.5 we continue our work and add the log loss column using the natural logarithm formula from the previous section. Now we have our three desired error functions.

**Table 5.5. The full table with the four points, their label, predicted label, absolute error, square error and log loss.**

Point	True label	Predicted label	Absolute Error	Square Error	Log loss
1	1 (Happy)	0.95	0.05	0.0025	0.051
2	0 (Sad)	0.8	0.8	0.64	1.609
3	1 (Happy)	0.3	0.7	0.49	0.916
4	0 (Sad)	0.1	0.1	0.01	0.105

If I haven't convinced you of the power of the log loss error function, let's look at an extreme point. Let's say we have a point with label 1 (happy), for which the classifier makes a prediction of 0.00001. This point is very poorly classified. The absolute error will be 0.99999, and the square error will be 0.9999800001. However, the log loss will be the negative of the

natural logarithm of  $(1-0.99999)$ , which is 11.51. This value is much larger than the absolute or square errors, which means the log loss error is a better alarm for poorly classified points.

### 5.1.3 More on the log loss error function

In section 5.1.1, I made a case for why I prefer the log loss error function over the absolute and square error functions. In this section I give you another reason, a reason that has to do with independent probabilities. We've calculated the log loss for one point. The total log loss for a classifier is defined as the sum of the log loss for every point in the dataset. But now we have a sum of logarithms; does this ring a bell? Whenever I see a sum of logarithms, the first thing I think of is the logarithm of a product. And whenever I see a product, I check if that product may have come from multiplying probabilities. Why do we multiply probabilities? Because when events are independent (or when we assume they are, for the sake of simplicity) their probabilities get multiplied. In Chapter 6 we delve much deeper into independent events and their probabilities, but for now, we can use the fact that if the occurrences of two events don't depend on each other, the probability of both of them happening is the product of the probabilities of both events happening.

What are the events we are talking about here? Let's say we have the dataset in Figure 5.7, with two classifiers. The classifier on the left is bad, as it misclassifies two of the four points, and the one on the right is good, as it classifies every point correctly.

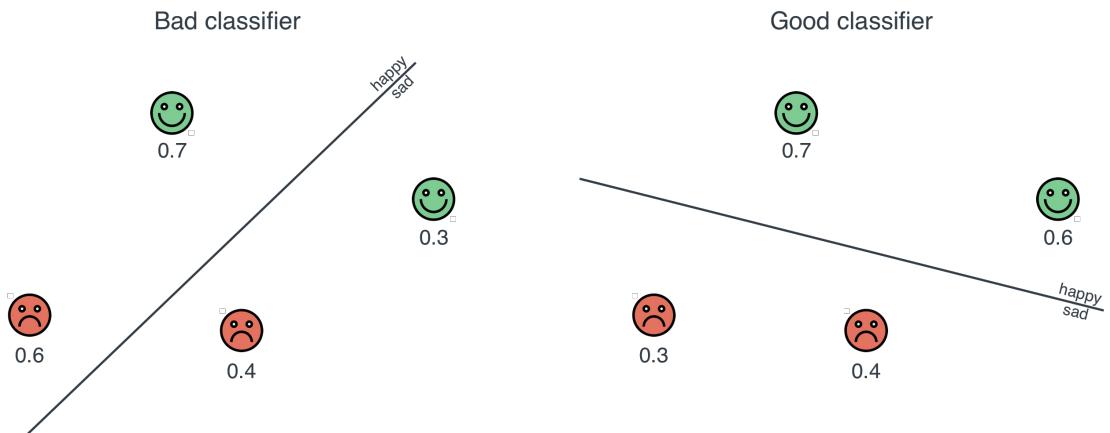


Figure 5.7. Left: A bad classifier that classifies 2 points correctly and 2 incorrectly. Right: A good classifier that classifies every point correctly.

Now, before we calculate the log loss, let's think of probabilities. In Figure 5.7, we have specified the probability assigned to each of the points. Recall that this is the probability that the point is happy. However, as in the previous section, we care about the probability that our classifier assigns to the actual label. Namely, if the point is happy, we care about the

probability that the point is happy (the prediction), and if it is sad, we care about the probability that the point is sad (one minus the prediction). Next, we consider all the points at the same time, and we are interested in the probability that all the points are their label *at the same time*. We don't really know what this probability is, since the points may influence each other, but if we assume that they don't, if we assume that the points are independent from each other, then the probability that all the points are their label is precisely the *product* of all these probabilities. We would imagine that this probability is high for a good classifier and low for a bad classifier. In Figure 5.8, we can see the calculations of this probability. Notice that the bad model assigns the labels of this dataset a total probability of 0.0504, while the good model assigns it a probability of 0.1764, which is higher.

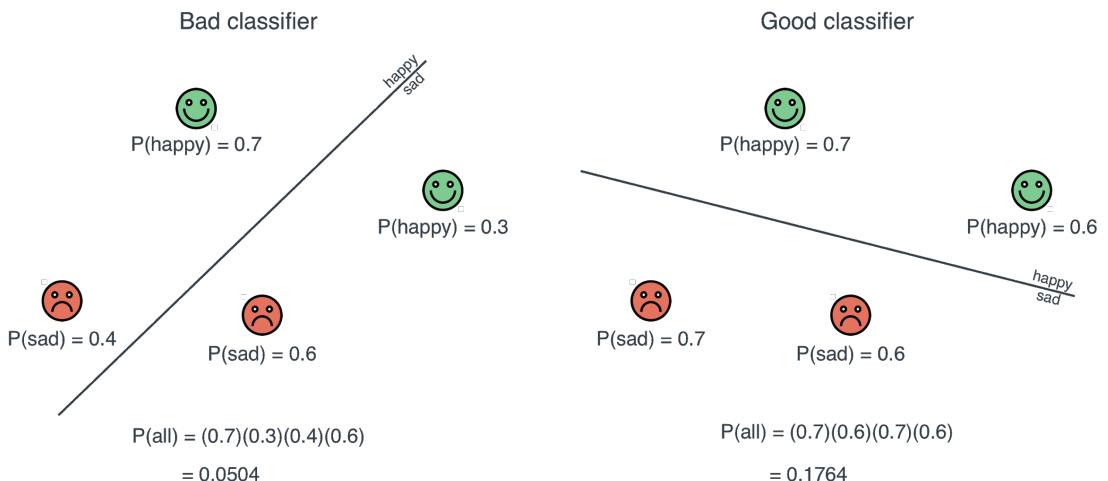


Figure 5.8. The same classifiers as in Figure 5.7. Except for the sad ones, we have calculated the predicted probability of being sad as 1 minus the predicted probability of being happy. We have assumed that these points are all independent from each other, and multiplied the probabilities to obtain the total probability that the classifier has predicted for the labels of the points.

Here is a problem. Products are hard. Let me explain why. Imagine if we had a dataset of 1 million points. Each one is assigned a probability between 0 and 1. The product of 1 million numbers between 0 and 1 would be tiny. Furthermore, it is volatile, as changing one of the factors could really change this product. In general, big products are something we don't want to handle. What do we like? We like sums, they are much easier to handle. And how do we turn products into sums? We have a very useful function called the logarithm, since the logarithm of a product is precisely the sum of the logarithms of the factors.

To summarize, these are the steps for calculating the log loss:

- For each point, we calculate the probability that the classifier predicts for its label (happy or sad).

- 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.
- Since the logarithm of a product is the sum of the logarithms of the factors, we obtain a sum of logarithms, one for each point.
- We notice that all the terms are negative, since the logarithm of a number less than 1 is a negative number. Thus, we multiply everything by minus 1 to get a sum of positive numbers.
- This sum is our log loss.

Notice that the classifier in the left, which is bad, has a log loss of 2.988. The classifier in the right, which is good, has a smaller log loss of 1.735. Thus, the log loss does its job, which is to assign a large error value to bad classifiers and a smaller one to good classifiers.

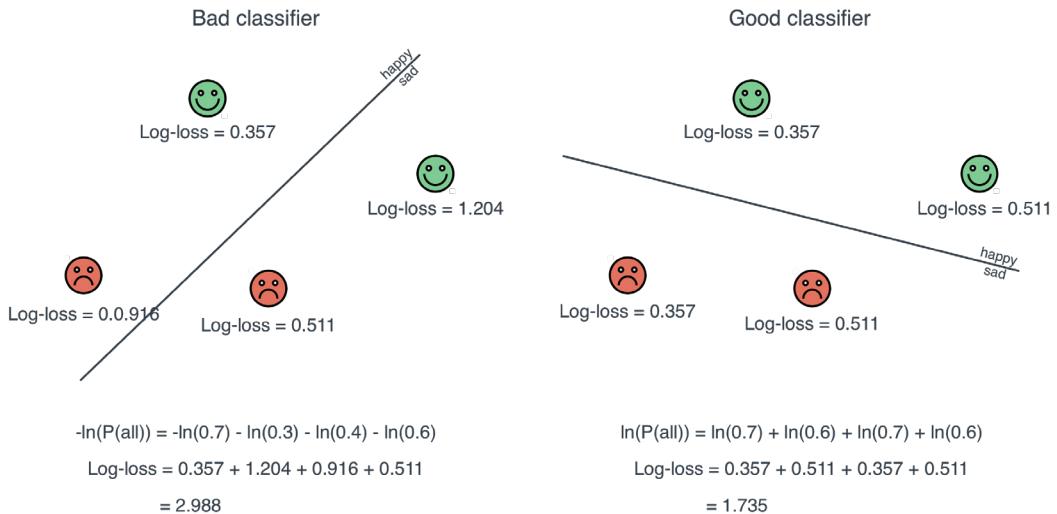


Figure 5.9. We now calculate the log loss by taking the probability calculated in Figure 5.8. Notice that the good classifier (right) has a smaller log loss than the bad classifier (left)

## 5.2 Reducing the log loss error: The logistic regression trick

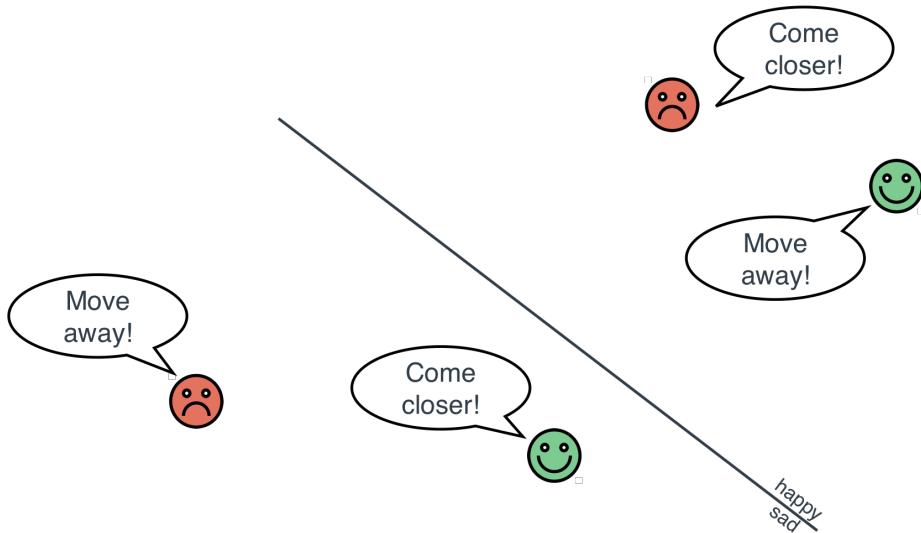
In Chapters 4 we defined an error function for our classifier, and then an algorithm which builds a classifier by minimizing that error function. The algorithm worked in an iterative way, namely, it started with a random classifier and slowly changed the weights step by step until the error was small. The classifier we obtained was a discrete perceptron. With continuous perceptrons, we will do the exact same thing. In this section I show you the logistic regression algorithm, which does precisely that: It builds a continuous perceptron by reducing the log loss of a classifier.

Recall from Chapter 4 that the perceptron algorithm consisted in starting with a random classifier and successively picking a random point and applying the perceptron trick. The perceptron trick had the following steps:

1. If the point was correctly classified, it would not move the line.
2. If the point was incorrectly classified, it would move the line slightly towards the point.

The logistic regression algorithm is very similar to the perceptron algorithm. The only thing that changes is that we use the logistic regression trick instead of the perceptron trick. The logistic regression trick has the following steps:

1. If the point is correctly classified, it moves the line slightly away from the point.
2. If the point is incorrectly classified, it moves the line slightly towards the point.



**Figure 5.10.** In the logistic regression algorithm, every point has a say. Points that are correctly classified tell the line to move farther away, in order to be deeper in the correct area. Points that are incorrectly classified tell the line to come closer, in hopes of one day being on the correct side of the line.

Does this make sense? Think about this: if a point is well classified, it means it is on the correct side of the line. How would this point be better classified? Well, if the line were to move farther *away* from the point, this would put the point deeper inside the correct side of the line. Let's illustrate this with an example.

### 5.2.1 An example with a discrete perceptron and a continuous perceptron

In order to really understand the difference between a discrete and a continuous perceptron, we use the same example as in section 4.3.1. Recall that in this section we started with a

classifier which predicts whether a sentence is happy or sad by assigning scores to the words. Then we used the perceptron algorithm to slightly improve this classifier.

The features, or words, were ‘aack’ and ‘beep’. The classifier and sentence are the following:

- Classifier (scores):
  - ‘Aack’: 1 pt.
  - ‘Beep’: 1 pts.
  - Bias: -4 pts.
- Sentence 1:
  - Words: ‘Aack beep beep beep aack beep beep!’
  - Label: Sad

The scores can define a discrete perceptron classifier or a continuous perceptron classifier, depending on if we use the step function or the sigmoid as the activation function. What we do next is we will improve the discrete perceptron classifier with the perceptron algorithm to get a slightly better classifier. Then I will show you a new technique, the *logistic regression algorithm*, which will improve the continuous perceptron classifier, in a very similar way.

### **Using the perceptron algorithm to improve our discrete perceptron classifier**

First let’s look at what the step that the perceptron algorithm would take in order to improve this classifier. This algorithm would calculate the score of the sentence, which is the sum of the scores of the words times the number of times each word appears in the sentence, plus the bias:

$$\begin{aligned}
 \text{Score} &= (\text{score of aack}) * \# \text{aack} + (\text{score of beep}) * \# \text{beep} - 4 \\
 &= 1 * 2 + 1 * 4 - 4 \\
 &= 2
 \end{aligned}$$

Since the score is positive, then the discrete perceptron classifier classifies the sentence as happy (which means it misclassifies it). In this example, we decided that we would update the scores by subtracting from them the error rate times the number of times each word appears in the sentence. If the error rate was 0.01, we did the following:

- Update the score of ‘aack’ by subtracting  $2 * 0.01$ , thus obtaining 0.98.
- Update the score of ‘beep’ by subtracting  $4 * 0.01$ , thus obtaining 0.96.
- Update the bias by subtracting 0.01, thus obtaining -4.01.

Why did we subtract from the scores? Recall that in the perceptron algorithm, if the label was 0 and the prediction 1, we would subtract from the scores, and if the label was 1 and the prediction 0, we would add to the scores. We won’t have to make this decision in the logistic regression algorithm; you’ll soon see why.

The new classifier would assign a score of  $0.98*2 + 0.96*4 - 4.01 = 1.79$ . This is better than the previous score of 2, because the sentence is sad, so in order to classify it properly, the classifier would need to give it a negative score.

### **Using the logistic regression algorithm to improve our discrete perceptron classifier**

What would happen with a continuous perceptron classifier? This one would apply the sigmoid function to the score, and use that as a prediction. The sigmoid function gives us

$$\text{Prediction} = \sigma(2) = 0.881.$$

Now please bear with me, I'll show you the example of the logistic regression algorithm without showing you the algorithm yet. This one will appear in Section 5.3.3 (feel free to skip the pages and read it if you need the definition before the example).

The difference between the label and the prediction is  $0 - 0.881 = -0.881$ . We'll update the scores just as before, except we will scale everything by this number, -0.881 and we will always add to the scores.

- Update the score of 'aack' by adding  $2*0.01*(-0.881)$ , thus obtaining 0.982.
- Update the score of 'beep' by adding  $4*0.01*(-0.881)$ , thus obtaining 0.965.
- Update the bias by adding  $0.01*(-0.881)$ , thus obtaining -4.009.

The new classifier will give the point the following score and prediction.

$$\text{Score} = 0.982*2 + 0.965*4 - 4.009 = 1.815$$

$$\text{Prediction} = \sigma(1.815) = 0.86$$

Notice that this prediction is smaller than the previous one, so the classifier improved again.

### **5.2.2 A second example with a discrete perceptron and a continuous perceptron**

One of the main benefits of the logistic regression algorithm over the perceptron algorithm is that if a point is correctly classified, the perceptron algorithm leaves it alone and doesn't improve, while the logistic regression algorithm still uses the information of that point to improve the model. In order to see this, let's look at a slightly different example. The scores are the same as the previous example, but now, our original point is correctly classified, so it's classified as sad. Notice that the perceptron algorithm would do absolutely nothing, since the point is correctly classified. However, if we use the logistic regression algorithm and follow the same approach of updating the weights as with the previous example, we see that the difference between the label and the prediction is now  $1 - 0.881 = 0.119$  (since the label is 1). Let's update the scores in the exact same way as before.

- Update the score of 'aack' by adding  $2*0.01*(0.119)$ , thus obtaining 1.002.
- Update the score of 'beep' by adding  $4*0.01*(0.119)$ , thus obtaining 1.005.
- Update the bias by adding  $0.01*(0.119)$ , thus obtaining -3.999.

Notice that since the difference between the label and the prediction is negative, we subtracted negative quantities, which is the same as adding positive quantities. The score and prediction that this new classifier gives to our point is:

$$\text{Score} = 1.002*2 + 1.005*4 - 3.999 = 2.025$$

$$\text{Prediction} = \sigma(2.025) = 0.883$$

Notice that the prediction went up from 0.881 to 0.883! This is better, even if only by a small amount, because it is closer to the label, which is 1.

### 5.2.3 Moving the line to fit the points - The logistic regression algorithm

Notice that in the previous two examples, we always added a small amount to the score. The amount we added could be positive or negative based on the difference between the label and the prediction, but this didn't matter. We are ready to formally define the logistic regression algorithm for our example.

#### **Logistic regression trick (pseudocode):**

Input:

- A classifier with the following scores:
  - Score of 'aack': a.
  - Score of 'beep': b.
  - Bias: c.
- A point with coordinates  $(x_1, x_2)$  (where  $x_1$  is the number of appearances of the word 'aack', and  $x_2$  of the word 'beep').
- A learning rate  $\eta$ .

Procedure :

- Calculate the prediction that the classifier gives to the datapoint as:
  - $\hat{y} = \sigma(ax_1 + bx_2 + c)$
- Output a classifier with the following scores:
  - Score of 'aack':  $a + \eta(y - \hat{y})x_1$
  - Score of beep:  $b + \eta(y - \hat{y})x_2$
  - Bias:  $c + \eta$

And now that we have the logistic regression trick, we can easily write the pseudocode for the logistic regression algorithm.

#### **Logistic regression algorithm (pseudocode):**

Input:

- A dataset of points, where every point has a positive or negative label.

- A number of epochs, n.
- A learning rate  $\eta$ .

Procedure:

- Start with a random line. In other words, start with random values for the score of each word, and the bias.
- Repeat the following procedure n times:
  - Pick a random point.
  - Apply the logistic regression trick to the point and the line. In other words, if the point is well classified, move the line a little farther from the point, and if it is misclassified, move the line a little closer to the point.
- Enjoy your well fitted line!

Notice something very special. If we were to use the logistic regression trick on a classifier that only outputs the predictions 0 and 1, we would get the perceptron trick in Chapter 4. I encourage you to verify this as an exercise, by running the logistic regression trick but only with the values  $\hat{y} = 0$  and  $\hat{y} = 1$ .

### 5.2.4 Coding the logistic regression algorithm

We use the same notation as in Chapter 4. The label of each sentence is the sentiment (happy or sad). We have a language with n words. The features are the number of times each word appears, and the weights are the scores corresponding to the words. These weights include the bias, which corresponds to no word, it simply gets added to the score of every sentence.

- Features:  $x_1, x_2, \dots, x_n$
- Label:  $y$
- Weights:  $w_1, w_2, \dots, w_n$
- 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 ( $b$ ) (which we called the dot product).

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

To follow along with the code, take a look at this book's repo at <http://www.github.com/luisquiserrano/manning>. We start by coding the sigmoid function, the score, and the prediction.

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

def lr_prediction(weights, bias, features):
```

```
    return sigmoid(score(weights, bias, features))
```

Now that we have the prediction, we can proceed to the log loss. We need to do a bit of math to figure out the formula for the log loss. Recall that the log loss is the natural logarithm of the probability that a point is given its own label by the classifier. To be more clear, if a point has label 1, then the log loss is the natural logarithm of the probability that the classifier has given to that data point. If the point has label 0, then the log loss is the natural logarithm of one minus that same probability. This is because the classifier outputs the probability that the point has label 1. Since we called the label  $y$  and the prediction  $\hat{y}$ , we get the following:

- If the label is  $y = 1$ , then  $\text{log loss} = \ln(\hat{y})$
- If the label is  $y = 0$ , then  $\text{log loss} = \ln(1 - \hat{y})$

Note that we can encode these two easily into one formula:

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

Let's code that formula.

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

We need the log loss over the whole dataset, so we can add over all the data points.

```
def total_log_loss(weights, bias, X, y):
    total_error = 0
    for i in range(len(X)):
        total_error += log_loss(weights, bias, X.loc[i], y[i])
    return total_error
```

Now we are ready to code the logistic regression trick, and the logistic regression algorithm.

```
def lr_trick(weights, bias, features, label, learning_rate = 0.01):
    pred = lr_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 lr_algorithm(features, labels, learning_rate = 0.01, epochs = 200):
    weights = [1.0 for i in range(len(features.loc[0]))]
    bias = 0.0
    errors = []
    for i in range(epochs):
        draw_line(weights[0], weights[1], bias, color='grey', linewidth=1.0,
                  linestyle='dotted')
        errors.append(total_error(weights, bias, features, labels))
        j = random.randint(0, len(features)-1)
        weights, bias = perceptron_trick(weights, bias, features.loc[j], labels[j])
    draw_line(weights[0], weights[1], bias)
```

```

plot_points(features, labels)
plt.show()
plt.scatter(range(epochs), errors)
plt.xlabel('epochs')
plt.ylabel('error')
return weights, bias

```

We'll test our code in the same dataset that we used in Chapter 4. The code for loading our small dataset is below, and the plot of the dataset is in Figure 5.11.

```

import pandas as pd
X = pd.DataFrame([[1,0],[0,2],[1,1],[1,2],[1,3],[2,2],[3,2],[2,3]])
y = pd.Series([0,0,0,0,1,1,1,1])

```

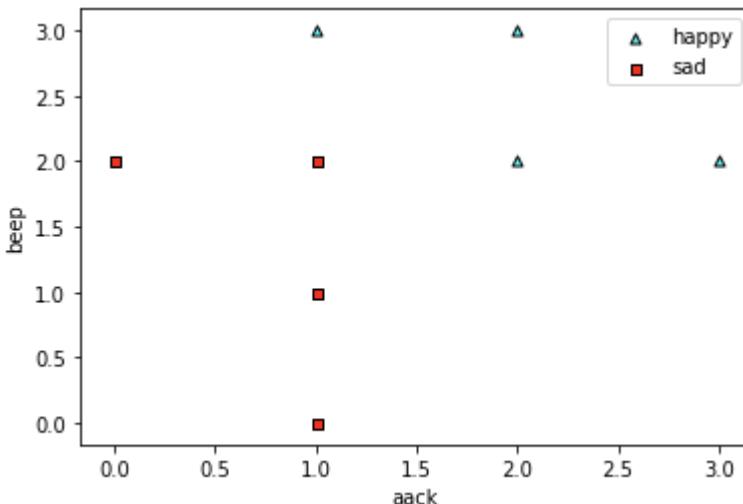


Figure 5.11. The plot of our dataset, where the happy points are triangles and the sad ones are squares.

We run the algorithm to get the classifier.

```

lr_algorithm(features, labels)
([0.4699999999999953, 0.0999999999999937], -0.6800000000000004)

```

The classifier we obtain has the following weights and biases.

- $w_1 = 4.7$
- $w_2 = 0.1$
- $b = -0.6$

The plot of the classifier (together with a plot of the previous classifiers at each of the epochs) is in Figure 5.12.

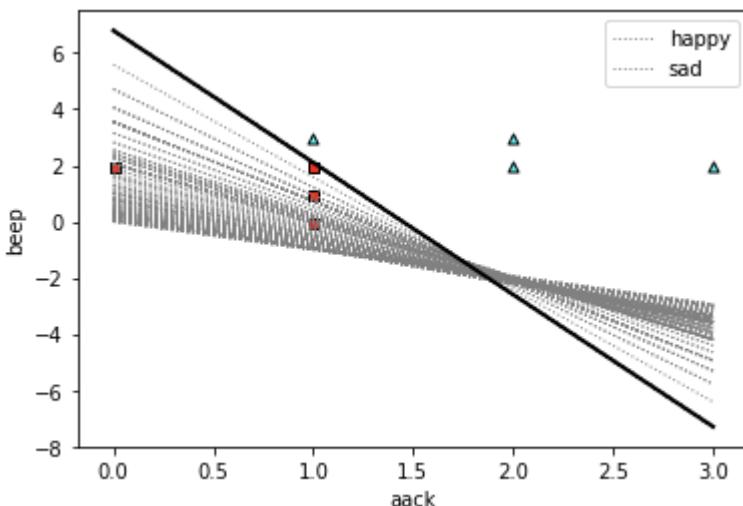


Figure 5.12. A plot of all the intermediate steps of the logistic regression algorithm. Notice that we start with a bad classifier, and slowly move towards a good one (the thick line).

And finally, the plot of the log loss can be seen in Figure 5.13. Notice that as we run the algorithm for more epochs, the log loss decreases drastically, which is exactly what we want.

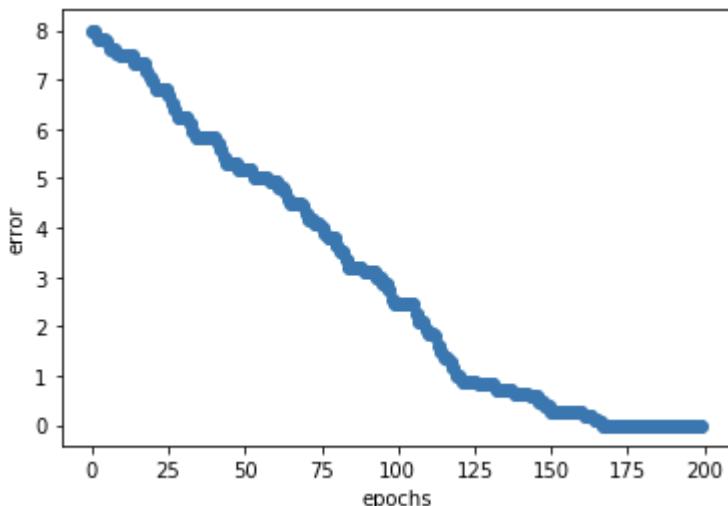


Figure 5.13. The error plot. Notice that the more epochs we run the logistic regression algorithm, the lower the error gets.

### 5.2.5 The logistic regression algorithm in Turi Create

In this section, I show you how to build a continuous perceptron classifier in Turi Create. The process is much easier, since the package will take care of most of the heavy lifting for us. First we have to load all the data into an SFrame.

```
import turicreate as tc
data = tc.SFrame({'X1': X[0], 'X2': X[1], 'y': y})
```

X1	X2	y
1	0	0
0	2	0
1	1	0
1	2	0
1	3	1
2	2	1
3	2	1
2	3	1

[8 rows x 3 columns]

Figure 5.14. The dataset that we have been working on.

Now, we train the classifier.

```
classifier = tc.logistic_classifier.create(data,
                                             features = ['X1', 'X2'],
                                             target = 'y',
                                             validation_set=None)
```

First, let's evaluate the coefficients of this model, with the following command.

```
classifier.coefficients
```

We get the following coefficients:

- $w_1 = 2.97$
- $w_2 = 2.5$
- $b = -8.96$  (the intercept)

They look different than the ones we obtained when we code it by hand, but that doesn't matter. The boundary lines are still similar, as we can see when we plot it in Figure 5.15.

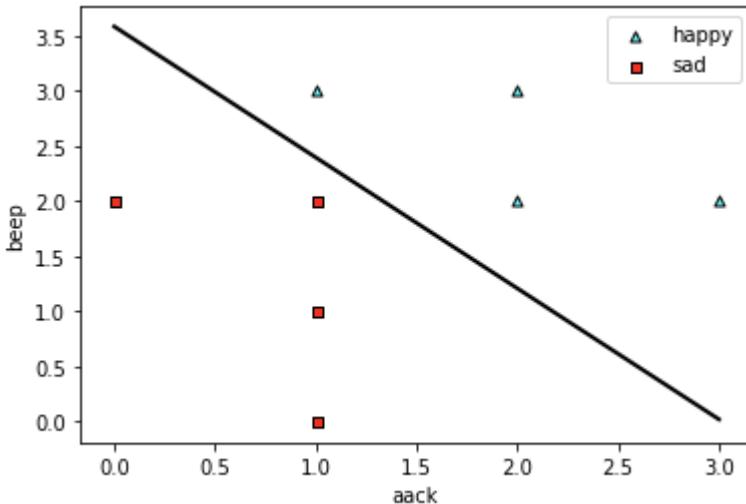


Figure 5.15. Plot of our dataset and a good classifier that splits the points.

### 5.3 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 4 I mentioned that classifying between more than two classes is hard for a discrete perceptron. However, for continuous perceptrons, this is possible.

If we had different labels, such as, for example, dog and cat, we could still use a perceptron. The perceptron would return a number between 0 and 1, which can be interpreted as the possibility that the data point is classified as a dog. To find the probability that the data point is classified as a cat, we simply take 1 minus the probability that the data point is classified as a dog.

However, if we have a dataset with three labels, say, dog, cat, and bird, we can't do this. What we can do instead is build three classifiers, one for dog, one for cat, and one for bird. Then we can make the classifiers vote, by deciding that the prediction for our data point corresponds to the classifier that scored it the highest. But there is a simpler way: each classifier returns a score, and then passes that score through a sigmoid function. Let's take one step back, and look only at the score. We'll generalize this sigmoid function to several classes.

As an example, say we have our three classifiers: a dog classifier, a cat classifier, and a bird classifier. For a particular data point, they all output scores in the following way:

- Dog classifier: 4
- Cat classifier: 2
- Bird classifier: 1

How do we turn these into probabilities? Well, here's an idea: why don't we divide all these three numbers by their sum, which is 7? Now we get the probabilities 4/7 for dog, 2/7 for cat, and 1/7 for bird. This works, but what if we have the following numbers:

- Dog classifier: 2
- Cat classifier: 0
- Bird classifier: -2

Their sum is 0, so when we try to divide by the sum, we have a problem. What if we square them? Then the sum is not zero, but our probabilities would be 2/4 for dog, 0 for cat, and 2/4 for bird. This also doesn't work, since we want the cat to have a higher probability than the bird (which has a negative score). We need some function to apply to these scores, which is high if the input is high, low if the input is low, and never negative. What function has these properties? If you said the exponential function, you are on the right track. Any power function, or function that raises some number to the power of the input, works. For example, the functions  $2^x$ ,  $3^x$ , or  $10^x$  would work perfectly. As a matter of fact, they all work the same, since all you need to go from one to the other is to scale the exponent. By default, in math, when you don't know what exponent to pick, you pick e, since the function  $e^x$  has wonderful mathematical properties (for instance, it is its own derivative!). So we'll go with  $e^x$ . We apply it to all the scores, to get the following.

- Dog classifier:  $e^2 = 7.389$
- Cat classifier:  $e^0 = 1$
- Bird classifier:  $e^{-2} = 0.135$

Now, we do what we did before, we normalize, or divide by the sum of these three numbers in order for them to add to 1. The sum is  $7.389 + 1 + 0.135 = 8.524$ , so we get the following:

- Probability of dog:  $7.389/8.524 = 0.867$
- Probability of cat:  $1/8.524 = 0.117$
- Probability of bird:  $0.135/8.524 = 0.016$

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

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

This formula is known as the *softmax formula*.

What would happen if we use the softmax formula for only two classes? We obtain the sigmoid function. I encourage you to convince yourself of this as an exercise.

## 5.4 Summary

- Continuous logistic classifiers (or continuous perceptrons) are very similar to their discrete counterparts, except instead of making a discrete prediction such as 0 or 1, they predict any number between 0 and 1.
- Continuous perceptrons are more useful than discrete perceptrons, since they give us more information. Continuous perceptrons don't just tell us which class the classifier predicts, but it also gives us a probability. It would assign low probabilities to points it predicts to have label 0, and high probabilities to points it predicts to have label 1.
- The log loss is an error function for continuous perceptrons. It is calculated separately for every point as the natural logarithm of the probability that the point is classified correctly according 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 regression trick takes a labelled data point and a 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, since the perceptron trick doesn't move the line if the point is correctly classified.
- The logistic regression algorithm is used to fit a continuous perceptron to a labelled dataset. It consists of starting with a continuous perceptron with random weights, and continuously picking a random point and applying the logistic regression trick in order 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.

# 6

## *Using probability to its maximum: The naive Bayes algorithm*

### This chapter covers

- What is Bayes theorem?
- When are events dependent or independent?
- The prior and the posterior probabilities.
- Calculating conditional probabilities based on events.
- What is the naive Bayes algorithm?
- Using the naive Bayes algorithm to predict if an email is spam or ham, based on the words in the email.
- Coding the naive Bayes algorithm in Python.

Naive Bayes is a very important machine learning algorithm used for prediction. As opposed to the previous algorithms you've learned in this book, such as the perceptron algorithm, in which the prediction is discrete (0 or 1), the naive Bayes algorithm is purely probabilistic. This means, the prediction is a number between 0 and 1, indicating the probability that a label is positive. The main component of naive Bayes is Bayes Theorem.

Bayes Theorem is a fundamental theorem in probability and statistics, and it helps us calculate probabilities. The more we know about a certain situation, the more accurate the probability is. For example, let's say we want to find the probability that it will snow today. If we have no information of where we are and what time of the year it is, we can only come up with a vague estimate.

However, if we are given information, we can make a better estimation of the probability. If I tell you, for example, that we are in Canada, then the probability that it will snow today

increases. If I were to tell you that we are in Jamaica, this probability decreases drastically. Bayes theorem helps us calculate these new probabilities, once we know more about the situation.

However, when we have a lot of information, the math becomes complicated. If I were to ask you the probability that it will snow today given the following: we are in Canada, it is February, the temperature is -10 degrees Celsius, and the humidity is 30%, then you would assume that the probability of snow is quite high, but exactly how high? Bayes theorem has a hard time dealing with so much information. But there is a very simple trick we can use, which makes the calculation much easier. The trick relies on making an assumption that is not necessarily true, but that works pretty well in practice and simplifies our life quite a lot. This assumption is the heart of the naive Bayes algorithm.

In this chapter, I will show you Bayes theorem with some real life examples. We'll start by studying an interesting and slightly surprising medical example. Then we'll dive deep into the naive Bayes algorithm by applying it to a very common problem in machine learning: spam classification. We'll finalize by coding the algorithm in Python, and using it to make predictions in a real spam email dataset.

## 6.1 Sick or healthy? A story with Bayes Theorem

Consider the following scenario. Your (slightly hypochondriac) friend calls you, and the following conversation unfolds:

**You:** Hello!

**Friend:** Hi, I have some terrible news!

**You:** Oh no, what is it?

**Friend:** I heard about this terrible and rare disease, and I went to the doctor to be tested for it. The doctor said she would administer a very accurate test. Then today she called me and told me that I tested positive! I must have the disease!

Oh oh! What do we say to our friend? First of all, let's calm them down, and try to figure out if it is likely that our friend has the disease.

**You:** First let's calm down, mistakes happen in medicine. Let's try to see how likely it is that you actually have the disease. How accurate did the doctor say the test was?

**Friend:** She said it was 99% accurate. That means I'm 99% likely to have the disease!

**You:** Wait, let's look at *all* the numbers. How likely is it to have the disease, regardless of the test? How many people have the disease?

**Friend:** I was reading online, and it says that on average, 1 out of every 10,000 people have the disease.

**You:** Ok, let me get a piece of paper (\*puts friend on hold\*).

Let's stop for a quiz. How likely do you think it is that your friend has the disease?

**Quiz** In what range do you think is the probability that your friend has the disease?

- a) 0-20%
- b) 20-40%
- c) 40-60%
- d) 60-80%
- e) 80-100%

Let's calculate it. To summarize, we have the following two pieces of information:

- The test is correct 99% of the time. To be more exact (we checked with the doctor to confirm this), on average, out of every 100 healthy people, the test correctly diagnoses 99 of them, and out of every 100 sick people, the test correctly diagnoses 99 of them. Therefore, both on healthy and sick people, the test has an accuracy of 99%.
- On average, 1 out of every 10,000 people has the disease.

Let's do some rough calculations to see what the probability would be. Let's pick a random group of 10,000 people. Since on average, one out of every 10,000 people are sick, then we expect one of these people to have the disease. Now, let's run the test on the remaining 9,999 healthy ones. Since the test makes mistakes 1% of the time, we expect 1% of these 9,999 healthy people to be misdiagnosed as sick. This means, 99.9 people on average will be diagnosed as sick. Let's round it up to 100. This means, out of 10,000, 100 of them would be healthy people misdiagnosed as sick, and one person will be a sick person correctly diagnosed as sick. Thus, the probability of being *the* one sick person among those diagnosed as sick, is simply 1%. Quite less than what we thought! So we can get back to our friend.

**You:** Don't worry, based on the numbers you gave me, the probability that you have the disease *given that* you tested positive is actually only around 1%!

**Friend:** Oh my God, really? That's such a relief, thank you!

**You:** Don't thank me, thank *math* (winks eye).

And you hang up. But you, being a math person, are still not content with the calculation you made. There was a bit of rounding up there. Let's get more rigorous with the math, and calculate the actual probability. The facts that we have so far are the following:

1. Out of every 10,000 people, 1 has the disease.
2. Out of every 100 sick people who take the test, 99 test positive, and one tests negative.
3. Out of every 100 healthy people who take the test, 99 test negative, and one tests positive.

Ok, let's take a large group of people, say, 1 million people. Out of 1 million people, how many of them have the disease? Since 1 out of every 10 thousand has the disease, then a quick calculation shows that among 1 million people, 100 of them are sick. So we have our first piece of information.

Out of 1 million people:

- 999,900 of them are healthy, and
- 100 of them are sick.

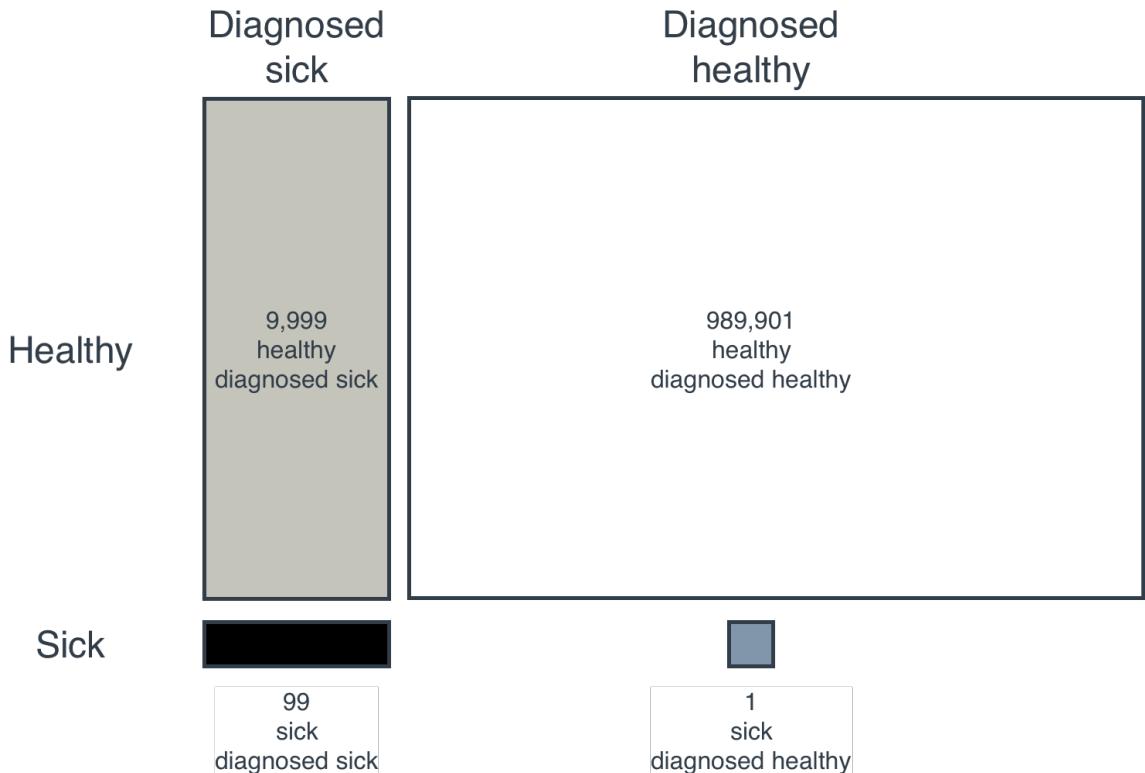
Now, let's say we test all of them. Since the test is correct 99% of the time, both among the healthy and among the sick, then we have a second piece of information.

Out of 99,900 healthy people:

- 99% of them, or 989,901, are diagnosed as healthy, and
- 1% of them, or 9,999, are diagnosed as sick.

Out of 100 sick people:

- 99% of them, or 99, are diagnosed as sick, and
- 1% of them, or 1, is diagnosed as healthy.



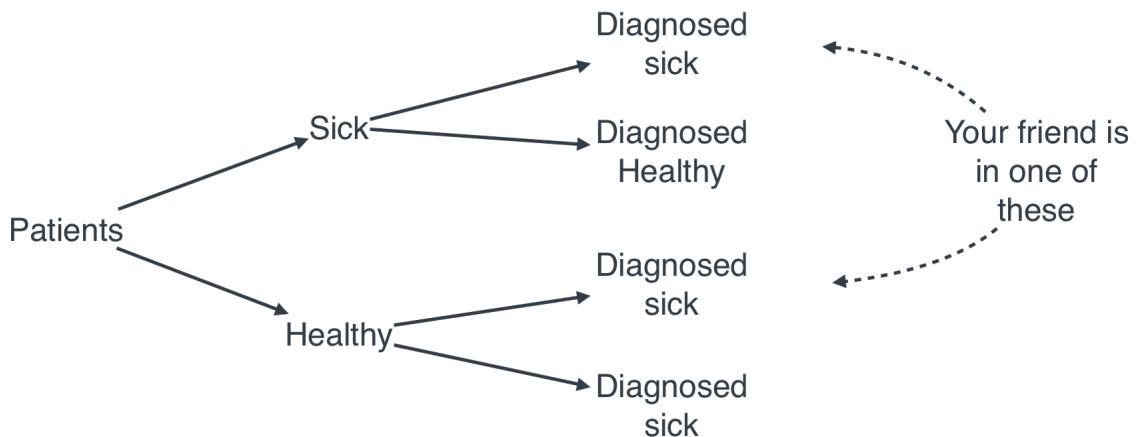
**Figure 6.1.** Among our 1,000,000 patients, only 100 of them are sick (bottom of the figure). Among the 10,098 diagnosed sick (on the left), only 99 of them are actually sick. The remaining 9,999 are healthy, and misdiagnosed as sick. Therefore, if our friend was diagnosed sick, he has a much higher chance to be among the healthy misdiagnosed as sick (top left) than to be among the sick correctly diagnosed as sick (bottom left).

Now the question, if your friend tested positive, which means he was diagnosed as sick, then that means he could belong to two groups. He could be one of the 9999 *healthy* people who were (incorrectly) diagnosed as sick, or he could be one of the 99 *sick* people who were (correctly) diagnosed as sick. What is the probability that he belongs to the 99, and not the 999? A simple calculation will do. In total, we have  $99+9999 = 10,098$  people who were diagnosed as sick. If your friend were to be among the 99 sick ones, the probability is  $99/10,098 = 0.009$ . That is 0.09%, less than 1%.

This is a bit surprising, if the test is correct 99% of the time, why on earth was it so wrong? Well, the test is not bad, if it's only wrong 1% of the time. But since one person out of every 10,000 is sick with the disease, that means a person is sick 0.01% of the time. What is more likely, to be among the 1% of the population that got misdiagnosed, or to be among the 0.01% of the population that is sick? The 1%, although a small group, is massively larger than

the 0.01%. So the test has an error rate so much larger than the rate of being sick, that it ends up not being as effective as we think.

A way to look at this is using tree-like diagrams. In our diagram, we start with a root at the left, which branches out into two possibilities: that your friend is sick or healthy. Each of these two branches out into two possibilities: that your friend gets diagnosed healthy or diagnosed sick. The tree looks like Figure 6.2.



**Figure 6.2.** The tree-like diagram. At the left, the root splits into two possibilities, sick and healthy. Each possibility splits into two more, diagnosed sick or diagnosed healthy. On the right, we have the four final possibilities. Since your friend was diagnosed sick, then he must be in the first or third group, namely, among the sick who got diagnosed sick, or among the healthy who got diagnosed sick.

Notice that in the tree in Figure 6.2, the patients get divided into four groups:

1. Sick patients who were diagnosed sick.
2. Sick patients who were diagnosed healthy.
3. Healthy patients who were diagnosed sick.
4. Healthy patients who were diagnosed healthy.

Now let's count how many people are in each of the groups, starting with 1,000,000 patients, as we did above. The results are in Figure 6.3.



Figure 6.3. The tree-like diagram with numbers. We start with 1 million patients. 100 of them are sick, and 999,900 of them are healthy. Out of the 100 sick, 1 gets misdiagnosed as healthy, and the remaining 99 get correctly diagnosed as sick. Out of the 999,900 healthy patients, 9,999 get misdiagnosed as sick, and the remaining 989,901 are correctly diagnosed as healthy.

From Figure 6.3, we can again see that the probability that since among the patients that are diagnosed sick, 99 are actually sick, and 9,999 are healthy, from which we again can deduce that the probability that our friend is sick is:

$$\text{Probability of being sick when being diagnosed sick} = \frac{99}{99 + 9,999} = 0.0098.$$

### 6.1.1 Prelude to Bayes Theorem: The prior, the event, and the posterior

We now have all the tools to state Bayes theorem. The main goal of Bayes theorem is calculating a probability. At the beginning, with no information in our hands, we can only calculate an initial probability, which we call the *prior*. Then, an event happens, which gives us information. After this information, we have a much better estimate of the probability we want to calculate. We call this better estimate the *posterior*.

**PRIOR** The initial probability that we calculate.

**EVENT** What gives us information to calculate better probabilities.

**POSTERIOR** The final (and more accurate) probability that we calculate using the prior probability and the event.

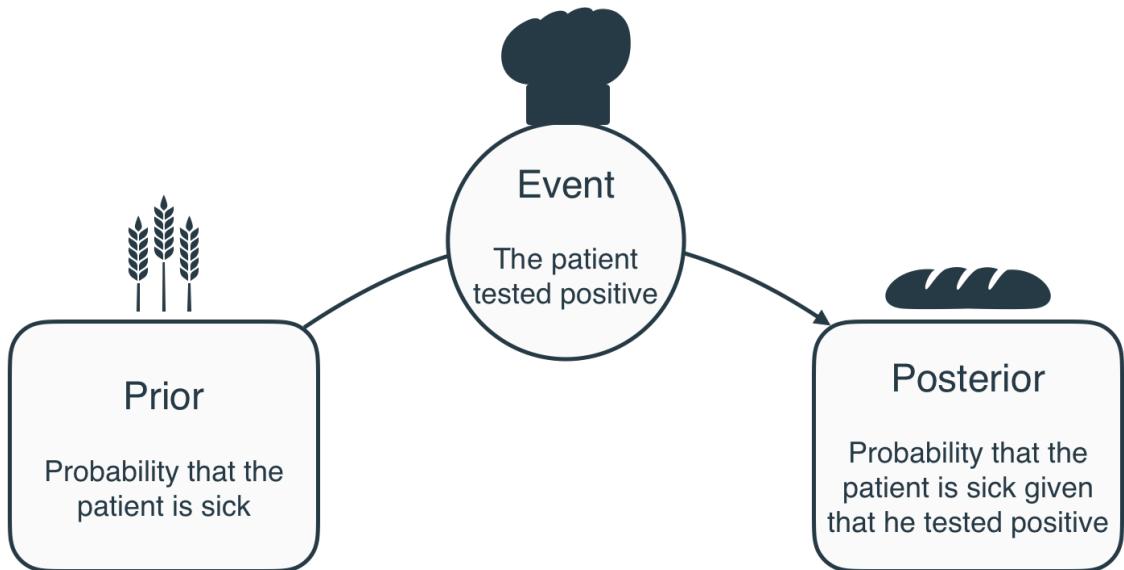


Figure 6.4. The prior, the event, and the posterior. The prior is the ‘raw’ probability, namely, the probability we calculate when we know very little. The event is the information that we obtain which will help us refine our calculation of the probability. The posterior is the ‘cooked’ probability, or the much more accurate probability that we calculate when we have more information.

In our case, we needed to calculate the probability that a patient is sick.

1. Initially, this probability is  $1/10,000$ , since we don’t have any other information, other than the fact that one out of every 10,000 patients is sick. So this  $1/10,000$ , or  $0.0001$  is the prior.
2. But all of a sudden, new information comes to light. In this case, the patient took a test, and tested positive.
3. After coming out positive, we recalculate the probability that the patient is sick, and that comes out to  $0.0098$ . This is the posterior.

Bayes Theorem is one of the most important building blocks of probability and of machine learning. It is so important that several fields are named after it, *Bayesian learning*, *Bayesian statistics*, *Bayesian analysis*, and so on. In this chapter, we’ll learn Bayes theorem and a very important algorithm that will help us put in practice when analyzing data: the naive Bayes algorithm. In a nutshell, the naive Bayes algorithm does what most classification algorithms do, which is predict a label out of a set of features. Naive Bayes will simply perform several probability calculations using the features, and output a final probability for what it thinks the label is.

## 6.2 Use-case: Spam detection model

Now consider the following situation, you are reading your email, and you are sick and tired of receiving spam (garbage) email. You would like to build a tool that helps you separate spam from non-spam emails (as you may recall from section 1.4.1, non-spam emails are called ham).

More specifically, given a new email, we'd like to find the probability that this email is spam or ham. In that way, we can send the emails with the highest probability of being spam directly to the spam folder, and keep the rest in our inbox. This probability should depend on information about the new email (words on the email, sender, size, and so on). For example, an email with the word 'sale' on it is more likely to be spam

### 6.2.1 Finding the prior: The probability that any email is spam

What is the probability that an email is spam? That is a hard question, but let's try to make an estimate. We look at our current inbox and count how many emails are spam and ham. Say there are 100 emails, of which 20 are spam, and 80 ham. Thus, 20% of the emails are spam. So if we want to make a decent estimate, we can say that *to the best of our knowledge*, the probability that a new email is spam is 20%. This is the prior probability. The calculation can be seen in Figure 6.5, where the spam emails are colored dark grey, and the ham emails white.

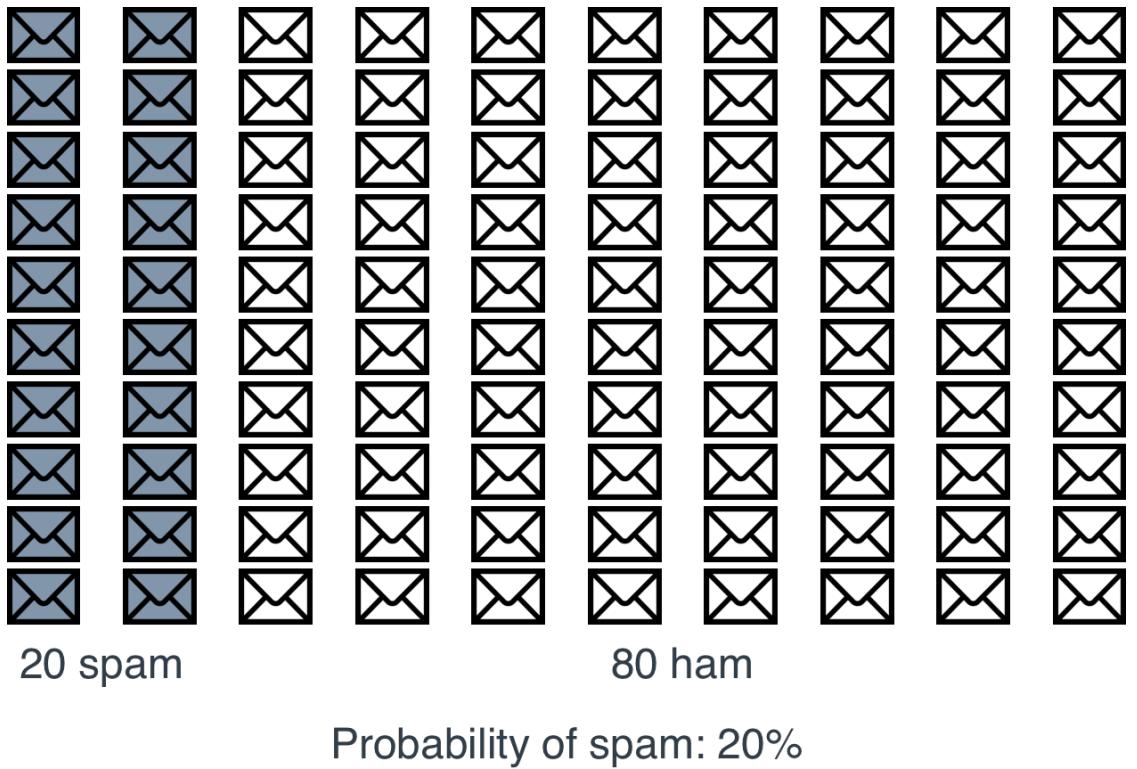


Figure 6.5: We have a dataset with 100 emails, 20 of which are spam. A good estimate for the probability that any email is spam is 20%. This is the prior probability.

### 6.2.2 Finding the posterior: The probability that an email is spam knowing that it contains a particular word

Of course, not all emails are created equally. We'd like to come up with a more educated guess for the probability, using properties of the email. We can use many properties, such as sender, size, words in the email, etc. For this particular application, we'll only use the words in the email, but I encourage you to go through the example thinking how this could be used with other properties.

Let's say we find that particular word, say, the word 'lottery', tends to appear in spam emails more than in ham emails. As a matter of fact, among the spam emails, it appears in 15 of them, while among the ham emails, it only appears in 5 of them. Therefore, among the 20 emails containing the word 'lottery', 15 of them are spam, and 5 of them are ham. Thus, the probability that an email containing the word 'lottery' is spam, is precisely 15/20, or 75%. That is the posterior probability.

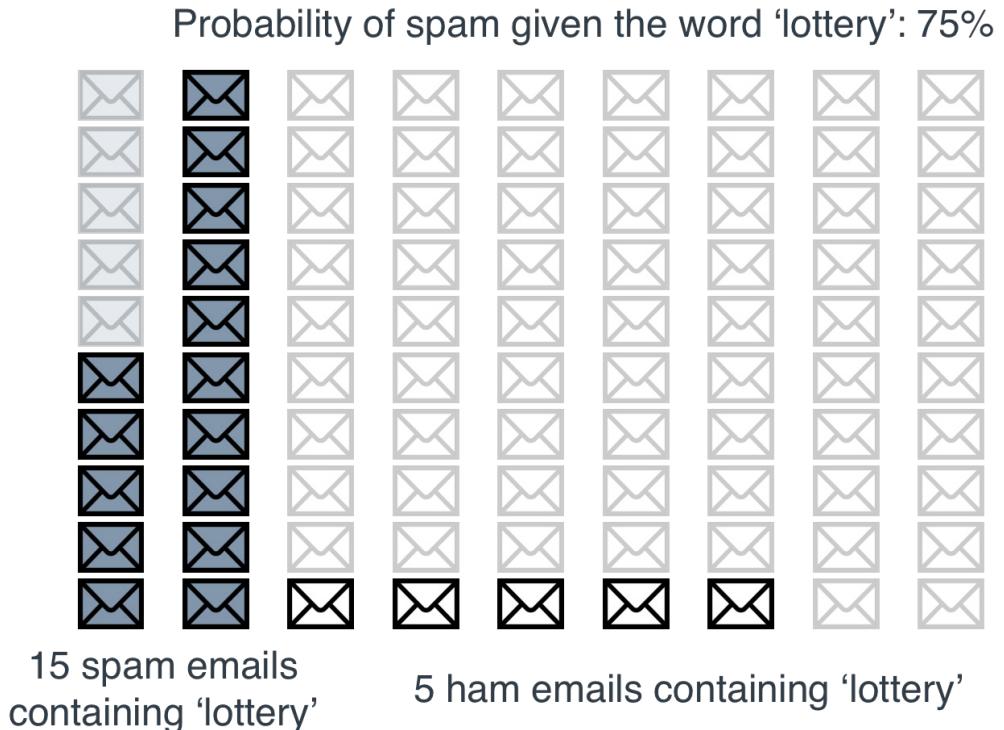


Figure 6.6: We have removed (blurred) the emails that don't contain the word 'lottery'. All of a sudden our probabilities change. Now, since among the emails that contain the word 'lottery', there are 15 spam emails and 5 ham emails. Thus, the probability that an email containing the word 'lottery' is spam, is 75%.

There we have it, that is Bayes Theorem. To summarize:

- The **prior** is 20%, the probability that an email is spam, knowing nothing about the email.
- The **event** is that the email contains the word ‘lottery’. This helped us make a better estimate of the probability.
- The **posterior** probability is 80%. This is the probability that the email is spam, *given that* it contains the word ‘lottery’.

### 6.2.3 What the math just happened? Turning ratios into probabilities

One way to visualize the previous example is with a tree of all the four possibilities, namely: that the email is spam or ham, and that it contains the word ‘lottery’ or not. We draw it in the following way, we start by the root, which splits into two branches. The top branch corresponds to spam, and the bottom branch corresponds to ham. Each of the branches splits into two more branches, namely, when the email contains the word ‘lottery’ or not. The tree

looks like Figure 6.6. Notice that in this tree, we've also attached how many emails out of the total 100 belong to each particular group.

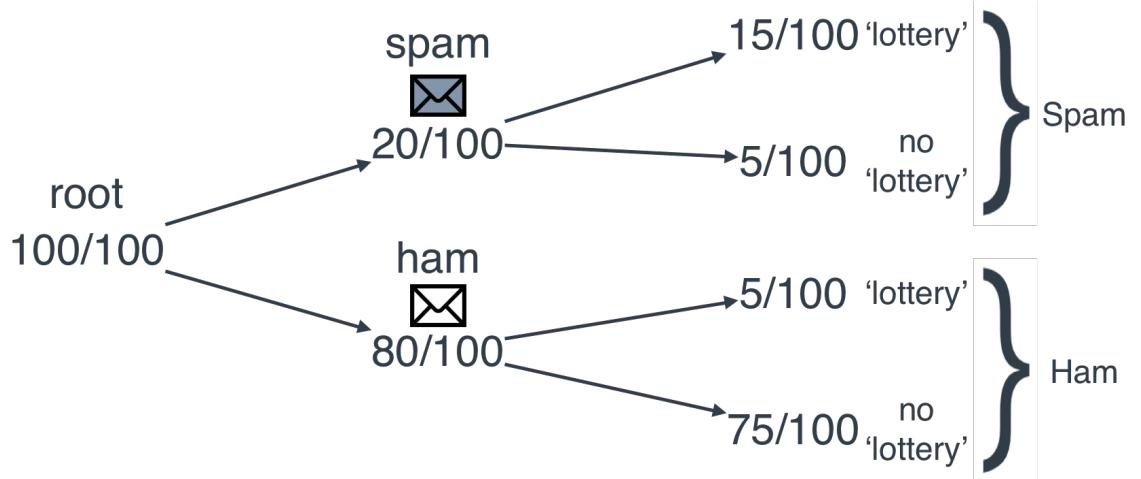


Figure 6.7: The tree of possibilities. The root splits into two branches: ‘spam’ and ‘ham’. Then each of these splits into two branches: when the email contains the word ‘lottery’, and when it doesn’t.

Once we have this tree, and we want to calculate the probability of an email being spam *given that* it contains the word ‘lottery’, we simply remove all the branches where the emails that don’t contain the word ‘lottery’, thus obtaining Figure 6.7.

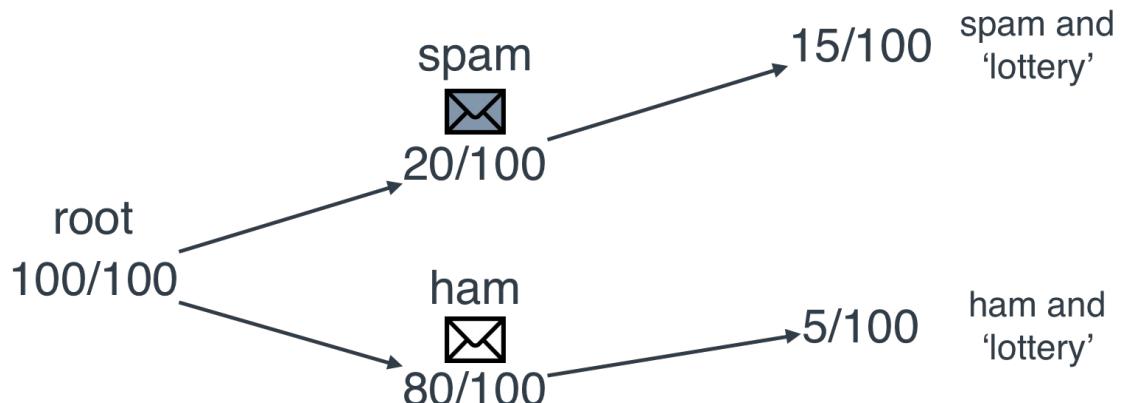


Figure 6.8: From the previous tree, we have removed the two branches where the emails don't contain the word ‘lottery’.

Now, we simply have 20 emails, 15 of them are spam and 5 of them are ham. Thus, the probability of an email being spam given that it contains the word 'lottery' is 15/20, also called  $\frac{3}{4}$ , or 75%.

But we've already done that, so what is the benefit of the diagram? Aside from making things simpler, the benefit is that normally, the information we have is based on probabilities, and not in number of emails. Many times, we don't know how many emails are spam or ham, all we know is the following:

- The probability that an email is spam is  $\frac{1}{5}$ .
- The probability that a spam email contains the word 'lottery' is  $\frac{3}{4}$ .
- The probability that a ham email contains the word 'lottery' is  $\frac{1}{40}$ .
- **Question:** What is the probability that an email that contains the word 'lottery' is spam?

First of all, let's check if this is enough information. Do we know the probability that an email is ham? Well, we know that the probability that it is spam is  $\frac{1}{5}$ , or 20%. Since the *only* other possibility is that an email is ham, then it must be the complement, or  $\frac{4}{5}$ , namely, 80%. This is a very important rule, the rule of complementary probabilities.

**RULE OF COMPLEMENTARY PROBABILITIES** For an event  $E$ , the complement of the event  $E^c$ , denoted  $E^c$ , is the event opposite to  $E$ . The probability of  $E^c$  is 1 minus the probability of  $E$ , namely,

$$P(E^c) = 1 - P(E).$$

So we have the following:

- $P(\text{spam}) = \frac{1}{5}$ . The probability of an email being spam.
- $P(\text{ham}) = \frac{4}{5}$ . The probability of an email being ham.

Now, let's look at the other information. The probability that a spam email contains the word 'lottery' is  $\frac{3}{4}$ . This can be read as, the probability that an email contains the word 'lottery' *given that* it is spam, is  $\frac{3}{4}$ . This is a conditional probability. The condition is that the email is spam. We denote condition by a vertical bar, so this can be written as  $P(\text{lottery}^f | \text{spam})$ . Now, what would  $P(\text{no lottery}^f | \text{spam})$ , i.e., the probability that an email does *not* contain the word 'lottery'? Well, the complimentary event is that a spam email contains the word 'lottery', so this probability has to be  $1 - P(\text{lottery}^f | \text{spam})$ . Now we can calculate other probabilities, as follows:

- $P(\text{lottery}^f | \text{spam}) = \frac{3}{4}$ . The probability that a spam email contains the word 'lottery'.
- $P(\text{no lottery}^f | \text{spam}) = \frac{1}{4}$ . The probability that a spam email does not contain the word 'lottery'.
- $P(\text{lottery}^f | \text{ham}) = \frac{1}{16}$ . The probability that a ham email contains the word 'lottery'.
- $P(\text{no lottery}^f | \text{ham}) = \frac{15}{16}$ . The probability that a ham email does not contain the word 'lottery'.

The next thing we do is find the probabilities of two events happening *at the same time*. More specifically, we want the following four probabilities:

- The probability that an email is spam *and* contains the word 'lottery'.
- The probability that an email is spam *and* does not contain the word 'lottery'.
- The probability that an email is ham *and* contains the word 'lottery'.
- The probability that an email is ham *and* does not contain the word 'lottery'.

These events are called *intersections* of events, and denoted with the symbol  $\cap$ . Thus, we need to find the following probabilities:

- $P(\text{'lottery'} \cap \text{spam})$
- $P(\text{no 'lottery'} \cap \text{spam})$
- $P(\text{'lottery'} \cap \text{ham})$
- $P(\text{no 'lottery'} \cap \text{ham})$

Let's look at some numbers. We know that  $\frac{1}{5}$ , or 20 out of 100 emails are spam. Out of those 20,  $\frac{3}{4}$  of them contain the word 'lottery'. At the end, we multiply these two numbers,  $\frac{1}{5}$  times  $\frac{3}{4}$ , to obtain  $\frac{3}{20}$ .  $\frac{3}{20}$  is the same as  $15/100$ , which is precisely the number of emails that are spam and contain the word 'lottery'. What we did was the following: We multiplied the probability that an email is spam *times* the probability that a spam email contains the word 'lottery', to obtain the probability that an email is spam *and* contains the word lottery. The probability that a spam email contains the word 'lottery' is precisely the conditional probability, or the probability that an email contains the word 'lottery' *given that* it is a spam email. This gives rise to the multiplication rule for probabilities.

**PRODUCT RULE OF PROBABILITIES** For events  $E$  and  $F$ , the probability of their intersection is precisely,

$$P(E \cap F) = P(E | F) \cdot P(F).$$

So now we can calculate these probabilities:

- $P(\text{'lottery'} \cap \text{spam}) = P(\text{'lottery'} | \text{spam}) \cdot P(\text{spam}) = \frac{1}{5} \cdot \frac{3}{4} = \frac{3}{20}$
- $P(\text{no 'lottery'} \cap \text{spam}) = P(\text{no 'lottery'} | \text{spam}) \cdot P(\text{spam}) = \frac{1}{5} \cdot \frac{1}{4} = \frac{1}{20}$
- $P(\text{'lottery'} \cap \text{ham}) = P(\text{'lottery'} | \text{ham}) \cdot P(\text{ham}) = \frac{4}{5} \cdot \frac{1}{16} = \frac{1}{20}$
- $P(\text{no 'lottery'} \cap \text{ham}) = P(\text{no 'lottery'} | \text{ham}) \cdot P(\text{ham}) = \frac{4}{5} \cdot \frac{15}{16} = \frac{15}{20}.$

These probabilities are summarized in Figure 6.9. Notice that the product of the probabilities on the edges are the probabilities at the right.

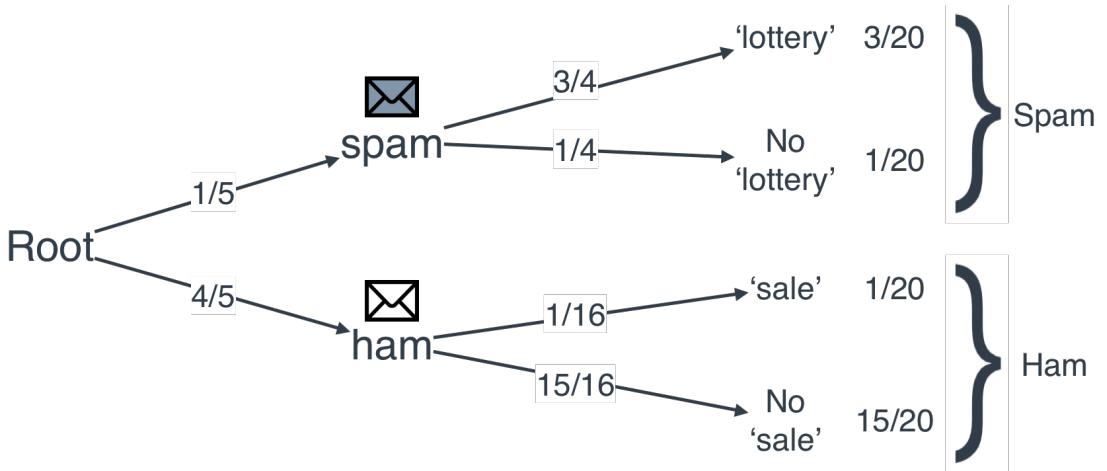


Figure 6.9: The same tree from Figure 6.8, but now with probabilities. From the root, two branches come out, one for spam emails and one for ham emails. In each one, we record the corresponding probability. Each branch again splits into two leaves, one for emails containing the word ‘lottery’, and one for emails not containing it. In each branch, again, we record the corresponding probability. Notice that the product of these probabilities is the probability at the right of each leaves. For example, for the top leaf,  $\frac{1}{5} * \frac{3}{4} = \frac{3}{20}$ .

We’re almost done. We want to find  $P(\text{spam} \mid \text{'lottery'})$ . This means, the probability that an email is spam *given that* it contains the word ‘lottery’. Among the four events we just studied, in only two of them does the word ‘lottery’ appear. Thus, we only need to consider those, namely:

- $P(\text{'lottery'} \cap \text{spam}) = \frac{3}{20}$
- $P(\text{'lottery'} \cap \text{ham}) = \frac{1}{20}$

In other words, we need only consider two branches from Figure 6.10. These are the first and the third, namely, those in which the email contains the word ‘lottery’.

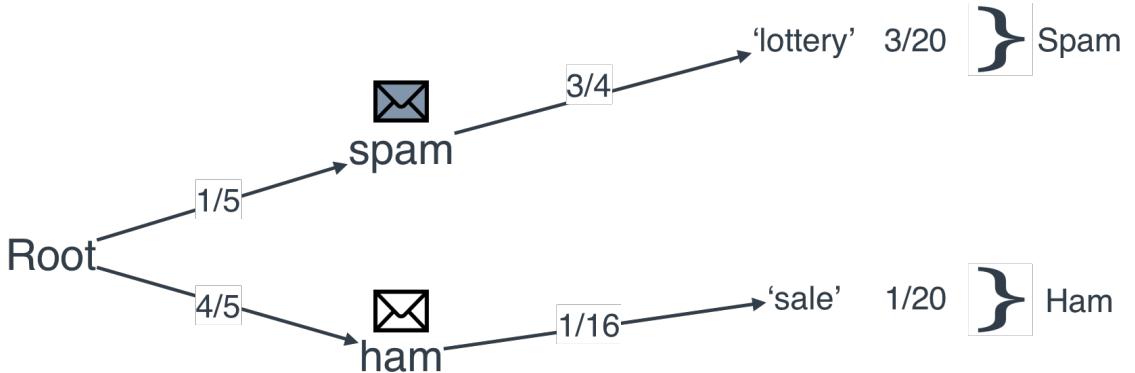


Figure 6.10: From the tree in Figure 6.9, we have removed the two branches where the emails don't contain the word 'lottery'.

The first should be the probability that an email is spam, and the second should be the probability that the email is ham. These two probabilities don't add to 1. However, since we now live in a world in which the email contains the word 'lottery', then these two are the only possible scenarios. Thus, their probabilities *should* add to 1. Furthermore, they should still have the same relative ratio with respect to each other. The way to fix this, is to normalize, namely, to find two numbers that are in the same relative ratio with respect to each other as 3/20 and 1/50, but that add to 1. The way to find these, is to divide both of them by the sum. In this case, the numbers become  $\frac{3/20}{3/20 + 1/20}$  and  $\frac{1/20}{3/20 + 1/20}$ . These simplify to  $\frac{3}{4}$  and  $\frac{1}{4}$ , which are the desired probabilities. Thus, we conclude that

- $P(\text{spam} \mid \text{'lottery'}) = \frac{3}{4} = 75\%$
- $P(\text{ham} \mid \text{'lottery'}) = \frac{1}{4} = 25\%.$

This is exactly what we figured out when we counted the emails. To really wrap up this information, we need a formula. Let's figure out this formula. We had two probabilities, the probability that an email is spam *and* contains the word 'lottery', and the probability that an email is spam *and* does not contain the word lottery. In order to get them to add to 1, we normalized the probabilities. This is the same thing as dividing each one of them by their sum. In math terms, we did the following:

$$P(\text{spam} \mid \text{'lottery'}) = \frac{P(\text{'lottery'} \cap \text{spam})}{P(\text{'lottery'} \cap \text{spam}) + P(\text{'lottery'} \cap \text{ham})}.$$

If we remember what these two probabilities were, using the multiplication rule, we get the following:

$$P(\text{spam} \mid \text{'lottery'}) = \frac{P(\text{'lottery'} \mid \text{spam}) \cdot P(\text{spam})}{P(\text{'lottery'} \mid \text{spam}) \cdot P(\text{spam}) + P(\text{'lottery'} \mid \text{ham}) \cdot P(\text{ham})}.$$

To verify, we plug in the numbers to get:

$$P(\text{spam} \mid \text{'lottery'}) = \frac{\frac{1}{5} \cdot \frac{3}{4}}{\frac{1}{5} \cdot \frac{3}{4} + \frac{4}{5} \cdot \frac{1}{16}} = \frac{\frac{3}{20}}{\frac{3}{20} + \frac{1}{20}} = \frac{\frac{3}{20}}{\frac{4}{20}} = \frac{3}{4}.$$

And this is the formula for Bayes Theorem! More formally:

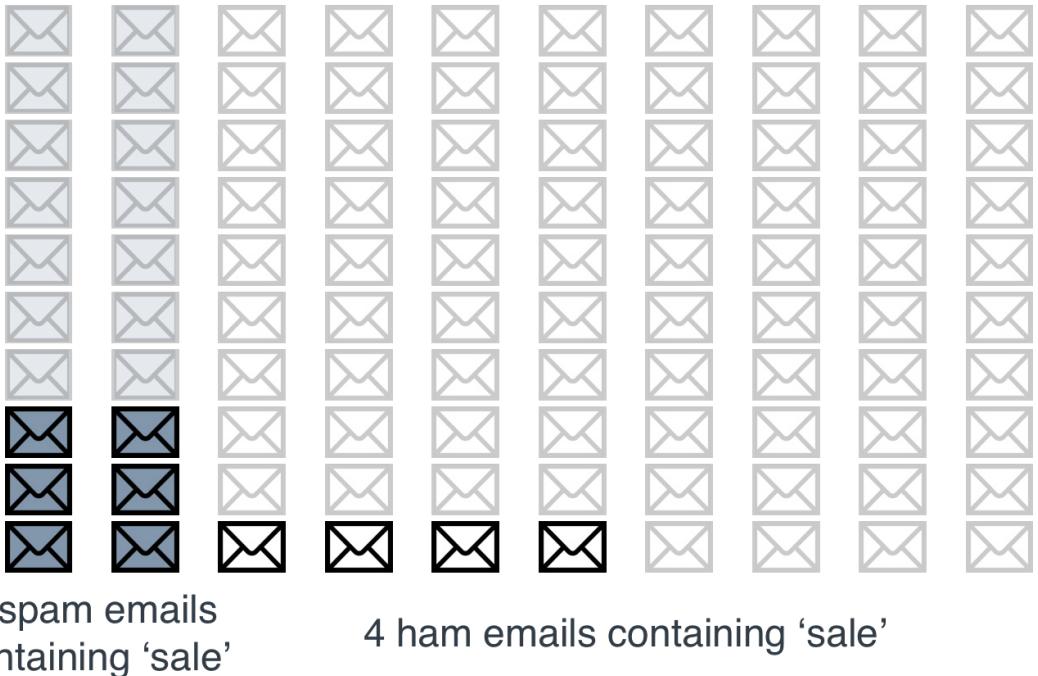
**BAYES THEOREM** For events E and F,

$$P(E \mid F) = \frac{P(F \mid E) \cdot P(E)}{P(F \mid E) \cdot P(E) + P(F \mid E^c) \cdot P(E^c)}.$$

#### 6.2.4 What about two words? The naive Bayes algorithm

Okay, this can't be the end of it, right? There are a lot more words in the dictionary. Let's say we notice that another word, the word 'sale' also tends to appear a lot in spam email. We can do a similar calculation. Let's count how many spam and ham emails contain the word 'sale'. We go over all our emails, and realize that the word 'sale' appears in 6 of the spam emails, and 4 of the ham emails. Thus, out of these 10 emails that contain the word 'sale', 60% (or 6 of them) are spam and 40% (or 4 of them) are ham. We conclude that based on our data, the probability that an email is spam containing the word 'sale' is 60%.

## Probability of spam given the word ‘sale’: 60%



**Figure 6.11:** In a similar calculation as with the word ‘lottery’, we look at the emails containing the word ‘sale’. Among these (unblurred) emails, there are 6 spam and 4 ham. Thus, the probability that an email containing the word ‘sale’ is spam is 60%.

We can do this for all the words. But the question is, how do we combine these probabilities? It’s not very useful to know that an email containing the words ‘lottery’, ‘sale’, and ‘hamburger’ has a 75% probability of being spam thanks to the word ‘lottery’, a 60% probability of being spam thanks to the word ‘sale’, and a 30% probability of being spam thanks to the word ‘hamburger’. We’d like to combine these into one probability. This is where the naive Bayes algorithm comes in.

For now, let’s simplify things by only using the two words ‘lottery’ and ‘sale’. So here’s an idea (a bad idea), let’s look at all the emails that have both words, ‘lottery’ *and* ‘sale’. Count how many of them are spam, how many of them are ham, and find the probability based on those two numbers using Bayes theorem. Tada!

Does that work? Well, not so much. Can you see a potential problem there? What if no emails in our dataset have the words ‘lottery’ and ‘sale’? We only have 100 emails, if only 20 of them have the word ‘lottery’ and 10 have the word ‘sale’, it could be that maybe only 1 or 2

have both words, or maybe none! Moreover, what if we also add the word ‘hamburger’. Out of 100 emails, how many may have all words, ‘lottery’, ‘sale’, and ‘hamburger’? Probably none.

So what’s the solution here, should we gather more data? That is always a good idea, but many times we can’t, so we have to deal with the data we have. In some way, we can’t rely on the emails that contain both words ‘lottery’ and ‘sale’. We need to make an estimate of this number of emails. We may have to make an assumption to do this. A *naive* assumption, perhaps? Yes a naive assumption is exactly what we’re going to make.

**NAIVE ASSUMPTION** Say we have 100 emails. 20 of them have the word ‘lottery’, and 10 have the word ‘sale’. This means that the probability that an email contains the word ‘lottery’ is 20%, and that it contains the word ‘sale’ is 10%. Let’s assume then, that among the 20% of emails that contain the word ‘lottery’, exactly 10% of them contain the word ‘sale’, that is, 2% of the total emails. Our naive assumption, then, is that 2% of the emails contain the words ‘lottery’ and ‘sale’.

What our naive assumption really means is that the event of containing the words ‘lottery’ and ‘sale’ are *independent*. This is, the appearance of one of the words in no way affects the appearance of the other one. This means, if 10% of the emails contain the word ‘sale’, then we can assume that if we restrict to the emails that contain the word ‘lottery’, 10% of *those* emails also contain the word ‘sale’. What we did here was multiply the probabilities, namely, multiplying 20% by 10% to obtain 2% (or in other words,  $2/10 * 1/10 = 2/100$ ). Most likely, this is not true. The appearance of one word can sometimes heavily influence the appearance of another. For example, if an email contains the word ‘peanut’, then the word ‘butter’ is more likely to appear in this email, since many times they go together. This is why our assumption is naive. However, it turns out in practice that this assumption works very well, and it simplifies our math a lot. This is called the product rule for probabilities.

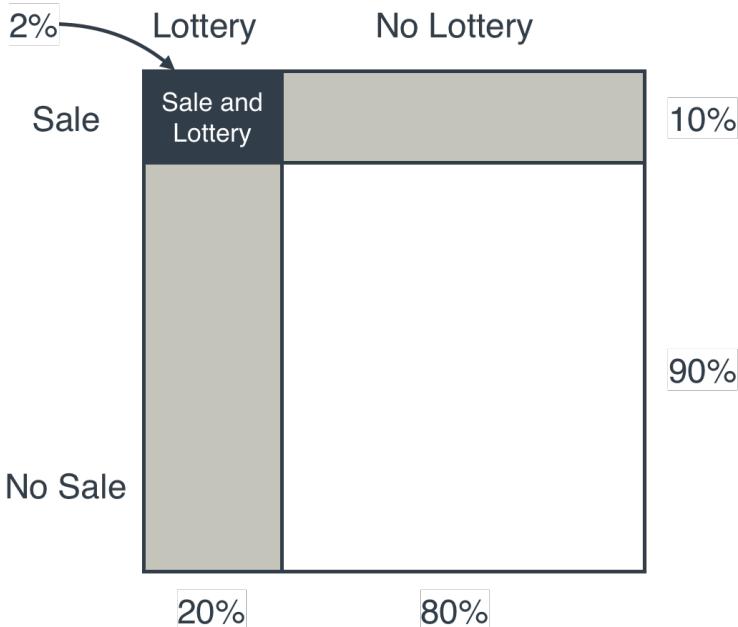


Figure 6.12: Say 20% of the emails contain the word 'lottery', and 10% of the emails contain the word 'sale'. We make the naive assumption that these two words are independent of each other. Under this assumption, the percentage of emails that contain both words can be estimated as 2%, namely, the product of 20% and 10%.

**PRODUCT RULE FOR INDEPENDENT PROBABILITIES** If two events  $E$  and  $F$  are independent, namely, the occurrence of one doesn't influence in any way the occurrence of the other one, then the probability of both of them happening is the product of the probabilities of each of the events. In other words,

$$P(E \cap F) = P(E) \cdot P(F).$$

So now, making this assumption, we can estimate how many spam and ham emails would contain the words 'lottery' and 'sale'. Let's remember the following numbers from before:

**Spam emails:**

- Among the 20 spam emails, 15 of them contained the word 'lottery'. Therefore, the probability of the word 'lottery' appearing in a spam email is 15/20, or 0.75.
- Among the 20 spam emails, 6 of them contained the word 'sale'. Therefore, the probability of the word 'sale' appearing in a spam email is 6/20, or 0.3.
- Therefore, the probability of both words appearing in a spam email is  $0.75 \cdot 0.3 = 0.225$ . This means, among our 20 spam emails, we expect to see  $0.225 \cdot 20 = 4.5$  of them containing both words.

This is a fractional number of emails! Why would we have a fractional number of emails? Well, since we are referring to probabilities, what we are saying when we say that among 20 spam emails, we expect to see 4.5 containing both words, is the same as saying that among 200 spam emails, we expect to see 45 containing both words. It's a matter of probabilities, not about counting actual emails. In probability terms, our assumption boils down to the following:

- $P(\text{lottery} \mid \text{spam}) = \frac{3}{4}$
- $P(\text{sale} \mid \text{spam}) = \frac{3}{10}$
- $P(\text{lottery} \cap \text{sale} \mid \text{spam}) = \frac{3}{4} \cdot \frac{3}{10} = \frac{9}{40} = 0.225.$

#### **Ham emails:**

- Among the 80 ham emails, 5 of them contained the word 'lottery'. Therefore, the probability of the word 'lottery' appearing in a ham email is 5/80, or 0.0625.
- Among the 80 ham emails, 4 of them contained the word 'sale'. Therefore, the probability of the word 'sale' appearing in a ham email is 4/80, or 0.05.
- Therefore, the probability of both words appearing in a ham email is  $0.0625 \cdot 0.05 = 0.003125$  (quite unlikely!). This means, among our 80 ham emails, we expect to see  $20 \cdot 0.003125 = 0.0625$  of them containing both words.

In probability terms, our assumption is the following:

- $P(\text{lottery} \mid \text{ham}) = \frac{5}{80} = \frac{1}{16}$
- $P(\text{sale} \mid \text{ham}) = \frac{4}{80} = \frac{1}{20}$
- $P(\text{lottery} \cap \text{sale} \mid \text{ham}) = \frac{1}{16} \cdot \frac{1}{20} = \frac{1}{320} = 0.0625.$

What does this mean? Well, it means that if we only restrict to emails that contain both words 'lottery' and 'sale', we have 4.5 of them that are spam, and 0.0625 that are ham. Thus, if we were to pick one at random among these, what is the probability that we pick a spam one? This may look harder with non-integers than with integers, but if we look at Figure 6.13, this may be more clear. We have 4.5 spam emails and 0.0625 ham emails (this is exactly one 16th of an email). We throw a dart and it falls in one of the emails, what's the probability that it landed on a spam email? Well, the total number of emails (or the total area, if you'd like to imagine it that way) is  $4.5 + 0.0625 = 4.5625$ . Since 4.5 are spam, then the probability that the dart landed on spam is  $4.5 / 4.5625 = 0.9863$ . This means, an email with the words 'lottery' and 'sale' has a 98.63% probability of being spam. That is quite high!



Figure 6.13: We have 4.5 spam emails, and 0.0625 ham emails. We throw a dart, and it hits one of the emails. What is the probability that it hit a spam email? The answer is 98.63%.

What we did here, using probability, was using Bayes Theorem, except with the events

- $E = \text{lottery} \cap \text{sale}$
- $F = \text{spam}$ ,

to get the formula

$$P(\text{spam} \mid \text{lottery} \cap \text{sale}) = \frac{P(\text{lottery} \cap \text{sale} \mid \text{spam}) \cdot P(\text{spam})}{P(\text{lottery} \cap \text{sale} \mid \text{spam}) \cdot P(\text{spam}) + P(\text{lottery} \cap \text{sale} \mid \text{ham}) \cdot P(\text{ham})}.$$

Then we (naively) assumed that the events of containing the words ‘lottery’ and ‘sale’ were independent, to get the following two formulas

$$P(\text{lottery} \cap \text{sale} \mid \text{spam}) = P(\text{lottery} \mid \text{spam}) \cdot P(\text{sale} \mid \text{spam}),$$

$$P(\text{lottery} \cap \text{sale} \mid \text{ham}) = P(\text{lottery} \mid \text{ham}) \cdot P(\text{sale} \mid \text{ham}).$$

Plugging them into the previous formula, we get

$$P(\text{spam} \mid \text{lottery} \cap \text{sale}) = \frac{P(\text{lottery} \mid \text{spam}) \cdot P(\text{sale} \mid \text{spam}) \cdot P(\text{spam})}{P(\text{lottery} \mid \text{spam}) \cdot P(\text{sale} \mid \text{spam}) \cdot P(\text{spam}) + P(\text{lottery} \mid \text{ham}) \cdot P(\text{sale} \mid \text{ham}) \cdot P(\text{ham})}.$$

Finally, plugging in the values

- $P(\text{lottery} \mid \text{spam}) = \frac{3}{4}$
- $P(\text{sale} \mid \text{spam}) = \frac{3}{10}$
- $P(\text{spam}) = \frac{1}{5}$
- $P(\text{lottery} \mid \text{ham}) = \frac{1}{16}$
- $P(\text{sale} \mid \text{ham}) = \frac{1}{20}$
- $P(\text{ham}) = \frac{4}{5}$

we get

$$P(\text{spam} \mid \text{lottery} \cap \text{sale}) = \frac{\frac{3}{4} \cdot \frac{3}{10} \cdot \frac{1}{5}}{\frac{3}{4} \cdot \frac{3}{10} \cdot \frac{1}{5} + \frac{1}{16} \cdot \frac{1}{20} \cdot \frac{4}{5}} = 0.9863.$$

### 6.2.5 What about more than two words?

Ok, we have discovered that if an email contains the words 'lottery' and 'sale', then its probability of being spam is 98.63%. What if we add a third word into our model? Say, the word 'mom'. We look at the data and discover that this word is not so much associated with spam, it actually appears more in ham emails. In fact, out of the 20 spam emails, only one of them contain the word 'mom', and out of the 80 ham emails, 40 of them contain it. We can do the same trick again, let's look at some probabilities:

#### Among the spam emails:

- The probability of an email containing 'lottery' is 15/20.
- The probability of an email containing 'sale' is 6/20.
- The probability of an email containing 'mom' is 1/20.

Therefore, the probability of an email containing the three words is the product of these three probabilities, namely,  $90/8000 = 0.01125$ .

#### Among the ham emails:

- The probability of an email containing 'lottery' is 5/80.
- The probability of an email containing 'sale' is 4/80.
- The probability of an email containing 'mom' is 40/80.

Therefore, the probability of an email containing the three words is the product of these three probabilities, namely,  $800/512000 = 0.0015625$ .

Thus, among 100 emails, we'd expect to find 0.01125 spam emails containing the three words, and 0.0015625 ham emails containing them. If we were to pick at random from these two, the probability that the email is spam is  $0.01125/(0.01125+0.0015625) = 0.878$ , or

87.8%. This makes sense, since even though the words 'lottery' and 'sale' make it very likely that the email is spam, the word 'mom' makes it less likely that it is.

### 6.3 Building a spam detection model with real data

Ok, now that we have developed the algorithm, let's roll up our sleeves and work with a real email dataset. Kaggle has a very good spam/ham email dataset which can be found at <https://www.kaggle.com/karthickveerakumar/spam-filter/downloads/emails.csv/1>. If you like coding, you can find this example worked out in our Github repo at [www.github.com/luisquiserrano/manning](https://www.github.com/luisquiserrano/manning).

In the repo, we have processed the dataset as a Pandas DataFrame with the following command:

```
import pandas  
emails = pandas.read_csv('emails.csv')
```

If we look at the first 10 rows of this dataset, this is how it looks (Table 6.1):

**Table 6.1.** The first 10 rows of our email dataset. The ‘text’ column shows the text in each email, and the ‘spam’ column shows a ‘1’ if the email is spam, and a ‘0’ if the email is ham. Notice that the first 10 emails are all spam.

		text	spam
0	Subject: naturally irresistible your corporate...		1
1	Subject: the stock trading gunslinger fanny i...		1
2	Subject: unbelievable new homes made easy im ...		1
3	Subject: 4 color printing special request add...		1
4	Subject: do not have money , get software cds ...		1
5	Subject: great nnews hello , welcome to medzo...		1
6	Subject: here ' s a hot play in motion homela...		1
7	Subject: save your money buy getting this thin...		1
8	Subject: undeliverable : home based business f...		1
9	Subject: save your money buy getting this thin...		1

This dataset has two columns. The first column is the text of the email (together with its subject line), in string format. The second column tells us if the email is spam (1) or ham (0). First we need to do some data preprocessing.

### 6.3.1 Data preprocessing

Let’s start by turning the text string into a list of words. This is done in the following function, which uses the `split()` function. Since we only check if each word appears in the email or not, regardless of how many times it appears, we turn it into a set, and then into a list again.

```
def process_email(text):
    return list(set(text.split()))
```

Now we use the `apply()` function to apply this change to the entire column. We call the new column `emails['words']`.

```
emails['words'] = emails['text'].apply(process_email)
```

Our email dataset now looks as follows (Table 6.2):

**Table 6.2. The email dataset with a new column called ‘words’, which contains a list of the words in the email and subject line.**

	text	spam	words
0	Subject: naturally irresistible your corporate...	1	[all, through, portfolio, its, guaranteed, , , ...
1	Subject: the stock trading gunslinger fanny i...	1	[and, merrill, is, nameable, clockwork, libret...
2	Subject: unbelievable new homes made easy im ...	1	[pre, and, all, show, being, visit, loan, 454,...
3	Subject: 4 color printing special request add...	1	[and, golden, 5110, 626, color, ca, an, canyon...
4	Subject: do not have money , get software cds ...	1	[comedies, all, old, tragedies, be, money, is...
5	Subject: great nnews hello , welcome to medzo...	1	[va, groundsel, allusion, ag, tosher, confide,...
6	Subject: here ' s a hot play in motion homela...	1	[precise, all, chain, limited, indicating, ena...
7	Subject: save your money buy getting this thin...	1	[right, want, just, money, is, within, it, rea...
8	Subject: undeliverable : home based business f...	1	[unknown, grownups, co, telecom, is, mts, 000,...
9	Subject: save your money buy getting this thin...	1	[right, want, just, money, is, within, it, rea...

### 6.3.2 Finding the priors

Let's first find out the probability that an email is spam (the prior). For this, we calculate the number of emails that are spam, and divide it by the total number of emails. Notice that the number of emails that are spam is simply the sum of entries in the ‘spam’ column. The following line will do the job.

```
sum(emails['spam'])/len(emails)
0.2388268156424581
```

The probability that the email is spam, then, is around 24%. Thus, the probability that an email is ham is around 76%.

### 6.3.3 Finding the posteriors with Bayes theorem

Now, we need to find the probabilities that spam (and ham) emails contain a certain word. We'll do this for all words at the same time. The following function creates a dictionary called ‘model’, which records each word, together with the number of appearances of the word in spam emails, and in ham emails.

```

model = {}

for email in emails:
    for word in email['words']:
        if word not in model:
            model[word] = {'spam': 1, 'ham': 1}
        if word in model:
            if email['spam']:
                model[word]['spam'] += 1
            else:
                model[word]['ham'] += 1

```

Let's examine some rows of the dictionary:

```

model
{'woods': {'ham': 4, 'spam': 2},
 'spiders': {'ham': 1, 'spam': 3},
 'hanging': {'ham': 9, 'spam': 2}}

```

This means that for example the word 'woods' appears 4 times in spam emails, and 2 times in ham emails. Let's find out the appearances of our words 'lottery' and 'sale'.

```

model['lottery']
{'ham': 1, 'spam': 9}

model['sale']
{'ham': 42, 'spam': 39}

```

This means that if an email contains the word 'lottery', the probability of it being spam is  $9/(9+1) = 0.1$ , and if it contains the word 'sale', the probability of it being spam is  $39/(39+42) = 0.48$ . This is Bayes theorem.

### 6.3.4 Implementing the naive Bayes algorithm

But we are interested in using more than one word, so let's code the naive Bayes algorithm. Our algorithm takes as input, an email. It goes through all the words in the email, and for each word, it calculates the probabilities that a spam email contains it, and that a ham email contains it. These probabilities are easily calculated by looking up the number of spam and ham emails that contain it (from the 'model' dictionary), and dividing. Then it multiplies these probabilities (the naive assumption), and applies Bayes theorem in order to find the probability that the email is spam, given that it contains the words. The code to train the model is the following:

```

def predict(email):
    words = set(email.split())
    spams = []
    hams = []
    for word in words:      #A

```

```

if word in model:      #B
    spams.append(model[word]['spam'])      #C
    hams.append(model[word]['ham'])
prod_spams = long(np.prod(spams))          #D
prod_hams = long(np.prod(hams))
return prod_spams/(prod_spams + prod_hams)    #E

```

#A For each word in the email, extract the probability that an email is spam (and ham), given that it contains that word.

These probabilities are all saved in the dictionary called 'model'.

#B Multiply all the probabilities of the email being spam. Call this variable `prod_spams`.

#C Multiply all the probabilities of the email being ham. Call this variable `prod_hams`.

#D Normalize these two probabilities, to get them to add to one (using Bayes' theorem).

#E Return the probability that an email is spam, given that it contains the words.

And that's it! Let's test the algorithm on some emails:

```

predict_naive_bayes('hi mom how are you')
0.0013894756610580057

predict_naive_bayes('meet me at the lobby of the hotel at nine am')
0.02490194297492509

predict_naive_bayes('enter the lottery to win three million dollars')
0.38569290647197135

predict_naive_bayes('buy cheap lottery easy money now')
0.9913514898646872

```

Seems to work well. Emails like 'hi mom how are you' get a very low probability (about 1%) of being spam, while emails like 'buy cheap lottery easy money now' get a very high probability (over 99%) of being spam.

### 6.3.5 Further work

This was a quick implementation of the naive Bayes algorithm. But for larger datasets, and larger emails, I recommend that you use a package. Packages like `sklearn` offer great implementations of the naive Bayes algorithm, with many parameters that you can play with. I encourage you to explore this and other packages, and use the naive Bayes algorithm on all types of datasets!

## 6.4 Summary

- Bayes theorem is a technique widely used in probability, statistics, and machine learning.
- Bayes theorem consists in calculating a posterior probability, based on a prior probability and an event.
- The prior probability is a basic calculation of a probability, given very little information.
- Bayes theorem uses the event to make a much better estimate of the probability in

question.

- The naive Bayes algorithm is used when one wants to combine a prior probability together with several events.
- The word 'naive' comes from the fact that we are making a naive assumption, namely, that the events in question are all independent.

## 7

# *Splitting data by asking questions: Decision trees*

## This chapter covers

- What is a decision tree?
- Recommending apps using the demographic information of the users.
- Asking a series of successive questions to build a good classifier.
- Accuracy, Gini index, and Entropy, and their role in building decision trees.
- Examples of decision trees in fields such as biology and genetics.
- Coding the decision tree algorithm in Python.
- Separating points of different colors using a line.

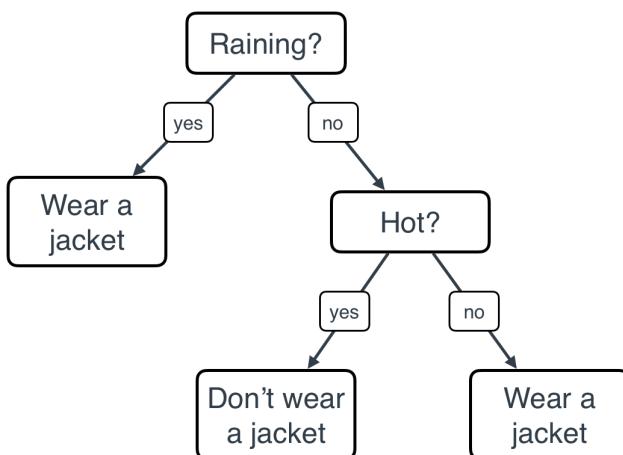
In this chapter, I cover decision trees. Decision trees are very powerful models which help us classify data and make predictions. Not only that, they also give us a great deal of information about the data. Like the models in the previous chapters, decision trees are trained with labelled data, where the labels (or targets) that we want to predict are given by a set of classes. The classes can be yes/no, spam/ham, dog/cat/bird, or anything that we wish to predict based on the features. Decision trees are a very intuitive way to build classifiers, and one that really resembles human reasoning.

In this chapter you will learn how to build decision trees that fit your data. The algorithm that we use to build decision trees is quite intuitive, and in a nutshell, it works the following way:

1. Picks the most determining feature, and divides the data according to that feature. In other words, among all the features in the data, picks the one that best divides the labels.

- Either makes the prediction based on the answer to that question, or picks another feature (the next most determining), and iterates.

This may sound complicated, but it is not, I promise. Consider the following scenario. You want to decide if you should wear a jacket today. What does the decision process look like? You may look outside and check if it's raining. If it's raining, then you definitely wear a jacket. If it's not, then it could be that it's hot outside, or cold. So then you check the temperature, and if it is hot, then you don't wear a jacket, but if it is cold, then you wear a jacket. In Figure 7.1 we can see a graph of your decision process.



**Figure 7.1.** A decision tree we use to decide if we want to wear a jacket on a given day.

As you can see, our decision process looks like a tree, except upside down. On the very top you can see the tree stump (which we call the *root*), from which two branches emanate. We call this a binary tree. Each branch leads to a new tree stump (which we call a *node*), from which it again splits in two. At every node there is a yes/no question. The two branches coming out of the node correspond to the two possible answers (yes or no) to this question. As you can see, the tree doesn't go on forever, and there are nodes where the tree simply stops branching out. We call these *leaves*. This arrangement of nodes, leaves, and edges is what we call a decision tree. Trees are very natural objects in computer science, since computers can be broken down to a huge number of on and off switches, which is why everything in a computer is binary.

**DECISION TREE** A classification model based on yes/no questions and represented by a binary tree. The tree has a root, nodes, and leaves.

**ROOT** The topmost node of the tree. It contains the first yes/no question.

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

**LEAF** When we reach a point where we don't ask a question and instead we make a decision, we have reached a leaf of the tree.

Figure 7.2 shows how a decision tree looks like in general.

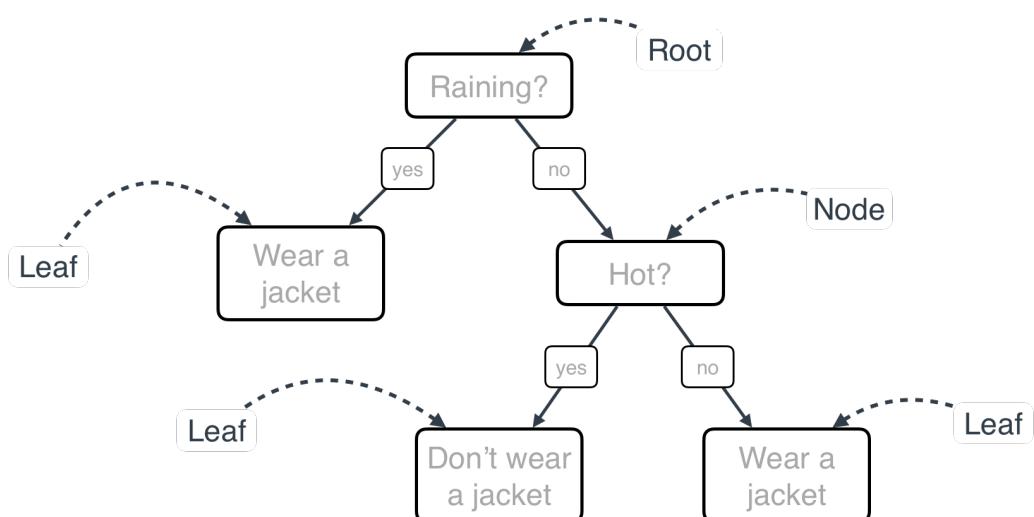


Figure 7.2. A regular decision tree with a root, nodes, and leaves. Note that each node contains a yes/no question. From each possible answer, one branch emanates, which leads to another node, or a leaf.

But how did we get to build that 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:

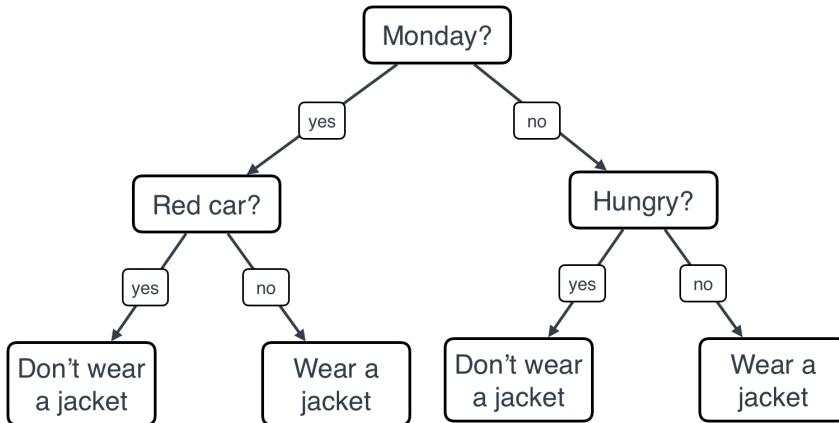


Figure 7.3. A second (maybe not as good) decision tree we could use to decide if we want to wear a jacket on a given day.

Which tree do we think is better, tree 1 (the one in Figure 7.1), or tree 2 (the one in Figure 7.3)? Well, as humans, we have enough experience to figure out that tree 1 is much better than tree 2. Now the question is, how would a computer know? Computers don't have experience per se, but they have something very similar, which is data. So 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, while if we used tree 2, we may have ended up freezing on a cold day without a jacket, or wearing a jacket in extreme heat. So 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, medical diagnosis, and much more. In this chapter, we'll go over an algorithm for constructing good decision trees quickly. In a nutshell, 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 this first question be? 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 hot outside?
3. Am I hungry?
4. Is there a red car outside?

### 5. Is it Monday?

Let's think, out of these five questions, which one seems like the best one to ask if our goal is to determine when we should wear a jacket or not. I'll say that the last three questions are pretty irrelevant when it comes to wearing a jacket or not. We should decide between the first two. In this chapter, we learn how to pick the right question based on data, so we'll get into details later. But for now, imagine that for an entire year, we decide if we wear a jacket or not based on if it's raining or not, and for another entire year, we decide if we wear a jacket or not based on if it was hot outside or not. We check how many times we were correct, namely, how many times we wore a jacket when we should have, and not wore one when we shouldn't have, and we see that in the first year, we were correct 250 times, and in the second year we were correct only 220 times. Therefore, the first question is better, based on our data. We'll go for that one. Therefore, we have a very simple decision tree, consisting of only one question. The tree is illustrated in Figure 7.4.

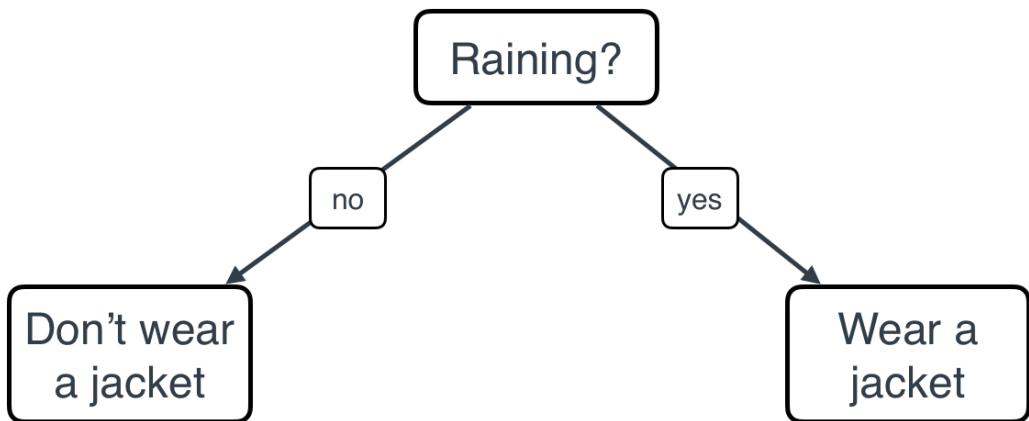
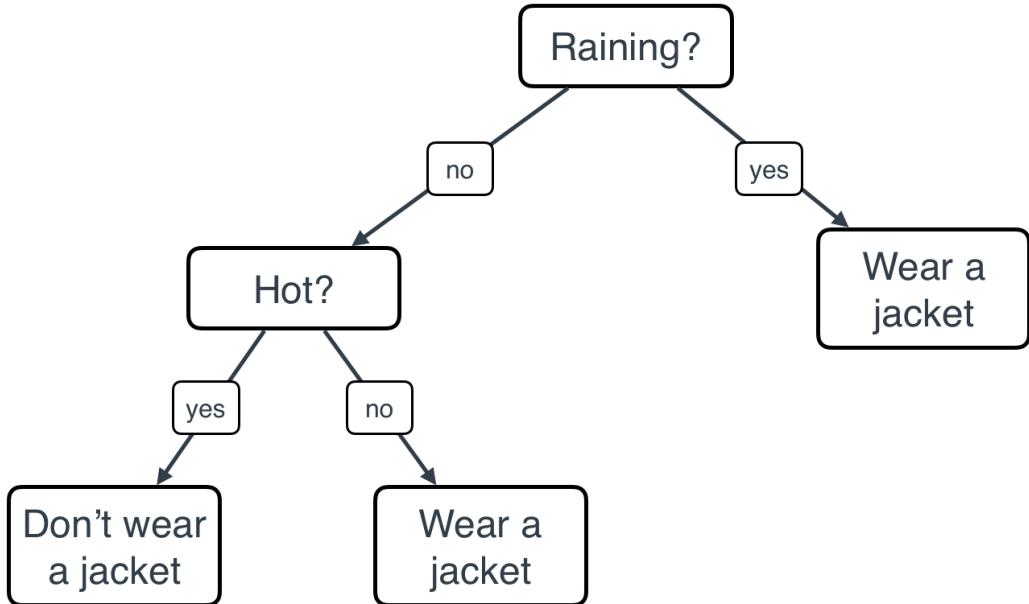


Figure 7.4. A very simple decision tree which consists of only the question “Is it raining?”.

Now, can we do better? Let's say we noticed that the right part of the tree is very accurate, namely, that whenever it's raining, and we wear a jacket, we made the correct decision. But the left part of the tree is not so correct, since there were many times when it wasn't raining, and we decided to not wear a jacket, but we were wrong. Maybe it was very cold, or maybe it was snowing! So what can we do? Well, now question 2 can come to our rescue. 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 in Figure 7.5.



**Figure 7.5.** A slightly more complicated decision tree than the one in Figure 7.4, where we have picked one leaf, and split it into a further two leaves.

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

Many questions may arise in your head, for example:

1. How exactly do you decide which is the best possible question to ask?
2. Does this process actually gets us to build *the* very best decision tree? What if I start in a different way, by not picking the best question every time, and I get to a better tree?
3. How is this process any faster than going through all possible decision trees?
4. How do we code this algorithm?
5. Where in real life can we find decision trees?

This chapter answers all of these questions, but let me give you some quick answers:

**1. How exactly do you decide which is the best possible question to ask?**

There are several ways to do this. The simplest one is using accuracy, which means, which question helps me be correct more often. However, others such as Gini index or entropy will be used in this book as well.

**2. Does this process actually gets us to build *the very best decision tree? What if I start in a different way, by not picking the best question every time, and I get to a better 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 move at that moment. They tend to work well, but it's not always the case that making the best possible move gets you to the best 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. I challenge you to 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. How is this process any faster than going through all possible decision trees?**

The number of possible decision trees is very large, specially if our dataset has many features. Going through all of them would be very slow. In here, finding each node only requires a search across the features, and not across all the possible trees, so it is much faster.

**4. How do we code this algorithm?**

Yep! By the end of this chapter we'll code most elements of this algorithm. However, you'll see that since the algorithm is recursive, the coding can get a bit tedious. Thus, we'll use a very useful package called scikit learn to build decision trees with real data.

**5. Where in real life can we find decision trees?**

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 is in recommendation systems (to recommend videos, movies, apps, products to buy, etc.), in spam classification (to decide if an email is spam or not), in sentiment analysis (to decide if a sentence is happy or sad), and in biology (to decide if a patient is sick or not, or to help identify certain hierarchies in species or in types of genomes).

All these will be answered in this chapter, and more! Let's begin by introducing a very popular application in machine learning: recommendation systems.

## 7.1 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. Have you wondered how Netflix recommends you movies, YouTube guesses which videos you may watch, or Amazon recommends you products you might be interested in buying? These are all uses of recommendation systems. One very simple and interesting way to see

recommendation problems is to see them as classification problems. Let's start with an easy example, we'll develop our very own app recommendation system using decision trees.

Let's say we are the App Store, or Google Play, and we want to recommend to our users the app that they are most likely to download. We have three apps in our store:

- 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.



Atom count



Beehive Finder



Check Mate Mate

App for counting the number of atoms in your body

App for locating the nearest beehives to your location

App for finding Australian chess players in your area

**Figure 7.6.** The three apps in our store. 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.

We look at a table of previous data, where we have recorded the gender and age of our users, and the app that they have downloaded. For simplicity, we will use a table of 6 people (Table 7.1)

**Table 7.1.** A customers dataset with their age and gender, and the app they downloaded.

Gender	Age	App
F	15	 Atom Count
F	25	 Check Mate Mate

M	32	 Beehive Finder
F	35	 Check Mate Mate
M	12	 Atom Count
M	14	 Atom Count

Now, the following customers start to log into the store, so let's recommend some apps to them. What would you recommend to each of the following customers?

**CUSTOMER 1: A GIRL AGED 13.** To this customer, I would recommend Atom Count, and the reason is because it seems (looking at the three customers on their teens) that young people tend to download Atom Count.

**CUSTOMER 2: A WOMAN AGED 27.** To this customer, I would recommend Check Mate Mate, since looking at the two adult women in the dataset (aged 25 and 35), they both downloaded Check Mate Mate.

**CUSTOMER 3: A MAN AGED 34.** To this customer, I would recommend Beehive Finder, since there is one man in the dataset who is 32 years old, and he downloaded Beehive Finder.

You may have had the same or different recommendations than me, and that is ok, as long as you have a justification for the recommendation. 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.

## 7.2 The solution: Building an app recommendation system

In this section I show you how to build an app recommendation system using decision trees. In a nutshell, what we'll do is decide which of the two features (gender or age) is more successful

at determining which app will the users download, and we'll pick this one as our root of the decision tree. Then, we'll iterate over the branches, always picking the most determining feature for the data in that branch, thus building our decision tree.

### 7.2.1 The remember-formulate-predict framework

As we do for most algorithms in this book, we'll build the remember-formulate-predict framework which we learned in Chapter 1. This one works as follows:

1. **Remember:** Look at previous data (previous customers, their age, gender, and what apps they have downloaded).
2. **Formulate:** Formulate a rule that tells us if which app a new customer is most likely to download.
3. **Predict:** When a new customer comes in, we use the rule to guess what app they are most likely to download, and we recommend them that app.

In this case, the model will be a decision tree. Let's build it step by step.

### 7.2.2 First step to build the model: Asking the best question

We are now focused on step 2, formulate a model. First, let's simplify our data a little bit, let's call everyone under 20 years old 'young', and everyone 20 or older 'adult'. Our table now looks like Table 7.2:

**Table 7.2. Our dataset, with the age column simplified to two categories, 'young' and 'adult'.**

Gender	Age	App
F	young	 Atom Count
F	adult	 Check Mate Mate
M	adult	 Beehive Finder

F	adult	 Check Mate Mate
M	young	 Atom Count
M	young	 Atom Count

The building blocks of decision trees are questions of the form “Is the user female?” 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 determines the app they downloaded best. In order to decide which question is better at this, let’s compare them.

#### **FIRST QUESTION: IS THE USER FEMALE OR MALE?**

This question splits the users into two groups, the females and the males. Each group has 3 users in it. But we need to keep track of what each user downloaded. A quick look at Table 7.2 helps us notice that:

- Out of the females, one downloaded Atom Count and two downloaded Check Mate Mate.
- Out of the males, two downloaded Atom Count and one downloaded Beehive Finder.

The resulting node is drawn in Figure 7.7.

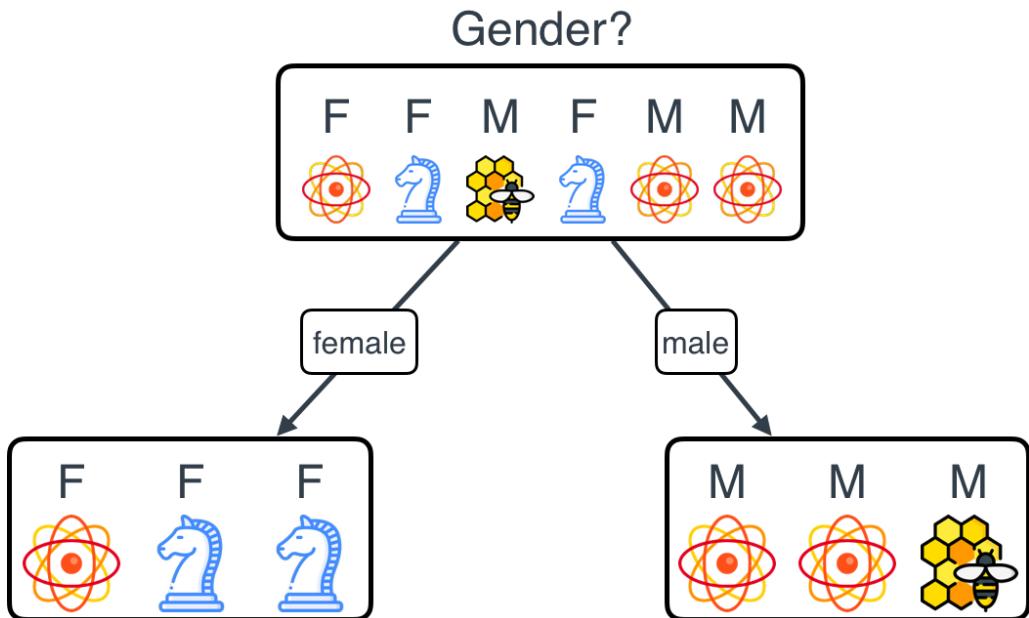


Figure 7.7. If we split our users by gender, we get this split. The females are at the left, and the males at the right. Out of the females, one downloaded Atom Count and two downloaded Check Mate Mate. Out of the males, two downloaded Atom Count and one downloaded Beehive Finder.

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 3 users in it. A quick look at Table 7.2 helps us notice what each user downloaded:

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

The resulting node is drawn in Figure 7.8.

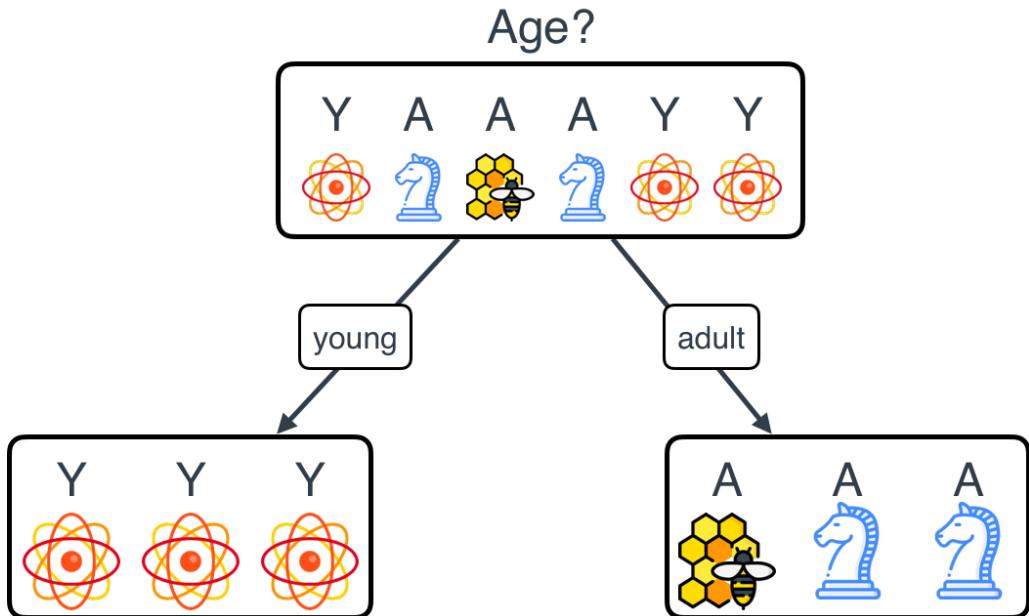


Figure 7.8. If we split our users by age, we get this split. The young are at the left, and the adults at the right. Out of the young users, all three downloaded Atom Count. Out of the adult users, one downloaded Beehive Finder and two downloaded Check Mate Mate.

From looking at figures 5.8 and 5.9, which one looks like a better split? To me it seems that age is better, since 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 need to give it some numbers to compare. Later in this chapter we learn three ways to attach a number to each of the features, in order to compare them. These are called accuracy, Gini impurity, and information gain. Let's start with the first one: accuracy.

### ACCURACY

Suppose that we are only allowed one question, and with that one question we must determine what app to recommend to our users. So we have two classifiers:

- **Classifier 1:** Asks the question ‘what is your gender?’, 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.

**Classifier 1:** If the answer to the question is ‘female’, then we notice that out of the females, the majority downloaded

From the previous reasoning, we have the hunch Classifier 1 is better than Classifier 2, but let's compare the two of them, based on accuracy. In other words, let's

The reason is that once we split by gender, we still have both apps on each of the sides, but if we split by age, we can see that every young person downloaded Atom Count. Therefore, if all we can do is ask one question, we're better off asking "What is your age?", than "What is your gender?". This last argument may sound a bit vague, but let me give you a convincing statement. If we were to only use one question for our classifier, and then use that to recommend products, what would we recommend? It makes sense to recommend the most popular app in a particular group, right? If, for example, two out of the three adults downloaded Check Mate Mate, it makes sense to recommend Check Mate Mate to every adult, and be correct 2 out of 3 times, as opposed to recommending Instagram and being correct 1 out of 3 times. If we do that with both trees, we can see the comparison of how many times we are correct in Figure 7.10.

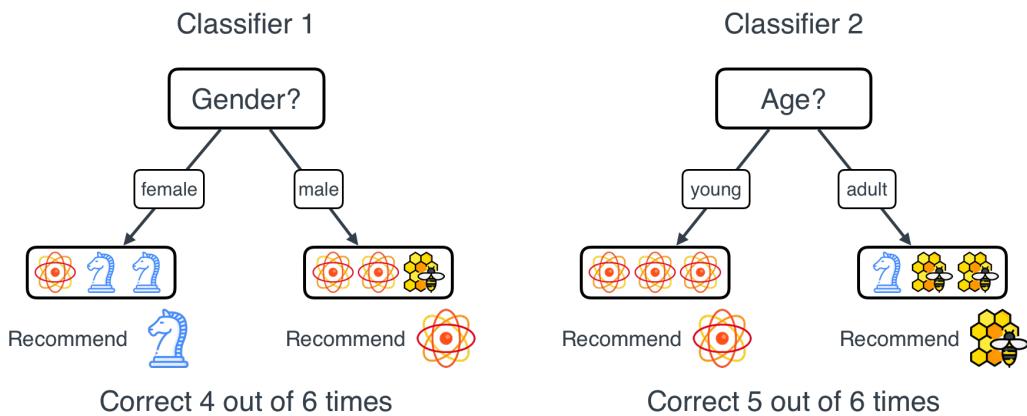


Figure 7.10. Tree 1 is formed by asking the gender, and tree 2 is formed by asking the age. Tree 1 is correct 4 out of 6 times, and tree 2 is correct 5 out of 6 times. Therefore we can say that tree 2 is better.

There are actually many ways to see that tree 2 is better than tree 1. Later in section 5.3, we will revisit accuracy, and introduce two new ways to compare the trees.

### 7.2.3 Next and final step: Iterate by asking the best question every time

Let's continue building our tree. We have decided that if a person is young, we'll recommend them Atom Count. This means at that point we stop. As we saw before, this is called a leaf.

If they are adults, however, we can recommend them Check Mate Mate, but we can do better. Let's look at the original table, reduced to only the adults (Table 7.3).

**Table 7.3.** Our dataset, restricted to only adults.

Gender	Age	App
F	adult	 Beehive Finder
M	adult	 Check Mate Mate
F	adult	 Beehive Finder

What do we see here? Well, notice that if we now divide the users by gender, we get that the males (only one male) downloaded Instagram, while the females downloaded Check Mate Mate. Therefore, we can just add this branch to our tree, to obtain the one in Figure 7.11.

## Final tree

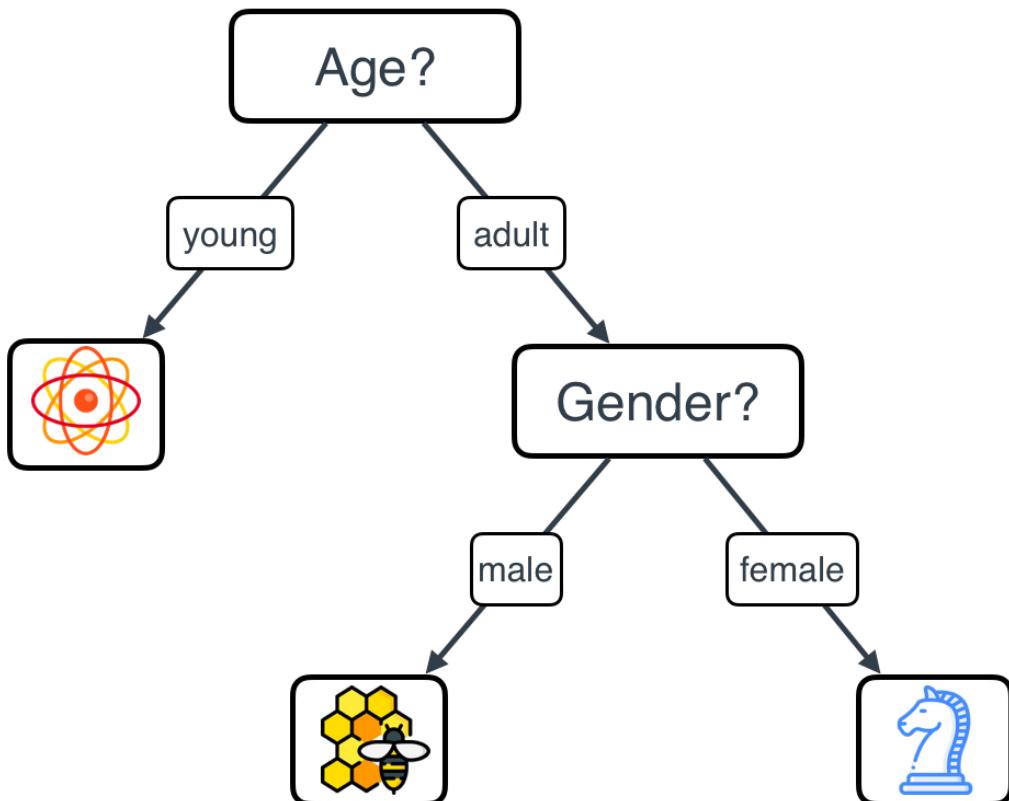


Figure 7.11. Our final decision tree. The prediction goes as follows. The tree makes a prediction as follows: First it checks if the user is young. If they are young, it recommends them Atom Count. If not, then it checks if the user is male or female. If the user is female, it recommends Check Mate Mate, and if the user is male, it recommends them Beehive Finder.

And we're done! This is our decision tree. Here is the pseudocode for the predictions that this model makes.

```

predict(user):
    if user is young:
        recommend Atom Count
    else:
        if user is female:
            recommend Check Mate Mate
        else:
            recommend Beehive Finder
  
```

### 7.2.4 Using the model by making predictions

Now that we have a tree, let's put it in practice by making predictions. This is the simplest part. Let's go back to our three users from the beginning:

**CUSTOMER 1: A GIRL AGED 13.** Based on the age question, since she is young, we recommend her Atom Count.

**CUSTOMER 2: A WOMAN AGED 27.** First we ask the age question, and since she is an adult, then we ask the gender question. Since she is a female, we recommend her Check Mate Mate.

**CUSTOMER 3: A MAN AGED 34.** Again we ask the age question, and since he is an adult, we ask the gender question. Since he is a male, we recommend him Instagram.

Notice that this matches our intuition from the beginning of the chapter. That means our decision tree worked!

Now, there was one little step here that we didn't elaborate much on. It's the step where we decided that age was a better split than gender. Let's study this one in more detail with a slightly different example.

## 7.3 Building the tree: How to pick the right feature to split

In the previous section, we picked the 'age' split against the 'gender' split, because age correctly classified five out of the six users, while gender correctly classified four. But there are actually many ways to select between the two splits. Let's modify our dataset a little bit. Our new dataset has now three features, gender, age, and location. The location is where each user lives, and it can only be one of two cities: city A and city B. Furthermore, there are only two apps to download, instead of three, and their names are App 1 and App 2. The table looks as follows (Table 7.4):

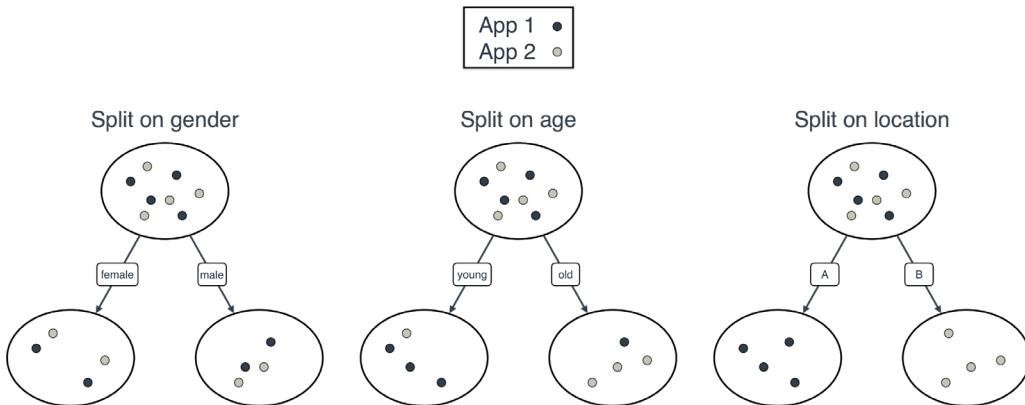
**Table 7.4. A new app recommendation table. It has three features: Gender, Age, and Location. The users have downloaded one of two apps: App1, and App 2.**

Gender	Age	Location	App
Female	Young	A	1
Male	Young	A	1
Female	Young	A	1
Male	Adult	A	1

Female	Young	B	2
Male	Adult	B	2
Female	Adult	B	2
Male	Adult	B	2

Note that we have three possible splits, gender, age, and location. We'll study each one separately.

- If we split on gender, we have four females and four males. Among the four females, two downloaded App 1, and two downloaded App 2. Same for the males.
- If we split on age, we have four young users and four adult users. Among the four young ones, three downloaded App 1, and one downloaded App 2. Among the adults, one downloaded App 1, and three downloaded App 2.
- If we split on location, we have four users who live on City A, and four on City B. The four that live on City A downloaded App 1, and the four that live on City B downloaded App 2.



**Figure 7.12.** Three different splits of our new app dataset. The first split is by gender, the second one by age, and the third one by location. The black points are users who downloaded App 1, and the light grey points are users who downloaded App 2.

This three splits are shown graphically in Figure 7.12. From looking at the graphs, it looks clear that the best split is on location, since it completely determines who downloaded App 1 and App 2. Here is a question, which split is better between the split by gender and age? From first glance, it seems that age is better, since it manages to put most of the users who downloaded App 1 on one side, and most of the users who downloaded App 2 on the other side. The split by

gender leaves us with two groups where two users downloaded App 1 and two users downloaded App 2, which gives us very little information. So for us, it seems that the order is: Location is best, age is next, and gender is the worst. However, computers need numbers, so we need a way to quantify how good these three splits are. We'll learn three ways to quantify how good these are: Accuracy (which we briefly used before), Gini impurity, and information gain. Let's study them one by one.

### 7.3.1 How to pick the best feature to split our data: Accuracy

The first way to split the points is based on accuracy. We briefed this in the previous section, but let's study it in more detail. The question we'd like to ask here is, "if my classifier only consisted on asking one question, what would be the best question to ask? Each question gives rise to a particular classifier, as follows.

#### **Classifier 1:** What is your gender?

The idea is to have a classifier that says "If you are a female, you downloaded App X, and if you are a male, you downloaded App Y". Since among the four females, two downloaded App 1, and two downloaded App 2, it doesn't make any difference to pick App 1 or 2. With the males we have the exact same situation. Therefore, we can decide that our classifier predicts App 1 for the females, and App 2 for the males.

Now the question is, what is the accuracy of this classifier? Among our data, the classifier was correct four times, and wrong four times. Therefore, the accuracy of this classifier is  $\frac{1}{2}$ , or 50%.

#### **Classifier 2:** What is your age?

Now, we'd like to build a classifier that says "If you are young, you downloaded App X, and if you are an adult, you downloaded App Y". Let's see what is the best classifier that we can build using this question. Notice that among the four young users, three downloaded App 1, and one downloaded App 2. Therefore, the best prediction we can make here, is that every young user downloaded App 1. Among the adults, the opposite happens, since one of them downloaded App 1, but three downloaded App 2. Therefore, the best prediction we can make is that every adult user downloaded App 2.

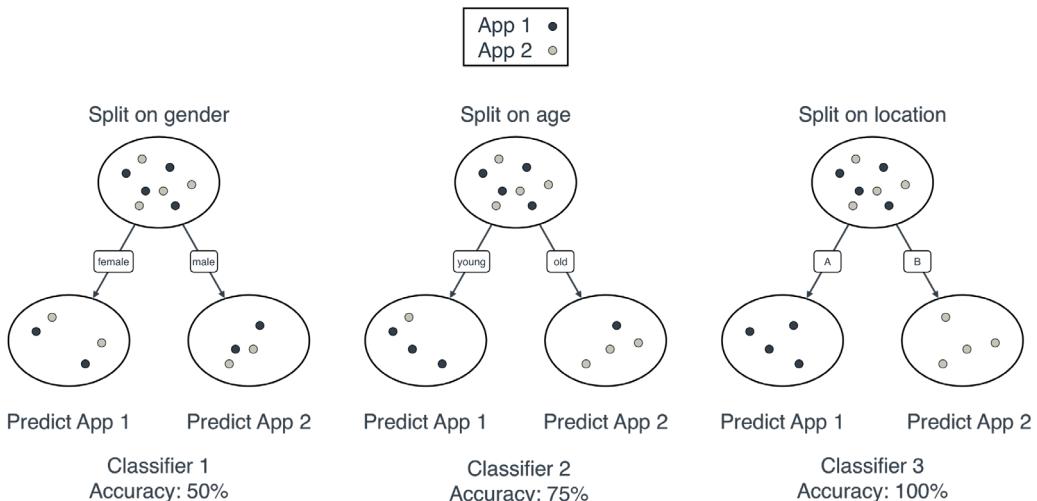
And again, the question is, what is the accuracy of this classifier? Among our data, the classifier was correct six times, and wrong two times. Therefore, the accuracy of this classifier is  $\frac{3}{4}$ , or 75%.

#### **Classifier 3:** What is your location?

Finally, we'd like to build a classifier that says "If you live in City A, you downloaded App X, and if you live in City B, you downloaded App Y". Notice that all the people in City A downloaded App 1, and all the people in City B downloaded App 2. Therefore, the best prediction we can make here, is that every person in City A downloaded App 1, and every person in City B downloaded App 2.

What is the accuracy of this classifier? Well, among our data, it is correct eight times out of eight. Therefore its accuracy is 100%.

The three classifiers, together with their accuracies, are shown in Figure 7.13.



**Figure 7.13.** Each feature gives rise to a simple classifier that only uses that feature to separate the users. The first classifier, based on gender, has an accuracy of 50%. The second one, based on age, has an accuracy of 75%. The last one, based on location, has an accuracy of 100%.

From these three classifiers, we can see that Classifier 3 is the best, as its accuracy is 100%. Therefore, we conclude that the best feature to split is location.

### 7.3.2 How to pick the best feature to split our data: Gini impurity

The idea behind the concept of impurity is to come up with a number that is low if a set is very homogeneous, and high if a set is very heterogeneous. In other words, if we have a set in which all the elements look the same, then the impurity index is very low, and if all the elements look different, then the impurity index is very high. In the previous section, we split our dataset into sets of four users. If we only consider what apps they downloaded, we got the following sets:

- **Set 1:** App 1, App 1, App 1, App 1.
- **Set 2:** App 1, App 1, App 1, App 2.
- **Set 3:** App 1, App 1, App 2, App 2.
- **Set 4:** App 1, App 2, App 2, App 2.
- **Set 5:** App 2, App 2, App 2, App 2.

Notice that sets 1 and 5 are completely homogeneous, since all users downloaded the same app. These two would have a very low impurity index. Set 3 is the most diverse, since it has two users downloading App 1, and two users downloading App 2. This set would have a high impurity index. Sets 1 and 4 are in between, so they would have a medium impurity index. Now the question is, how do we come up with a good formula for the impurity index? There are many ways to quantify the impurity in a set, and in this section I will show you one called the Gini impurity index.

In a nutshell, the Gini impurity index measures the diversity in a set. Let's say, for example, that we have a bag full of balls of several colors. A bag where all the balls have the same color, has a very low Gini impurity index (in fact, it is zero). A bag where all the balls have different colors has a very high Gini impurity index.



Low Gini  
impurity index



High Gini  
impurity index

**Figure 7.14.** A bag full of balls of the same color has a low Gini impurity index. A bag full of balls of different colors has a very high Gini impurity index.

Like everything in math, we need to attach a number to the Gini impurity index, and for this, we turn to our good old friend, probability. In a bag full of balls of different colors, we play the following game. We pick a ball out of this set, randomly, and we look at its color. Then we put the ball back. We proceed to randomly pick another ball from the bag (it could happen that it is the same ball, or a different one, we don't know). We look at its color. We record the two colors we obtained, and we check if they are equal, or different. Here is the main observation of Gini index:

- If the set has low Gini impurity index, then most balls are of the same color. Therefore the probability that the two balls are of different color is very low.
- If the set has high Gini impurity index, then the balls tend to be of different colors. Therefore the probability that the two balls are of different color is very high.

That probability is the Gini impurity index. In this section, we'll find a nice closed formula that calculates it.

Let's go back to the example with App 1 and App 2. For convenience, we'll think of App 1 as a black ball, and App 2, as a white ball. Our sets are then the following:

**Set 1:** {black, black, black, black}

**Set 2:** {black, black, black, white}

**Set 3:** {black, black, white, white}

**Set 4:** {black, white, white, white}

**Set 5:** {white, white, white, white}

Notice that in sets that are very pure, like Set 1 or Set 5, the probability of picking two balls of different colors are very low (0, in fact). In sets that are very impure, like Set 3, this probability is high. Let's actually calculate this probability for all the sets.

#### **Set 1:**

Since there are no white balls, the probability of picking a pair of different balls is exactly zero. To remove any shadow of doubt, Figure 7.15 shows all the 16 possible scenarios of picking the first and second balls, and none of them contain a pair in which the balls have different colors.

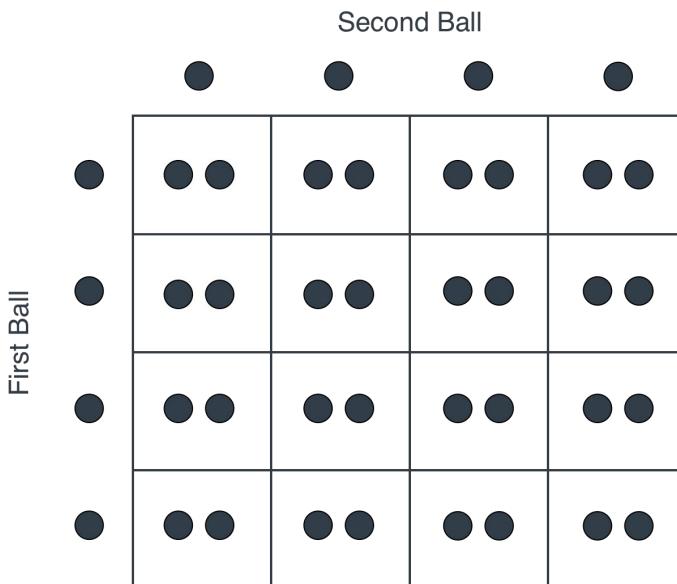
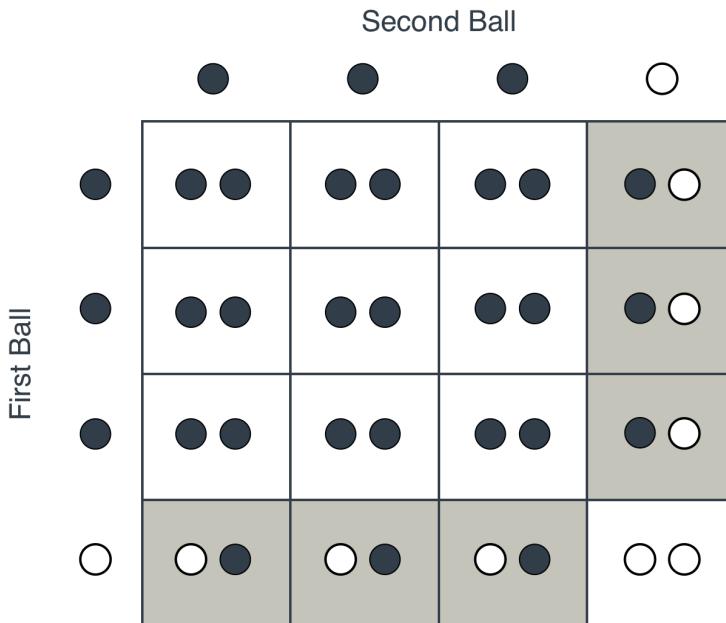


Figure 7.15. If we pick our balls from a bag containing four black balls, we get 16 scenarios. All of them give us the pair (black, black). Thus, the probability of obtaining two balls of different colors is zero.

Therefore, we define the Gini impurity of Set 1 to be 0.

### **Set 2:**

Now things start getting interesting. We need to pick either the pair (black, white), or the pair (white, black). What is the probability of picking the pair (black, white)? Since we have 4 ways of picking the first ball, and 4 ways of picking the second ball, then there are 16 possible scenarios. Figure 18 contains the all the scenarios for black and white balls, and you can see that only 6 of them produce the desired result of two balls of different colors. We've highlighted these 6 scenarios.

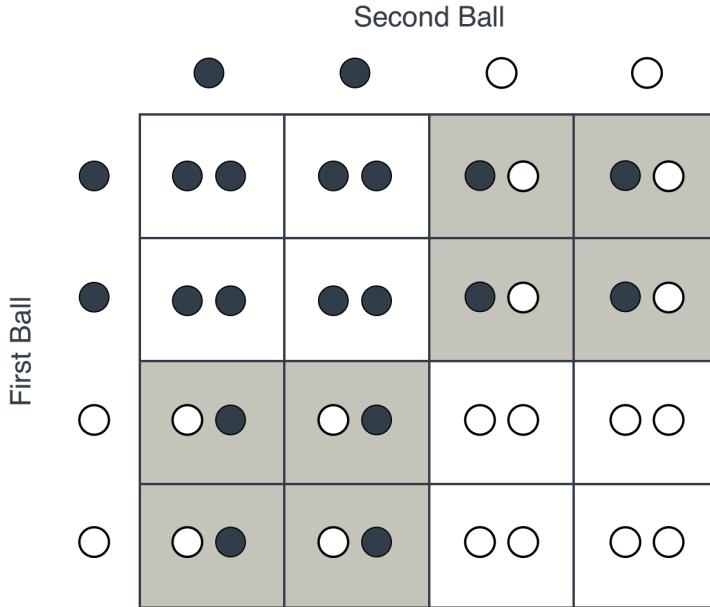


**Figure 7.** 16. If we pick our balls from a bag containing three black balls and one white ball, we again get 16 scenarios. Six of them give us a pair of balls of different colors (highlighted), and the remaining 10 give us pairs of balls of the same color (nine black-black and one white-white). Therefore, the probability of picking two balls of different colors is 6/16

Therefore, the probability of picking two balls of different colors is 6/16. We then define the Gini impurity of Set 2 to be 6/16 (or 0.375).

### **Set 3:**

Again, we have 16 possible scenarios. Figure 7.17 shows them, and you can notice that 8 of them (exactly half) give us a pair of balls of different colors.



**Figure 7.17.** If we pick our balls from a bag containing two black balls and two white balls, we again get 16 scenarios. Eight of them give us a pair of balls of different colors (highlighted), and the remaining 10 give us pairs of balls of the same color (four black-black and four white-white). Therefore, the probability of picking two balls of different colors is  $8/16$ , or  $\frac{1}{2}$ .

Therefore, the probability of picking two balls of different colors is  $8/16$ . We then define the Gini impurity of Set 2 to be  $8/16$  (or 0.5).

#### Sets 4 and 5:

Notice that Set 4 is just the mirror image of Set 1, and Set 5 is the mirror image of Set 2. Therefore, their Gini impurity index should be the same as the original one. In other words, the Gini impurity index of Set 4 is 0.375, and that of Set 5 is 0.

Going back to App 1 and App 2 and summarizing, this is what we have calculated for the Gini impurity indices of our sets.

- **Set 1:** (App 1, App 1, App 1, App 1}. Gini impurity index: 0.
- **Set 2:** (App 1, App 1, App 1, App 2}. Gini impurity index: 0.375.
- **Set 3:** (App 1, App 1, App 2, App 2}. Gini impurity index: 0.5.
- **Set 4:** (App 1, App 2, App 2, App 2}. Gini impurity index: 0.375.
- **Set 5:** (App 2, App 2, App 2, App 2}. Gini impurity index: 0.



**Figure 7.18. Our five sets with their corresponding Gini impurities. Notice that sets 1 and 5 are the most pure, and they have an index of 0. The least pure set is set 3, which has an index of 0.5. Sets 2 and 4 are in between, with indices of 0.375.**

Seems like exactly what we wanted. Set 3 has a high Gini impurity index (0.5), Sets 1 and 5 have a low one (0), and Sets 2 and 4 have a medium one (0.375).

#### FORMULA FOR GINI INDEX

Like everything in machine learning, Gini index has a formula. And the formula is rather simple. Let's calculate it, for example, for the bag with three black balls and one white ball. What is the probability that we pick a ball out of this bag, we look at it and put it back, then we pick another ball, and both balls are of different color? This is called picking a ball with repetition, since we put the first ball back after we pick it. Let's consider the complementary event, namely, that both balls are of the same color. Clearly, whatever we obtain for this last probability, should be one minus the original one, since two probabilities of complementary events must add to one (as we learned in Chapter 5). Now, what is the probability that both balls are the same color? Well, the balls could both be black, or white. The probability that a ball is black is  $\frac{3}{4}$ , so the probability that both balls are white is  $\frac{3}{4} * \frac{3}{4} = 9/16$ . Similarly, the probability that both balls are white is  $\frac{1}{4} * \frac{1}{4} = 1/16$ . We add these two to obtain the probability that the balls are of the same color, and we get  $10/16$ . The complementary probability, then, is  $1 - 10/16$ , which is  $6/16$ . This is the probability that the balls are of a different color.

Now, let's do this in general. Let's say we have  $N$  balls, that could be of  $k$  colors, 1, 2, ...,  $k$ . Say that there are  $a_i$  balls of color  $i$  (therefore,  $a_1 + a_2 + \dots + a_k = N$ ). The probability of picking a ball of color  $i$  is  $p_i = \frac{a_i}{N}$ . Therefore, the probability of picking two balls of color  $i$  is  $p_i^2$ . Thus, the probability of picking two balls of the same color is  $p_1^2 + p_2^2 + \dots + p_k^2$ , since the two balls could be of colors 1, 2, ...,  $k$ . The complement of this event is the event that we picked two balls of different color. Therefore, the probability that we picked two balls of different color (or the Gini impurity index) is

$$\begin{aligned}
 \text{Gini Impurity Index} &= P(\text{picking two balls of different color}) \\
 &= 1 - P(\text{picking two balls of the same color}) \\
 &= 1 - p_1^2 - p_2^2 - \dots - p_N^2
 \end{aligned}$$

P(Both balls are color 1)      P(Both balls are color N)  
 P(Both balls are color 2)

Figure 7.19. The Gini impurity index is the probability that if we pick two different balls (with repetition), the balls will have the same color. This is the same as 1 minus the probability that we pick two balls of the same color. Since the balls can be of colors 1 up to N, then the probability that we pick two balls of the same color is the sum of the probabilities that we obtain two balls of color i, for i from 1 to N.

Let's look at an example with more colors. Let's say we have a bag with 6 balls, 3 of them black, 2 of them gray, and 1 of them white. What is the Gini impurity index here? The probabilities of picking each color are the following:

- Picking blue:  $p_1 = \frac{3}{6} = \frac{1}{2}$ .
- Picking red:  $p_2 = \frac{2}{6} = \frac{1}{3}$ .
- Picking yellow:  $p_3 = \frac{1}{6}$ .

Thus, the Gini impurity index is precisely:

$$1 - p_1^2 - p_2^2 = 1 - \left(\frac{1}{2}\right)^2 - \left(\frac{1}{3}\right)^2 - \left(\frac{1}{6}\right)^2 = \frac{22}{36}$$



$$\text{Gini Impurity Index} = P(\text{picking two balls of different color})$$

$$\begin{aligned}
 &= 1 - \left(\frac{3}{6}\right)^2 - \left(\frac{2}{6}\right)^2 - \left(\frac{1}{6}\right)^2
 \end{aligned}$$

P(Both balls are black)      P(Both balls are grey)  
 P(Both balls are white)

Figure 7.20. The Gini impurity index of a set with three black balls, two grey balls, and one white ball. The probability of picking two balls of different color is the same as 1 minus the probability

of picking two balls of the same color. Since the balls can be black, grey, or white, we add over the three probabilities to obtain the last one.

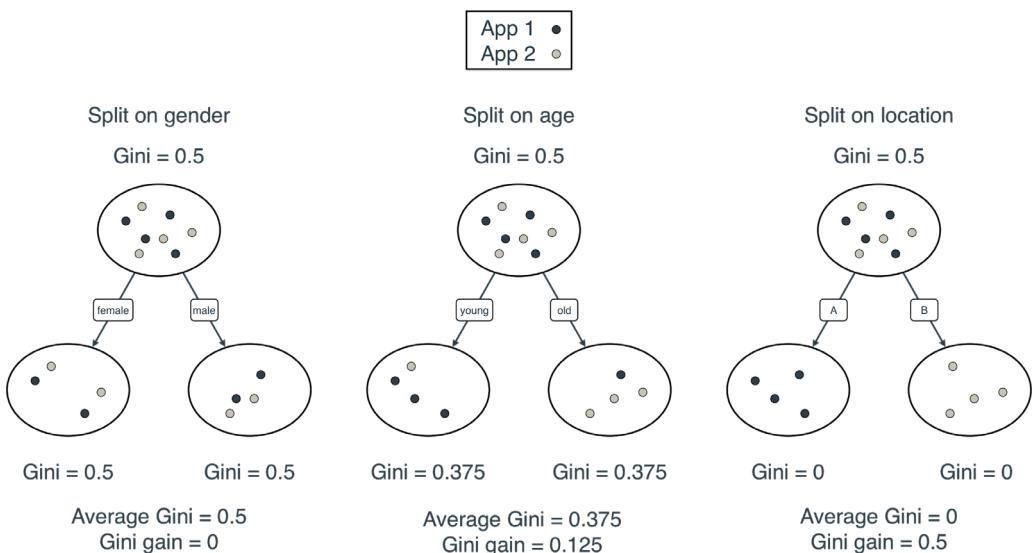
### HOW TO PICK THE BEST FEATURE? GINI GAIN

Now that we know how to calculate the Gini impurity index, we'll use this for our decision tree. All we have to do is simply calculate the Gini impurity index of the root, and of the two leaves, and see where we had the most gain. Let me be more specific.

Procedure to measure a split:

- Calculate the Gini impurity index of the root.
- Calculate the average of the Gini impurity indices of the two leaves.
- Subtract them, to obtain the gain on Gini impurity index.

Now, among all the splits, we simply pick the one with the highest gain on Gini impurity index. Figure .23 shows this calculation for our app example.

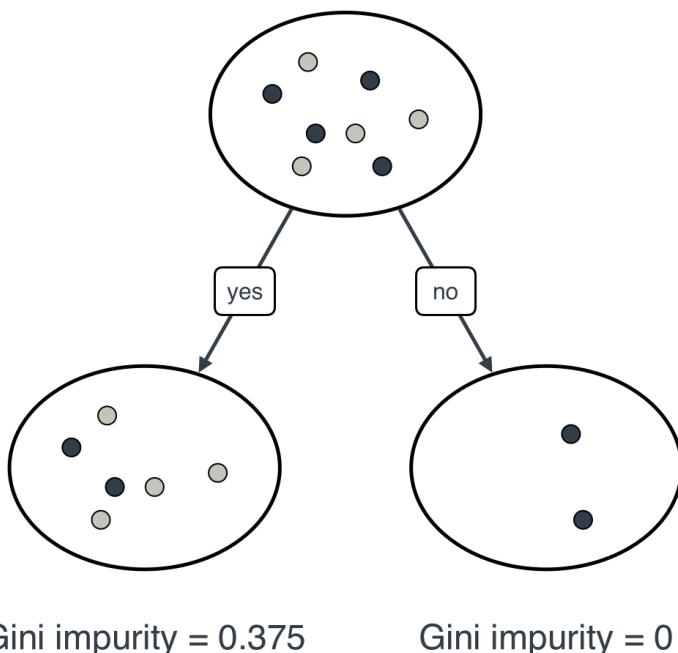


**Figure 7.21. Calculating the Gini impurity index on each one of the three splittings. The Gini impurity index of the root is always 0.5. When we split on gender, we get an average Gini impurity index of 0.5 on the leaves, giving us a gain of 0. When we split on age, the average Gini index of the leaves is 0.375, giving us a gain of 0.125. When we split on location, the average Gini index of the leaves is 0, giving us a gain of 0.5. Therefore, the largest Gini gain is obtained by splitting on location, and that is the splitting we pick.**

Now we simply select the split that gives us the smallest possible Gini impurity, which is the third one.

### WEIGHTED GINI IMPURITY

You may be asking a question, which is, "Why did we average out the Gini impurity?". In reality what we want to consider is a weighted average. This is because when we split our data based on a feature, we want the two sets to be as pure as possible, but we also want them to be balanced in terms of size. For example, if we split a set of 100 data points into one of size 99, and one of size 1, then this is not a very good splitting, even if the set of 1 is pure (which it is). Thus, we want to add the Gini impurity of the first set, times 99/100, and the Gini impurity of the second set, times 1/100. In Figure 7.22 we can see an example of a splitting of 8 data points into sets of size 6 and 2, where the Gini impurity is calculated as a weighted sum.



**Figure 7.22.** The Gini impurity of the split is the weighted average of the Gini indices of the two leaves. Since the left leaf has 6 elements and the right leaf has 2 elements, then the weights associated to the Gini indices of the leaves are 6/8 and 2/8.

## 7.4 Back to recommending apps: Building our decision tree using

## Gini index

In section 5.2 we built a decision tree for our app recommendation dataset, based on accuracy. In this section we'll build it using Gini impurity. Will we get the same decision tree? Let's see!

In Table 7.5 we can see the dataset again.

**Table 7.5. Our app dataset again, with three columns: Gender, Age, and App that the user downloaded.**

Gender	Age	App
F	young	 Atom Count
F	adult	 Check Mate Mate
M	adult	 Beehive Finder
F	adult	 Check Mate Mate
M	young	 Atom Count
M	young	 Atom Count

In order to build this tree, we'll first calculate the Gini impurity index of the root (the labels). Let's abbreviate our apps as follows:

- A: Atom Count
- B: Beehive Finder
- C: Check Mate Mate

Our goal is to find the Gini impurity index of the set {A, C, B, C, A, A}. Note that the probabilities that we pick a letter out of this set of three letters are the following:

- Letter A:  $\frac{1}{2}$  (since there are 3 A's out of 6 letters).
- Letter B:  $\frac{1}{6}$  (since there is 1 B out of 6 letters).
- Letter C:  $\frac{1}{3}$  (since there are 2 C's out of 6 letters).

Using the formula that we learned in the previous section, the Gini impurity index for this set is:

$$\text{Gini impurity index of } \{A, C, B, C, A, A\} = 1 - \left(\frac{1}{2}\right)^2 - \left(\frac{1}{6}\right)^2 - \left(\frac{1}{3}\right)^2 = 0.611.$$

Now, let's look at the two possible ways to split this dataset: by Gender or by Age.

### **SPLITTING BY GENDER**

As we've seen before, splitting by gender gives us the two following sets:

- Females: {A, C, C}
- Males: {B, A, A}

The Gini impurity indices of these two are the following:

- Females:  $1 - \left(\frac{1}{3}\right)^2 - \left(\frac{2}{3}\right)^2 = 0.444$ .
- Males:  $1 - \left(\frac{1}{3}\right)^2 - \left(\frac{2}{3}\right)^2 = 0.444$ .

Therefore, the average Gini impurity index of this splitting is 0.611. Since the Gini impurity of the root was 0.444, we conclude that the Gini gain is:

$$\text{Gini gain} = 0.611 - 0.444 = 0.167.$$

### **SPLITTING BY AGE**

As we've seen before, splitting by gender gives us the two following sets:

- Young: {A, A, A}
- Adults: {C, B, C}

The Gini impurity indices of these two are the following:

- Young:  $1 - \left(\frac{3}{3}\right)^2 = 0$ .
- Adults:  $1 - \left(\frac{1}{3}\right)^2 - \left(\frac{2}{3}\right)^2 = 0.444$ .

Therefore, the average Gini impurity index of this splitting is 0.222 (the average of 0 and 0.444). Since the Gini impurity of the root was 0.611, we conclude that the Gini gain is:

$$\text{Gini gain} = 0.611 - 0.222 = 0.389.$$

### COMPARING GINI INDICES

If we compare the two Gini gains, we see that 0.389 is larger than 0.167. This implies that the age feature is better than the gender feature for splitting the data. This comparison can be seen in Figure 7.23.

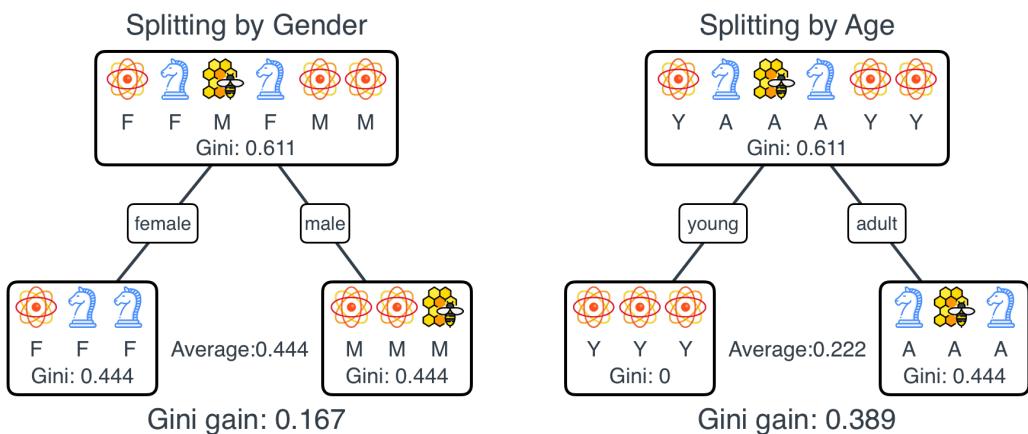


Figure 7.23. The two splittings, by gender and age, and their Gini index calculations. Notice that the splitting by age has a higher Gini gain, so we pick this one.

So we have decided to go for the age split. We get two nodes:

- The left node, where everyone downloaded Atom Count. This node has Gini impurity index 0.
- The right node, where one person downloaded Beehive Finder and two downloaded Check Mate Mate. This node has Gini impurity index 0.444.

The left node with Gini impurity index 0 can no longer be split. As a matter of fact, we don't want this node to be split any longer, since we can already safely recommend Atom Count to everyone who lands in this node. This node, then becomes a leaf.

The right node can still be split. Note that we've already split by age, so everyone in this node is an adult. The only split we can make is by gender. Let's see what happens when we split by gender.

- The two females downloaded Check Mate Mate. If we make a node with these two, the node has Gini impurity index of 0.

- The male downloaded Beehive Finder. If we make a node with this one, the node has a Gini impurity index of 0.

So we are done! The resulting split is in Figure 7.24.

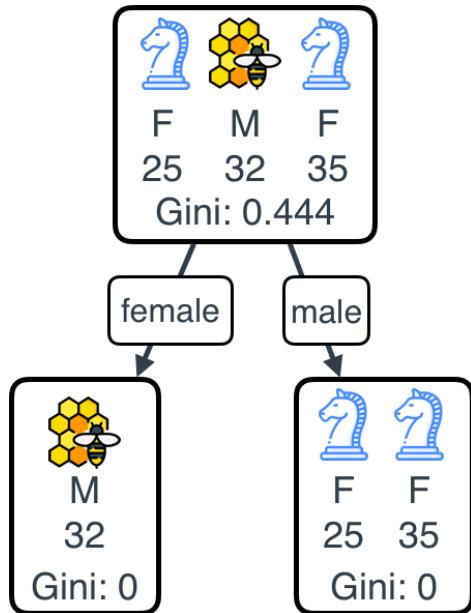


Figure 7.24. The only split we can do here is by gender. This gives us two groups, one with one male who downloaded Beehive Finder, and one with two females who downloaded Check Mate Mate. Both have Gini impurity index zero, since everyone in the group downloaded the same app.

Now we finally have our decision tree! It's illustrated in Figure 7.25.

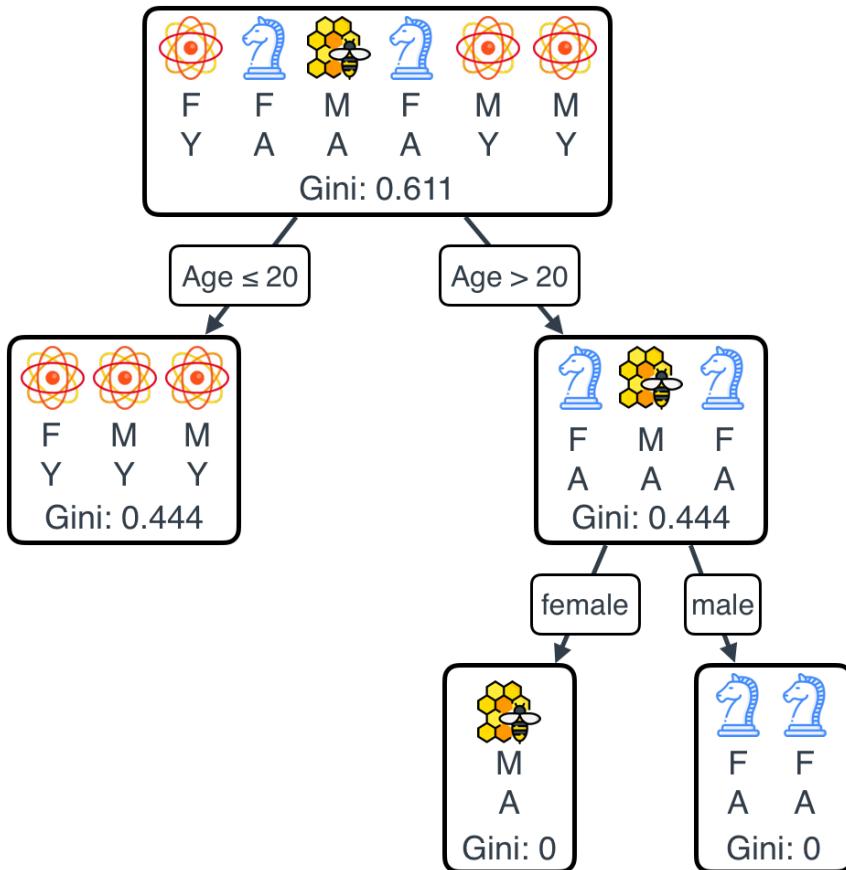


Figure 7.25. The final decision tree that we've obtained. The first split divides the users between those whose age is 20 or less, and those whose age is more than 20. For the younger than 20, we recommend Atom Count. Those older than 20 get split them among males and females. The males get recommended Beehive Finder, and the females get recommended Check Mate Mate.

## 7.5 When do we stop building the tree? Hyperparameters

As you can imagine, if we continue building this tree forever, we'll end up with many leaves, which causes tremendous overfitting, and we should not do it. In our process, we stopped building the tree in the previous section when each node only had elements of one class, but if we are not that lucky, we could potentially end up with a very deep tree with many leaves. We want to know a good place to stop. For that, we can introduce several hyperparameters. The following work well in the practice:

- Stop the training when the gain in accuracy or Gini index is below some threshold.
- Don't split a node if it has less than a certain number of elements.

- Don't split a node if the split results in one of the leaves results to be smaller than a certain size.
- Stop building the tree after you reach a certain number of levels.

The way we pick this hyperparameters is either by experience, or running grid search.

## 7.6 Beyond questions like yes/no

We learned a way to split data based on a feature that only has two classes. Namely, features that split the users into two groups such as female/male, or young/adult. What if we had a feature that split the users into three or more classes? For example, what if our users had a pet, which could be a dog, a cat or a bird, and our table had information on these? Or even worse, what if the age feature actually had numbers, such as 15, 20, 32, instead of just two categories such as young/adult? It turns out that these two cases are not much more complicated than the original one. We'll study them separately.

### 7.6.1 Features with more categories, such as Dog/Cat/Bird

The idea to tackle features with more than two classes, is break down the information into a series of binary (yes/no) questions. In this way, a feature such as gender, which only has two classes (female and male), can be broken down using the question "Is the user female?". In this way, the females receive a 'yes', and the males a 'no'. The same result can be obtained with the question "Is the user male?", and the opposite answer. Therefore, with one yes/no question, we can split this feature using one binary question.

When the feature has more classes, such as Dog/Cat/Bird, we simply use more binary (yes/no) questions. In this case, we would ask the following:

- Does the user own a dog?
- Does the user own a cat?
- Does the user own a bird?

You can see that no matter how many classes there are, we can split it into a number of binary questions.

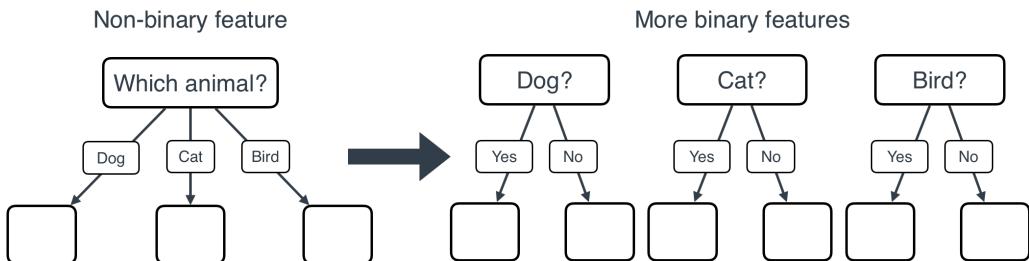


Figure 7.26. When we have a non-binary feature, for example one with three or more possible categories, we instead turn it into several yes/no features, one for each category.

Now, what happens with the table of data? Simply, the one column with three classes turns into three columns, one for each of the classes. The technical term for this process is *one-hot encoding*. One-hot encoding simply removes the column of the table that has three or more categories, and adds as many rows as categories, with each row filled with 0s and 1s. The term one-hot encoding comes from the fact that among the resulting new columns, only one of them has a 1 per row, and the others have a zero. Here's an example.



Animal
Dog
Cat
Bird
Dog
Bird

Dog?	Cat?	Bird?
1	0	0
0	1	0
0	0	1
1	0	0
0	0	1

Figure 7.27. One-hot encoding turns a non-binary feature into several binary features, which helps us build our decision tree. Notice that the table in the right conveys the same information as the table in the left. For example, observe the first row. The table in the left contains a dog. The table in the right has three columns, and there is a 1 under the 'dog' column, and zeroes under the 'cat' and 'bird' columns.

### 7.6.2 Continuous features, such as a number

So far, we have only seen categorical features, namely, those that have a certain fixed number of classes, such as female/male, young/adult, or dog/cat/bird. But some features are numerical, such as age, size, length, price, and they can't be put into buckets. Even if we were able to put it into buckets, say, one bucket for each age, we may still want to benefit from the numerical nature of this feature. For example, if someone is 15 years old, and they downloaded a certain app, we would imagine that a 14 year old is likely to also download that app. If we had a bucket for 15 year olds, and a different bucket for 14 year olds, we would lose this information. So how do we deal with these? Simple, we still want to ask yes/no questions, we just have to ask a bunch more.

In order to see this, let's go back to the original dataset, where age was given as a number, and not as a category young/adult (Table 7.6).

Table 7.6. Our original dataset where users have age and gender, and age is given as a number.

Gender	Age	App
F	15	 Atom Count
F	25	 Check Mate Mate
M	32	 Beehive Finder
F	35	 Check Mate Mate
M	12	 Atom Count
M	14	 Atom Count

When the age column was given as two categories, 'young' and 'adult', it was easy to split, since we only had one question to ask, namely "Is the user young or old?". But now, we have many questions, such as:

- Is the user younger than 16?
- Is the user older than 24?
- Is the user younger than 27.5?

It seems that we have an infinite number of questions. However, notice that many of them give us the same answer. For example, if we ask "Is the user younger than 20?", our answers for the 6 users in the dataset would be no, yes, yes, yes, no, no. If we ask "Is the user younger than 21?", the answers are exactly the same. Both questions would split our users into two sets, one of ages 12, 14, 15, and the other one of ages 25, 32, 35. As a matter of fact, the only thing that matters is where we decide to cut the ages into two groups. Our ages are, in order, 12, 14, 15, 25, 32, 35. We can add the cutoff anywhere in between consecutive pairs of numbers, and no matter what number we pick as a cutoff, we'd get the same result. The number we pick can be completely arbitrary.

So here are the 7 possible questions, with arbitrary cutoffs, as illustrated in Table 7.7.

**Table 7.7. The 7 possible questions we can pick, each one 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. The cutoffs are picked arbitrarily.**

Question	First set	Second set
Is the user younger than 7?	empty	12, 14, 15, 25, 32, 35
Is the user younger than 13?	12	14, 15, 25, 32, 35
Is the user younger than 14.5?	12, 14	15, 25, 32, 35
Is the user younger than 20?	12, 14, 15	25, 32, 35
Is the user younger than 27?	12, 14, 15, 25	32, 35
Is the user younger than 33?	12, 14, 15, 25, 32	35
Is the user younger than 100?	12, 14, 15, 25, 32, 35	empty

Notice that the first and last questions don't add much to our data, since they don't split the users into two non-empty groups. We'll forget about these, and we end up with 5 splittings, as illustrated in Figure 7.28.

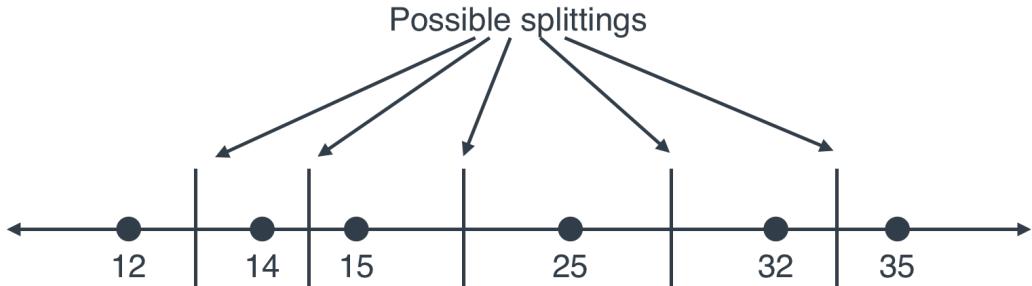


Figure 7.28. A graphic of the 5 possible splittings we can do of the users by age.

Now all we have to do is check the accuracy (or Gini impurity) of each of the splittings. For example, if we split in between 25 and 32 (say, if we make the cut at 30 years old) we get the following:

- **Younger than 20:** {12, 14, 15, 25}
- **Older than 20:** {32, 35}

Notice that this split gives us two leaves, as illustrated in Figure 7.29.

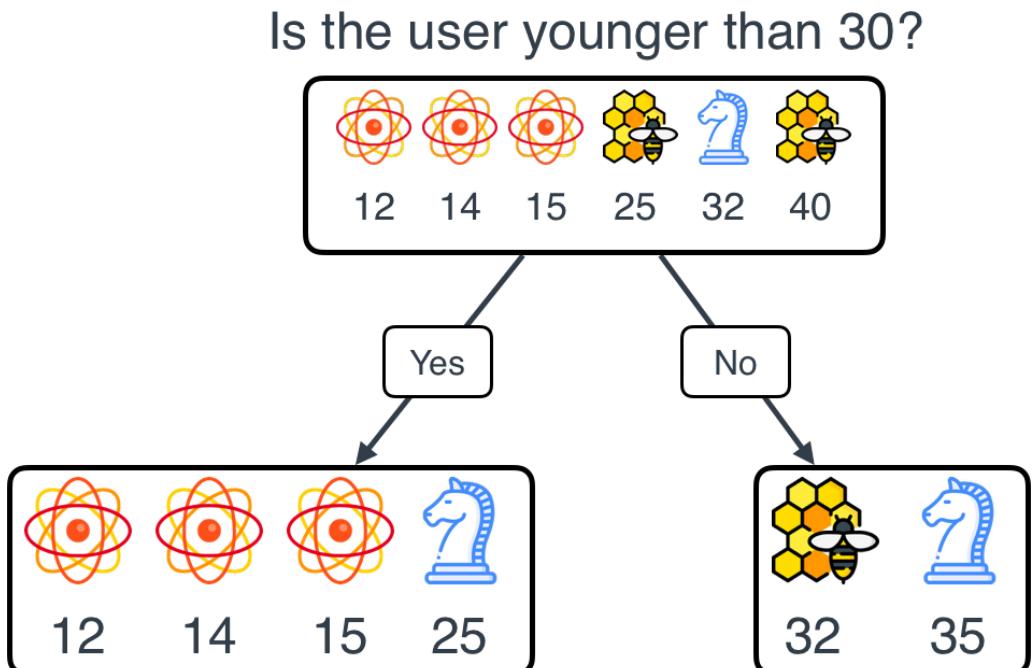


Figure 7.29. We split the users using age 30 as a cutoff. In the node to the left, we get the ages 12, 14, 15, and

25. In the node to the right, we get the ages 32, and 35.

### COMPUTING THE ACCURACY OF THIS SPLIT

As seen previously, the idea of measuring the accuracy is to use this sole question to build a classifier, namely, to recommend apps to users based on their age respective to 30. The most popular app for the group of people under 30 is Atom Count, so we'll recommend that one to this group. Notice that we get three out of four correct. As for the people 30 or older, we can either recommend Check Mate Mate or Instagram, since with either one we are correct one out of two times. In total, we are correct four times out of six, so our accuracy is 4/6.

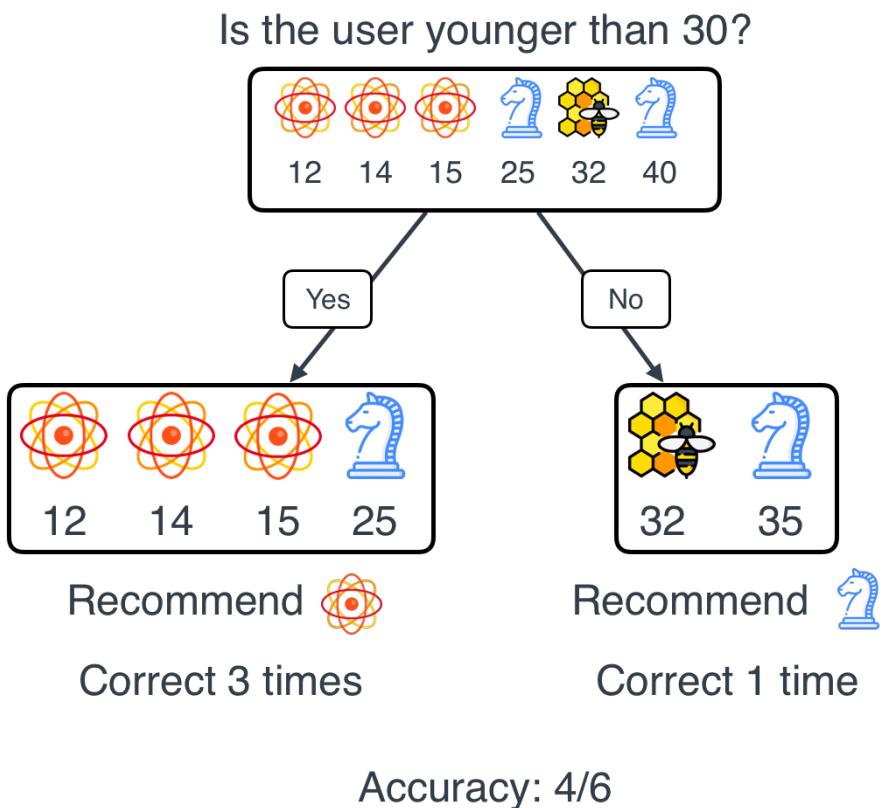


Figure 7.30. Using accuracy to measure this split. For the left leaf, we can recommend Atom Count (the most popular), and we get 3 users out of 4 correct. For the right leaf, we can recommend either Beehive Count or Check Mate Mate, and we'll get 1 correct out of 2. In total, we get 4 correct out of 6, so our accuracy is 4/6, or 0.666.

### **COMPUTING THE GINI IMPURITY INDEX OF THIS SPLIT**

Now we'll compute the gain in Gini impurity index of this split. Note that the Gini impurity index of the root is 0.611, as we've previously calculated.

The left leaf has three users downloading Atom Count, and one downloading Check Mate Mate, so the Gini impurity is

$$1 - \left(\frac{3}{4}\right)^2 - \left(\frac{1}{4}\right)^2 = 0.375.$$

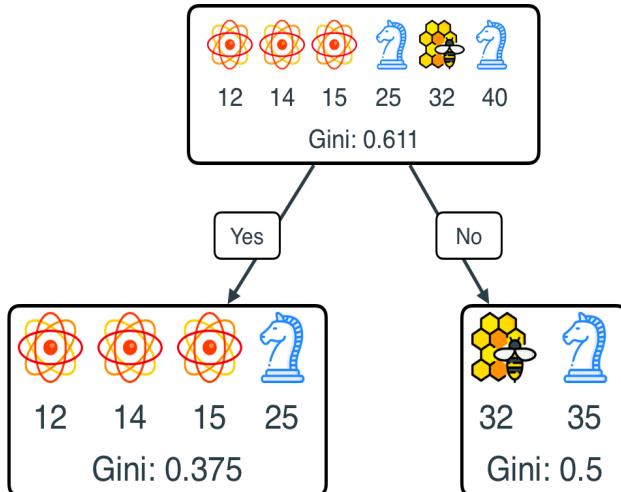
The right leaf has one user downloading Instagram, and one downloading Check Mate Mate, so the Gini impurity is

$$1 - \left(\frac{1}{2}\right)^2 - \left(\frac{1}{2}\right)^2 = 0.5.$$

Also note that the left leaf has four users and the right leaf has two users. Since there are six users in total, then the weights for the entropies of the left and right leaves is 4/6 and 2/6, respectively (or  $\frac{2}{3}$  and  $\frac{1}{3}$ ). Thus, our weighted Gini impurity at the leaves is:

$$\frac{4}{6} \cdot 0.375 + \frac{2}{6} \cdot 0.5 = 0.417.$$

Is the user younger than 30?



$$\text{Gini (weighted) average} = 0.375 \left( \frac{4}{6} \right) + 0.5 \left( \frac{2}{6} \right) = 0.417$$

$$\text{Gini gain} = 0.611 - 0.417 = 0.194$$

Figure 7.31. Using the Gini impurity index to measure this split. The Gini index for the root is 0.611. The Gini indices for the left and right leaves, respectively, are 0.375 and 0.5. The Gini average is the weighted average of the Gini indices of the leaves. Since the left leaf has 4 elements and the right leaf has 2 elements, then we weight the left Gini index by 4/6, and the right one by 2/6. This gives us a weighted average of 0.417. This gives us a Gini gain of 0.194 for this split.

#### COMPARING OUR METRICS FOR ALL THE SPLITS

Now, in order to find the best split, all we have to do is calculate the accuracy (or the Gini gain) for all of our possible 5 splits. I encourage you to calculate them by yourself, and check that you get the same answers I got. If you need some help, I have calculated them in the Github repo, in the notebook corresponding to Decision Trees. The results are in Table 7.8.

**Table 7.8.** In the left column we can see the splittings over age of our set. The next columns calculate

the accuracy, Gini impurity, and information gain of each of the splittings.

Splitting	Labels	Accuracy	Gini gain
{12} {14, 15, 25, 32, 40}	A A, A, C, B, C	3/6	0.078
{12, 14} {15, 25, 32, 40}	A, A A, C, B, C	4/6	0.194
{12, 14, 15} {25, 32, 40}	A, A, A C, B, C	5/6	0.389
{12, 14, 15, 25} {32, 40}	A, A, A, C B, C	4/6	0.194
{12, 14, 15, 25, 32} {40}	A, A, A, C, B C	4/6	0.144

From here, we can see that the third splitting, namely the one that split the users into the sets of ages {12, 14, 15} and {25, 32, 40}, is the best in terms of highest accuracy, lowest Gini impurity, and highest information gain.

Great! We've determined that based on age, our best split is to separate them into two groups: {12, 14, 15}, and {25, 32, 40}. We can ask the question "Is the user younger than 20?" (again, any number between 15 and 25 would work, but to have a nice split, we are taking the midpoint). Is this the best split overall? Well, we don't know yet. We still have to compare it with the split according to the other feature, namely Gender. We recorded this in Table 7.9.

**Table 7.9. Remembering the split by gender, its accuracy, Gini impurity, and information gain.**

Splitting	Labels	Accuracy	Gini impurity	Information gain
Male/Female	P, P, I P, W, W	0.666	0.444	

Notice that if we compare the gender split with the best age split, the best age split still wins in terms of accuracy, Gini impurity, and entropy. Therefore, the feature we'll pick for our first split is age, and the threshold we pick is 20.

Our final tree looks as follows (Figure 7.32). The pseudocode for predicting a recommendation for a user, then, is the following:

```

predict(user):
    if user's age is less than or equal to 20:
        recommend Atom Count
    else:
        if user's gender is female:
            recommend Check Mate Mate
        else:
            recommend Beehive Finder

```

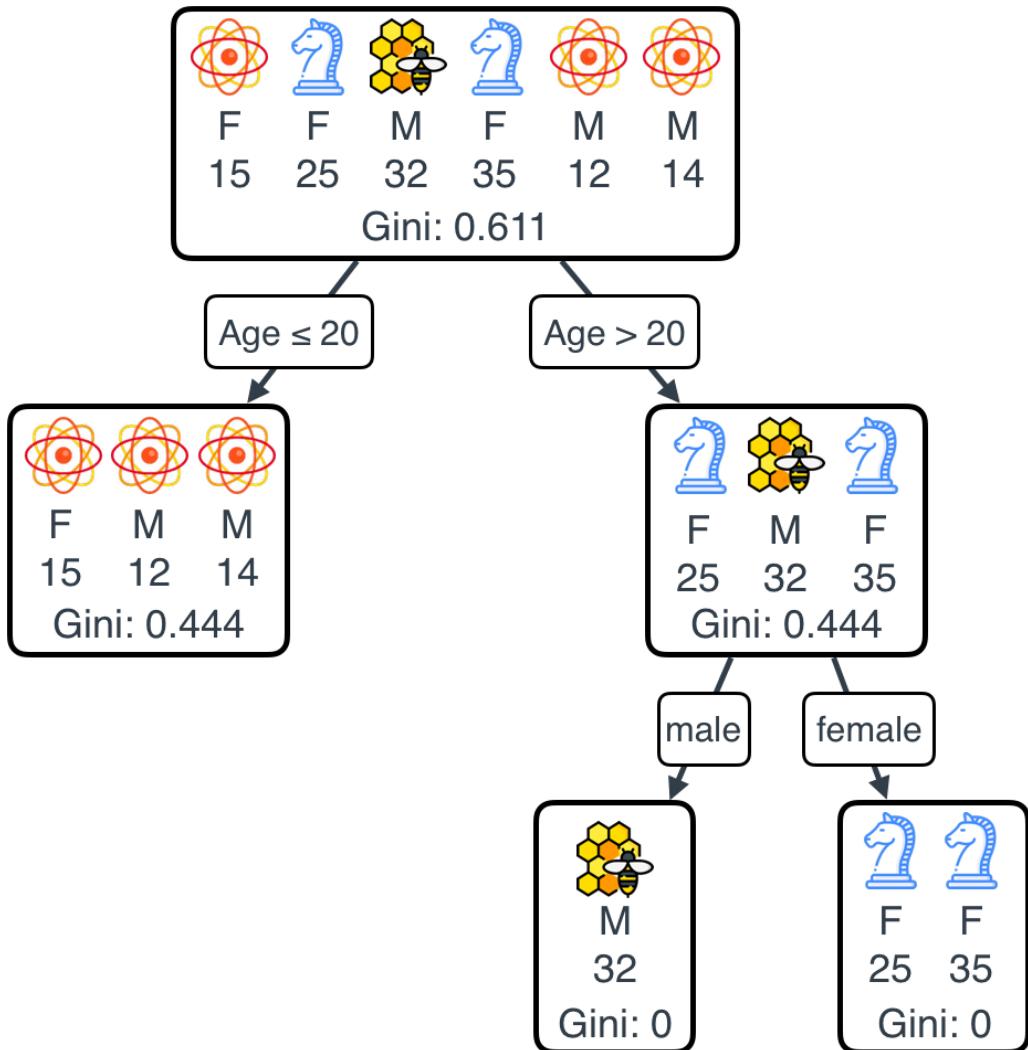


Figure 7.32. Our resulting decision tree. Just like before, the first split is based on age (less than or equal to 20), and the second is based on gender (male or female). The recommendations are made accordingly..

## 7.7 Coding a decision tree with sklearn

In order to code our decision trees, we'll use a great and very well-known package called Scikit Learn, or sklearn for short. We need to slightly massage our data in order to have it ready for sklearn. We need to make sure that among the features, only numbers are fed. A feature with values such as 'male' or 'female', will not be accepted. Therefore, we'll one-hot encode the Gender feature. We'll store the dataset as a Pandas dataframe using the code below, and the result is in Table 7.10.

```
import pandas as pd
app_dataset = pd.DataFrame({
    'Gender_Female':[1,1,0,1,0,0],
    'Gender_Male':[0,0,1,0,1,1],
    'Age': [15, 25, 32, 35, 12, 14],
    'App': ['Atom Count', 'Check Mate Mate', 'Beehive Finder', 'Check Mate Mate', 'Atom Count', 'Atom Count']})
print(app_dataset)
```

**Table 7.10.** Our app dataset, as a Pandas dataframe. The columns Age and App stay the same. The Gender column has been one-hot encoded and turned into two columns, Gender\_Female, and Gender\_Male, in order to only have numerical features. The App column doesn't need to be one-hot encoded, since these are the labels, not the features.

	Age	Gender_Female	Gender_Male	App
0	15	1	0	Atom Count
1	25	1	0	Check Mate Mate
2	32	0	1	Beehive Finder
3	35	1	0	Check Mate Mate
4	12	0	1	Atom Count
5	14	0	1	Atom Count

Now, we will split the features and the labels, so that our features DataFrame is formed by the 'Age', 'Gender\_Female', and 'Gender\_Male' columns, and our labels DataSeries is formed by the column apps

```
features = app_dataset_new[['Age','Gender_Female','Gender_Male']]
labels = app_dataset_new[['App']]
```

Finally, we train our model by using the `DecisionTreeClassifier()` class from sklearn. The model will be called 'app\_decision\_tree', and the command for fitting the data is simply 'fit()'.

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

That's all we need to fit a decision tree to our data! Let's plot this tree (the commands for plotting it are in the Github repo).

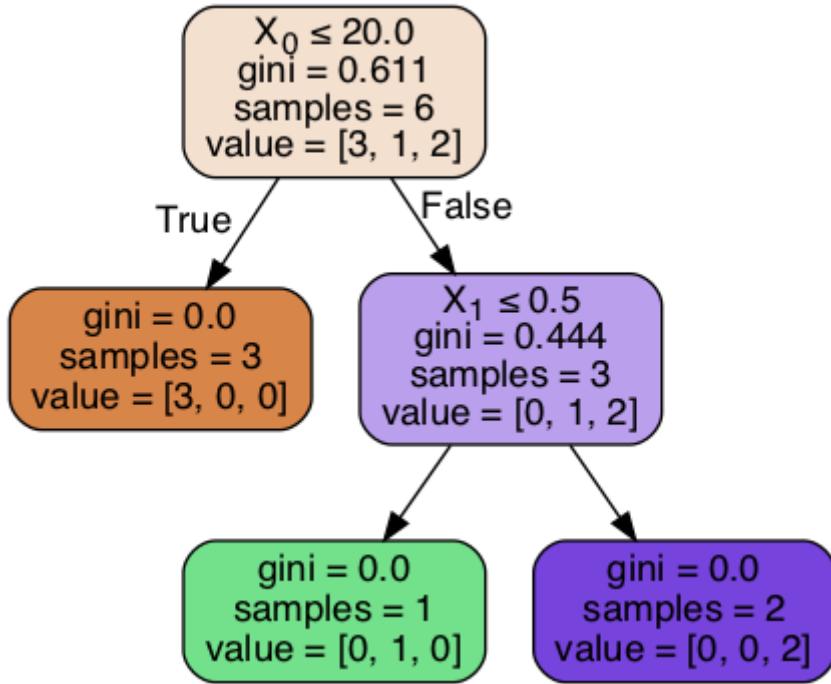


Figure 7.33. The plot of the decision tree that we trained in sklearn.

Let's analyze what this plot is. In each of the nodes, we have some information, and it means the following:

- $X_0$ : The first column in our features, namely, age.
- $X_1$ : The second column in our features, namely, Gender\_Female.
- Gini: The Gini impurity index of the elements that fall in each node.
- Samples: The number of samples on each node.
- Value: This vector shows the distribution of labels in each node. For example, [3, 1, 2] means that in this node, there are 3 users who downloaded Atom Count, one user who downloaded Beehive Finder, and 2 who downloaded Check Mate Mate.

In this case, note that the top node gets split using the rule ' $X_0 \leq 20$ ', or 'Age  $\leq 20$ '. To the left, we get 'True', which is those users whose ages are less than or equal to 20, and to the right, we get 'False', which is those whose age is greater than 20. The node to the right is split using

the rule ' $X_1 \leq 0.5$ ', or 'Gender\_Female  $\leq 0.5$ '. This is a bit strange, but if we remember that this column 'Gender\_Female' assigns a 1 to females, and a 0 to males, then 'Gender\_Female  $\leq 0.5$ ' is true when the user is male (0), and false when the user is female (1). Thus, this node splits the males to the left, and the females to the right.

Figure 7.34 shows a comparison of the tree that sklearn obtained and the tree that we obtained, for clarity.

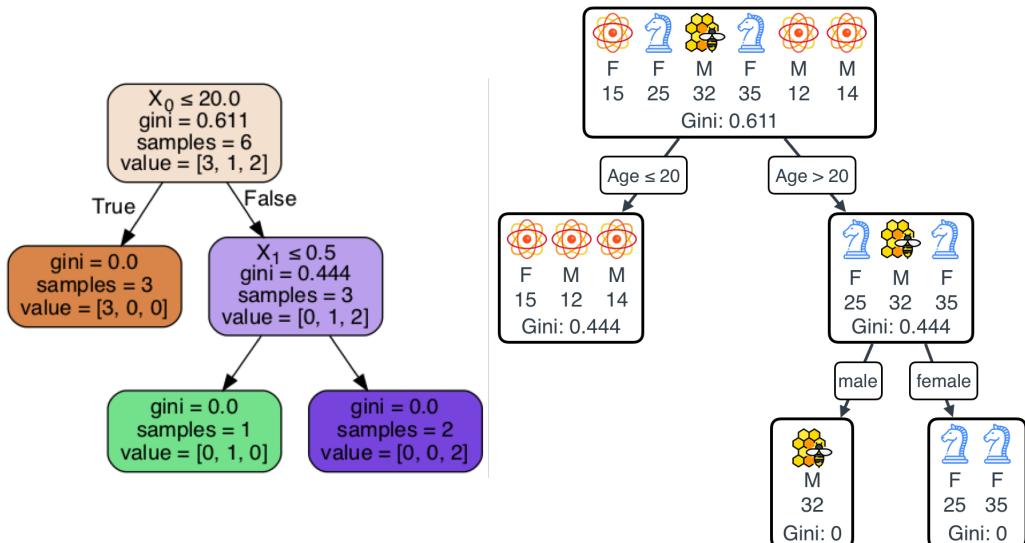


Figure 7.34. The plot of the decision tree that we trained in sklearn.

Sklearn obtained the same tree that we obtained with our reasoning. This is great! Now let's try with a slightly larger example, let's go back to our days of spam recommendation.

## 7.8 A slightly larger example: Spam detection again!

Let's go back to our example from Chapter 5, where we predicted spam/ham emails from the number of appearances of the word 'lottery' and 'sale'. Can we use decision trees for spam detection too? Of course! Let's look at a bigger table of emails, such as shown in Table 7.11.

Table 7.11. A bigger table of spam/ham emails, with the number of times the words 'lottery' and 'sale' appear.

Lottery	Sale	Spam
7	1	No

3	2	No
3	9	No
1	3	No
2	6	No
4	7	No
1	9	Yes
3	10	Yes
6	5	Yes
7	8	Yes
8	4	Yes
9	6	Yes

Let's try to build a good decision tree here. We can use accuracy, Gini impurity, or simply plug it in sklearn. Let's use sklearn to see what we get.

First, let's put our data into a Pandas DataFrame.

```
spam_dataset = pd.DataFrame({
    'Lottery':[7,3,9,1,2,4,1,3,6,7,8,9],
    'Sale':[1,2,3,3,6,7,9,10,5,8,4,6],
    'Spam': ['spam','spam','spam','spam','spam','spam','ham','ham','ham','ham','ham','ham']})
```

In order to do this graphically, let's plot the points in a graph (this appears in the Github repo).

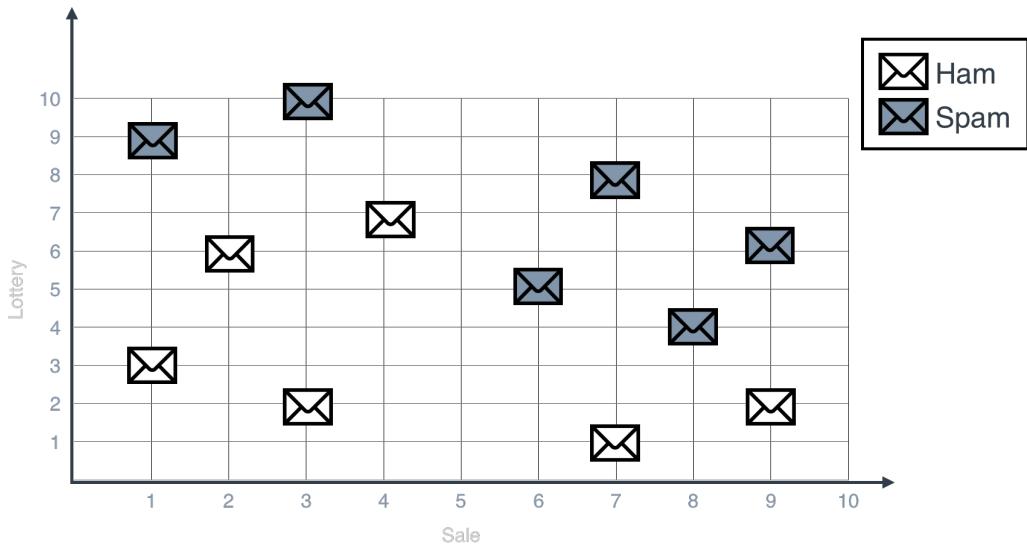


Figure 7.35. The graph of the set of spam/ham emails, where the horizontal (x) axis represents the number of times the word 'sale' appears, and the vertical (y) axis represents the number of times the word 'lottery' appears. The gray emails are spam, and the white emails are ham.

Next, we split into features and labels

```
features = spam_dataset[['Lottery', 'Sale']]
labels = spam_dataset['Spam']
```

Now, let's define our decision tree model and train it.

```
spam_decision_tree = DecisionTreeClassifier()
spam_decision_tree.fit(X,y)
```

The decision tree we get is in Figure 7.36.

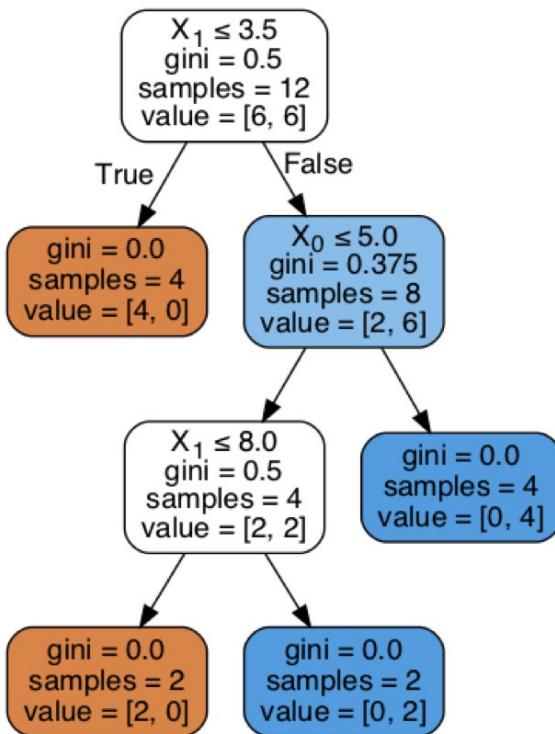


Figure 7.36. The decision tree that fits our spam/ham dataset.

It looks complicated, but let's analyze it step by step. First, let's look at what it really does to predict if emails are spam or ham. This is the pseudocode for generating the predictions:

```

predict(email):
    if number of appearances of 'sale' is less than or equal to 3.5:
        classify as ham
    else:
        if number of appearances of 'lottery' is greater than 5.0:
            classify as spam
        else:
            if number of appearances of 'sale' is less than or equal to 8.0:
                classify as ham
            else:
                Classify as spam

```

But I like to see decision trees in a graphical way too. Let's analyze this tree node by node, and see what it is doing to the dataset. The top node splits the data based on the rule  $X_1 \leq 3.5$ . This feature is the second column, namely, the appearances of the word 'lottery'. Thus, the dataset is broken into those emails with 3.5 or less appearances of the word 'lottery', and those with more than 3.5 appearances. This is a horizontal split, as Figure 7.37 shows.

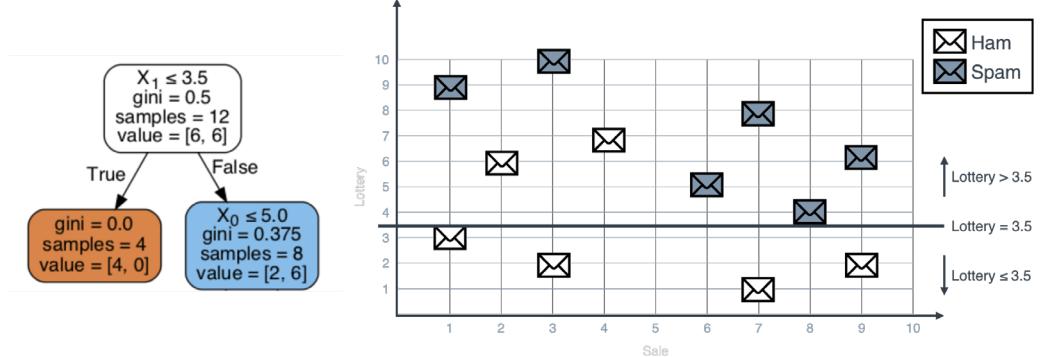


Figure 7.37. The first split in the decision tree. Note that the left leaf corresponds to the area underneath the horizontal line, with four ham emails. The right node corresponds to the area over the horizontal line, with 6 spam emails and 2 ham emails.

Note that all four emails below the horizontal line are ham, so these all form a leaf. This leaf is shown in the tree, to the left of the top node. Notice that it has 4 samples,  $\text{gini} = 0.0$  (since they are all ham, so the data is pure), and  $\text{value} = [4,0]$ , as there are four ham emails, and zero spam.

The tree must now decide how to split the data above the line, which corresponds to the node in the second row, to the right. Note that it has 8 samples, 2 of them ham, and 6 of them spam. The tree chooses to split using the rule  $X_1 \leq 3.5$ , which corresponds to 'Sale  $\leq 5.0$ '. This means, the split corresponds to a vertical line, as Figure 7.38 shows.

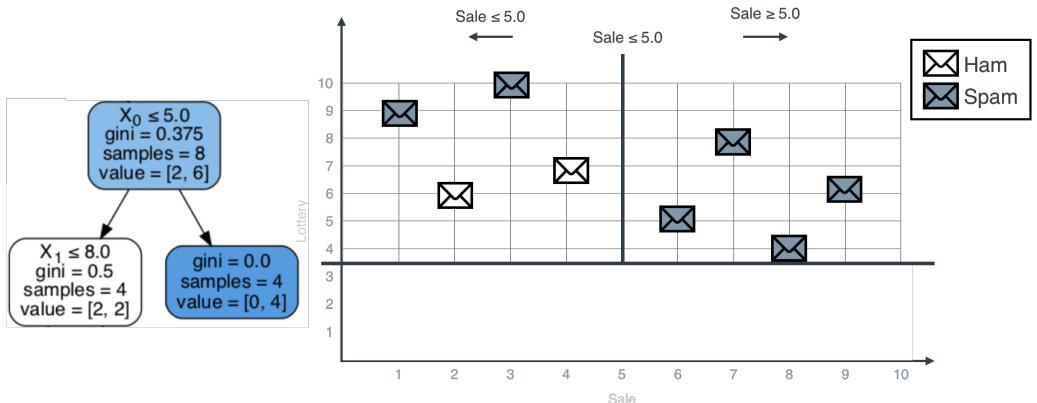


Figure 7.38. The second split in the decision tree. Note that the left leaf corresponds to the area to the left to the vertical line, with four ham emails. The right node corresponds to the area to the right of the vertical line, with 6 spam emails and 2 ham emails.

Note that all the four emails to the right of the line are spam, so this is a pure leaf. It corresponds to the leaf to the right. The one in the left contains two spam and two ham emails, so we need to split it further. This split is in Figure 7.39.

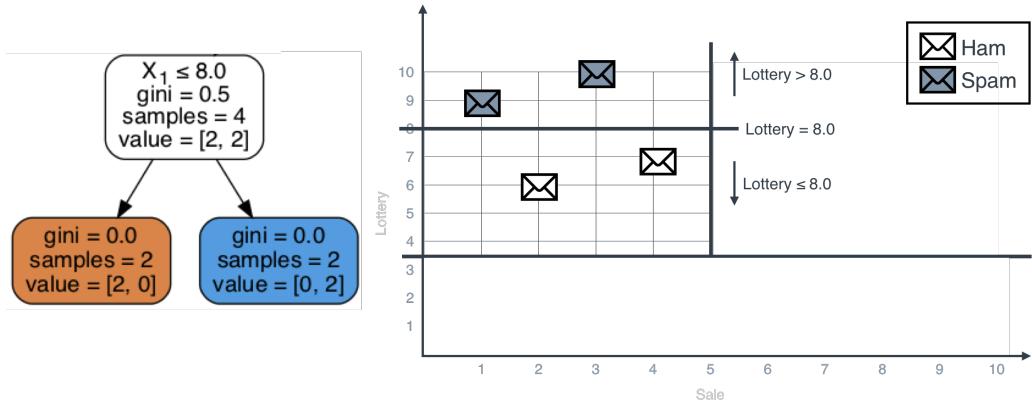


Figure 7.39. The last split in the decision tree. Note that the left leaf corresponds to the area over the horizontal line, with two spam emails. The right node corresponds to the area underneath the horizontal line, with 2 ham emails.

So if we plot all these lines together, our tree delineates the boundary seen in Figure 7.40. Note that all the emails to one side of the boundary are spam, and to the other side are ham, a perfect split!

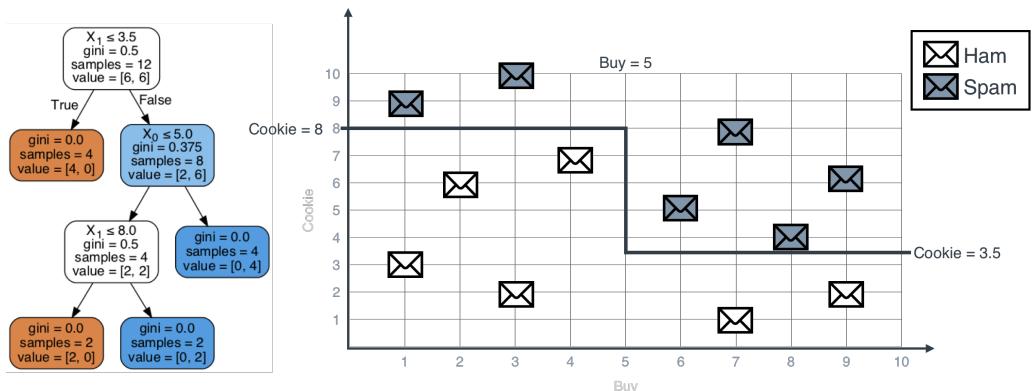


Figure 7.40. The decision tree on the left indicates a boundary on the right, that splits the data.

## 7.9 Decision trees for regression

Throughout this chapter we've seen how useful decision trees are for classification problems. But you'll be happy to know that they are just as useful for regression problems too. The construction of the tree is almost the same, except at each step of the process we divide the nodes in a different way. For classification, we choose the split that maximizes the accuracy or that minimizes the Gini impurity index. For regression, we choose the split that minimizes the total squared error. We saw this in chapter 3, but since it's been a while, I'll remind you: to calculate the total squared error of a set of values, we first calculate the mean, or average, of the values. Then, we calculate the squares of the difference of each element and the mean, and

finally we add them. For example, for the set of numbers 1,2,9, their average is  $\frac{1+2+9}{3} = 4$ , so their total squared error is

$$MSE = (1 - 4)^2 + (2 - 4)^2 + (9 - 4)^2 = 9 + 4 + 25 = 34.$$

Let's build a small regression tree in an example data set. Let's say that we are trying to study the customers of our app Atom Count, and we are trying to model their engagement based on age. In this case, engagement is a number from 0 to 7, based on how many days of the previous week the user was engaged with the app. The engagement level based on age is in table 7.12.

**Table 7.12.** A table of customers of Atom Count, where we record their age in years, and their engagement with the app as the number of days in which they used it in the previous week.

Age	Engagement
10	7
20	5
30	6
40	0
50	1
60	0
70	4
80	3

In Figure 7.41 we plot the points from table 7.12, and notice that there is no line that goes close to these points. Thus, this is not a good data set to use linear regression on. It instead seems

that the young people (10, 20, 30 years old) are highly engaged, the middle aged (40, 50, 60) are lowly engaged, and the elderly population (70, 80) have a medium level of engagement. In this section we see that this type of situation is perfect for a regression decision tree.

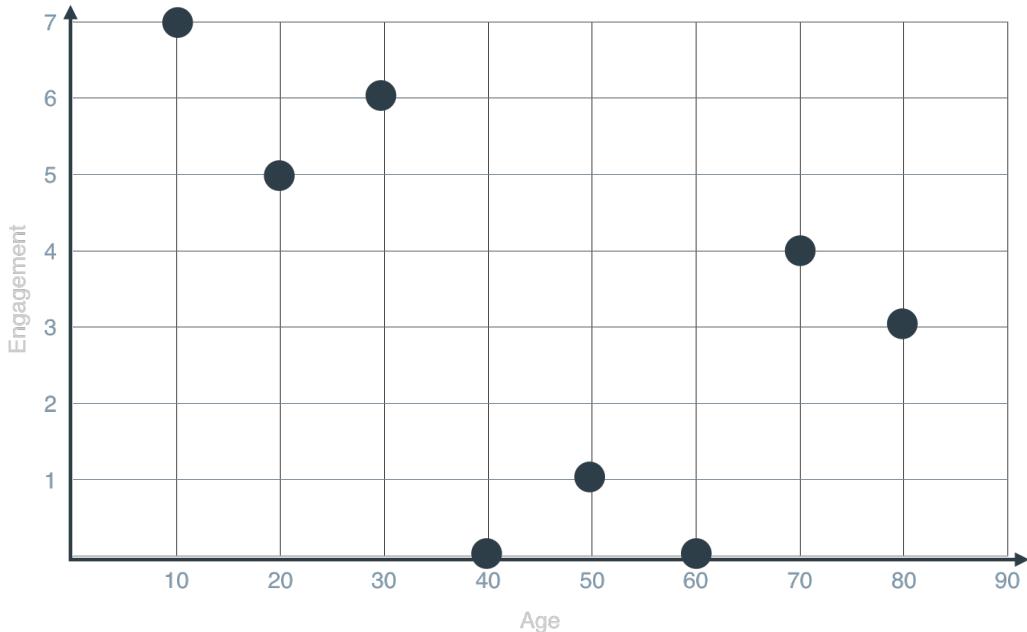


Figure 7.41. The plot of Table 7.12. The horizontal axis represents the age of the users and the vertical axis represents the number of days in the previous week where the users were engaged with the app. We will use a regression decision tree to predict the engagement level based on the age.

### 7.9.1 Building the simplest possible classifier - A decision tree that predicts the same value for each data point

Before we start building our decision tree, let's ask the following question: If we had the simplest possible classifier, namely, one that assigns the same output to any input, what would this output be? It makes sense to think that the best estimate to use is the average. For example, if we wanted to find such a model for the data in Table 7.12, we would simply output for every point a prediction of  $\frac{5+6+7+0+1+0+4+3}{8} = 3.25$ . This is equivalent to finding a horizontal line that fits the data as best as possible. In that case, the plot of this classifier is simply a horizontal line at height 3.25, as in Figure 7.42.

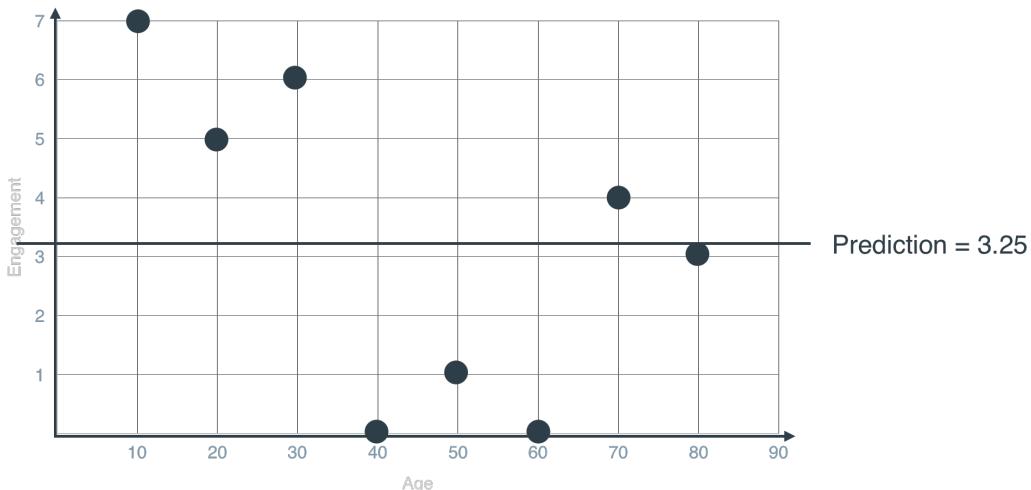


Figure 7.42. The simplest possible classifier for this dataset is a horizontal line at height 3.25.

Now, to measure this classifier, let's find the mean square error, namely, the average square distance between the points and the line at height 3.25. This is equivalent to asking: Among all the horizontal lines, which one is the 'closest' to our data? If we are thinking about mean squared error, then the answer is certainly the average. I encourage you to verify this mathematically. For this classifier, the total square error is:

$$\text{Total square error} = (5 - 3.25)^2 + (6 - 3.25)^2 + (7 - 3.25)^2 + (0 - 3.25)^2 + (1 - 3.25)^2 + (0 - 3.25)^2 + (4 - 3.25)^2 + (3 - 3.25)^2 = 51.5.$$

We'll be using this mean squared error in the next section.

### 7.9.2 Iterating on the simplest possible classifier by reducing the total square error

Of course, we can do better than that. Let's try to split the data into two sets, and build the simplest possible classifier for each of them. For example, say we split the data at 25 years, meaning that one group is formed by those 25 years old or younger, and the other group by those that are older than 25. The groups are formed as follows (keeping track of the level of engagement).

- Group 1: 7, 5.
- Group 2: 6, 0, 1, 0, 4, 3.

Now we fit a simplest possible classifier to each of the groups. This is precisely the classifier that always outputs the average value.

- Group 1: Classifier that always outputs 6.
- Group 2: Classifier that always outputs 2.33.

Now, on to calculate the total square errors for each one of these.

- Group 1: Total square error =  $(7 - 6)^2 + (5 - 6)^2 = 2$ .
- Group 2: Total square error =  $(6 - 2.33)^2 + (0 - 2.33)^2 + (1 - 2.33)^2 + (0 - 2.33)^2 + (4 - 2.33)^2 + (3 - 2.33)^2 = 29.33$ .

Notice that we started with a total square error of 51.5, calculated in the last section. We now have a total error of  $2 + 29.33 = 31.33$ . Therefore, our error gain was this difference  $51.5 - 31.33 = 20.17$ . This is illustrated in Figure 7.43.

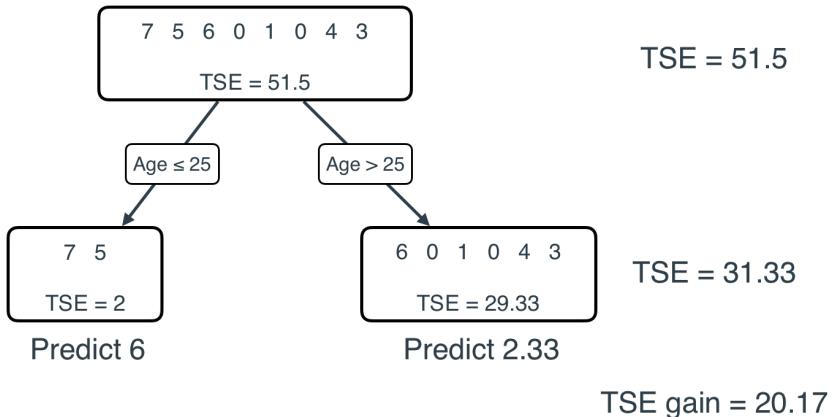


Figure 7.43. A simple decision tree on our data. The total square error gain is 20.17.

The question now is, out of all the ways of splitting the dataset, which one has the largest TSE gain? Let's study them all. The possible splits here are in between two consecutive points, which is at ages 15, 25, 35, 45, 55, 65, and 75

- Split at 15 years old. TSE = 16.07
- Split at 25 years old. TSE = 20.17
- **Split at 35 years old. TSE = 36.3**
- Split at 45 years old. TSE = 12.5
- Split at 55 years old. TSE = 4.03
- Split at 65 years old. TSE = 0.17
- Split at 75 years old. TSE = 0.07

As we can see, the highest gain is when we split at 35 years old. When we look at our data, this makes sense, since the people 35 and under are highly engaged, while those that are older are less engaged. Therefore, the first node in our decision tree splits the data as those 35 or younger, and those older than 35. The node on the left contains the customers with engagement levels 7, 5, and 6, and the one in the right contains those with levels 0, 1, 0, 4, and 3. The

prediction for the customers in each group is precisely the average of their engagement levels. For the group at the left, this average is 6, and for the group at the right, it is 1.6. This means that for customers 35 years old or younger, we'll predict an engagement level of 6, and for customers that are older than 35, we'll predict an engagement level of 1.6. The plot of the classifier is precisely a horizontal line at height 6, and a horizontal line at height 1.6, as in Figure 7.44.

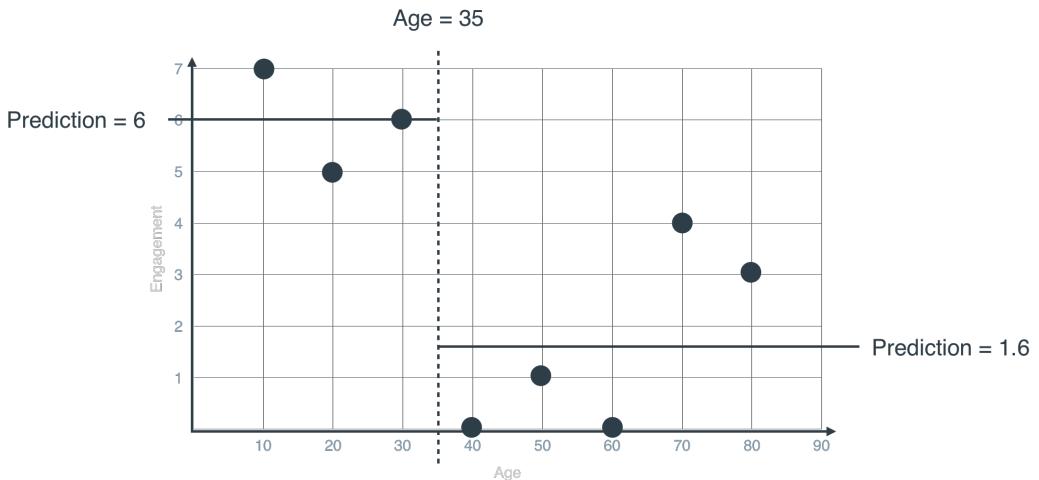
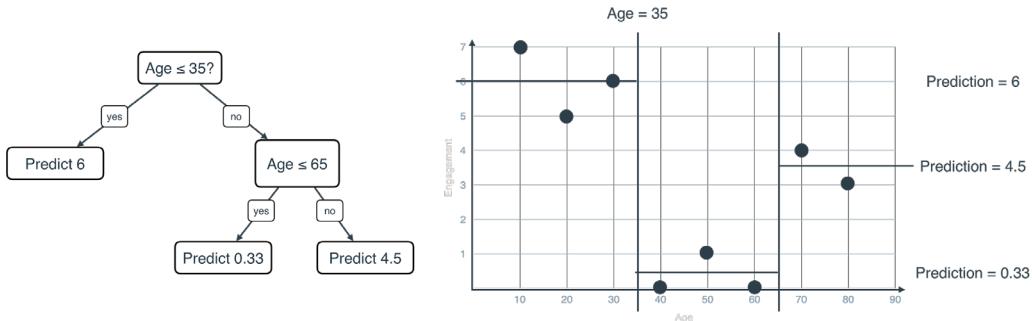


Figure 7.44. The plot of our decision tree corresponds to two lines, one for each of the two groups. The height of the horizontal line is precisely the average of the heights of the points in each group.

The way we continue building our decision tree is to do this exact same split on each of the nodes, as we did for classification. As you can see, the engagements among the left node are very homogeneous, whereas the ones in the right node aren't. This results in a split that doesn't improve the total square error by much, so we decide not to do it. If we then decide to split the node at the right, we can see (and I encourage you to do this by yourself), that the best split is at age 65. This gives us the decision tree from Figure 7.45.



**Figure 7.45.** The plot of our decision tree corresponds to two lines, one for each of the two groups. The height of the horizontal line is precisely the average of the heights of the points in each group.

## 7.10 Applications

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

- Health care
- Recommendation systems

### 7.10.1 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, their family medical history, their habits, and many other factors.

### 7.10.2 Decision trees are useful in recommendation systems

In recommendation systems, decision trees are also very 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 which 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 do this was using gradient boosted decision trees to combine over 500 different models. Other recommendation engines, such as the video recommendation engine at YouTube, also use decision trees to study the engagement of their users, and figure out what are the demographic features that best determine engagement.

In Chapter 10 we will learn more about gradient boosted decision trees and random forests, but you can see them as a collection of many decision trees, for simplicity.

## 7.11 Summary

- Decision trees are a very important algorithm in machine learning, used for classification and regression.
- The way decision trees work is by asking binary questions about our data, and making the 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.
- There are several ways to tell if a feature determines the label best. The two that we learn in this chapter are accuracy and Gini impurity index.
- Gini impurity index measures the purity of a set. In that way, a set in which every element has the same label has gini impurity index 0. A set in which every element has a different label has a Gini impurity label close to 1.
- Graphically, we can see decision trees as drawing vertical and horizontal lines (or planes) that split our data according to the different labels.
- The algorithm for building a decision tree for regression is very similar than the one for classification. The only difference is that now, we try to minimize the total squared error.
- 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.

# 8

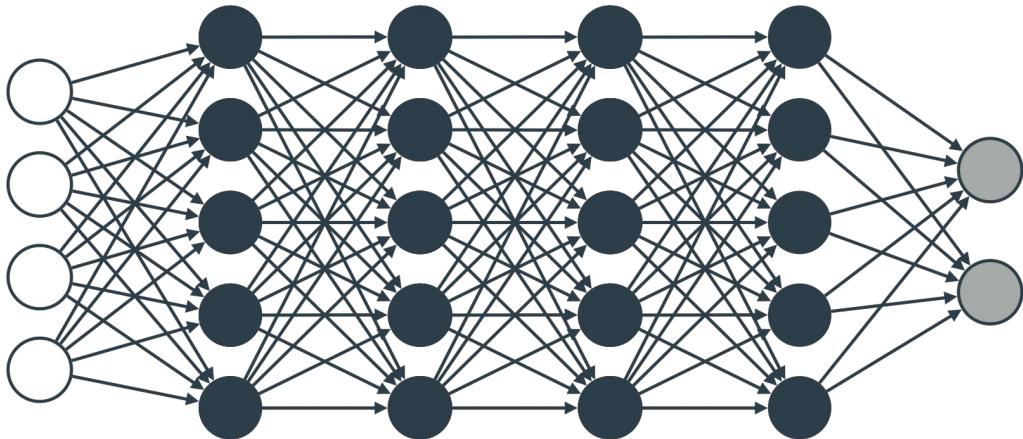
## *Combining building blocks to gain more power: Neural Networks*

### This chapter covers

- What is a neural network?
- What is a perceptron?
- Using neural networks in a simple application: sentiment analysis.
- Training neural networks using backpropagation.
- Potential problems in training neural networks, and techniques that can be used to avoid these problems.
- How to code the linear regression algorithm in Keras.
- Applications of neural networks in image recognition, text processing, and more!

In this chapter we learn neural networks. Neural networks are one of the most popular (if not the most popular) machine learning algorithms out there. They are used so much that the field has its own name: *deep learning*. Deep learning has numerous applications in the most cutting edge areas of machine learning, such as image recognition, natural language processing, medicine, self driving cars, you name it.

Neural networks are meant to, in a broad sense of the word, mimic how the human brain operates. They can be very complicated, as a matter of fact, the following image shows what a neural network looks like.



**Figure 8.1.** A neural network.

That's a scary image, right? Lots of nodes, edges, etc. However, there are much simpler ways in which neural networks can be understood. I like to see neural networks as superpositions of linear boundaries which turn into more complicated curves. This chapter is about the intuition, details, and training of neural networks. Let's begin.

## 8.1 The problem - A more complicated alien planet!

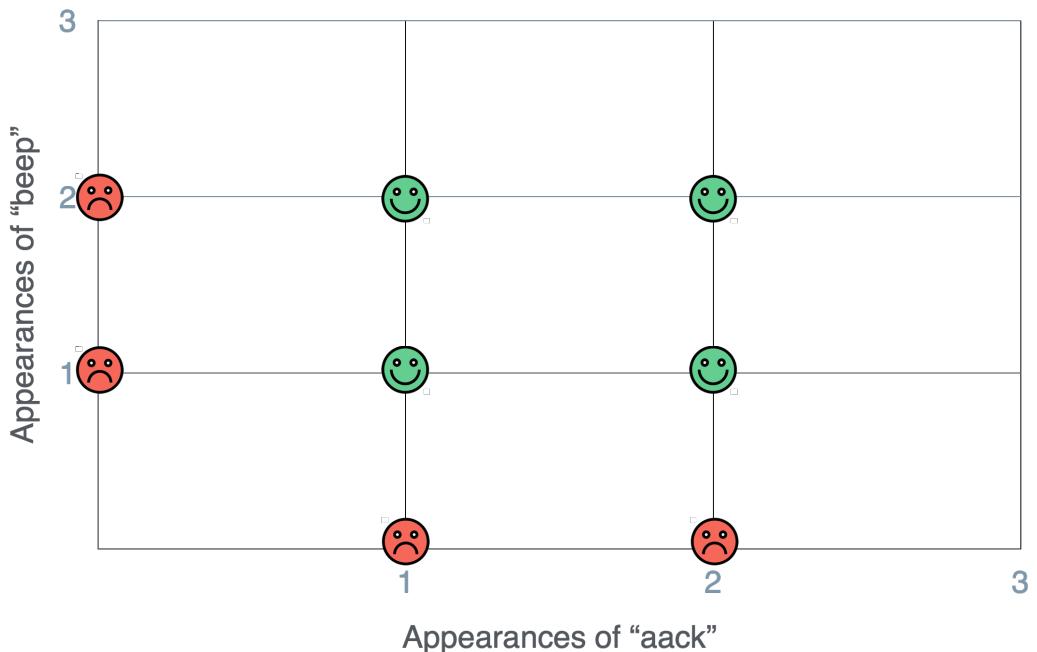
In this chapter we will continue with the example from Chapters 4 and 5, when we learned the perceptron algorithm and logistic regression. The scenario was the following: We find ourselves in a distant planet populated by aliens. They seem to speak a language formed by two words, ‘aack’ and ‘beep’, and we want to build a machine learning model that helps us determine if an alien is happy or sad based on the words they say. This is called sentiment analysis, as we need to build a model to analyze the sentiment of the aliens. We record some aliens talking, and manage to identify by other means if they are happy or sad, and we come up with the dataset in Table 8.1.

**Table 8.1.** Our dataset, in which each row represents an alien. The first column represents the sentence they uttered. The second and third column, the number of appearances of each of the words in the sentence. The fourth column represents the alien's mood.

Sentence	Aack	Beep	Mood
“Aack”	1	0	Sad

“Aack aack”	2	0	Sad
“Beep”	0	1	Sad
“Beep beep”	0	2	Sad
“Aack beep”	1	1	Happy
“Aack aack beep”	1	2	Happy
“Beep aack beep”	2	1	Happy
“Beep aack beep aack”	2	2	Happy

This looks like a nice enough dataset, and we should be able to fit a classifier to this data. Let's plot it first.



**Figure 8.2.** The plot of the dataset in Table 8.1. The horizontal axis corresponds to the number of appearances of the word ‘aack’, and the vertical axis to the number of appearances of the word ‘beep’. The happy faces correspond to the happy aliens, and the sad faces to the sad aliens.

At first glance, it looks like we won’t be able to fit a linear classifier to this data. If you try to draw a line that splits the happy and the sad faces apart, you won’t be able to. What can we do, then? We’ve learned other classifiers that can do the job, such as naive Bayes (chapter 6) or decision trees (chapter 7). But in this chapter we will stick with perceptrons and logistic regression. The reason for this is because we’ll be able to concatenate these simple classifiers and form complex architectures, which will give rise to much stronger classifiers. If our goal is to separate the points in Figure 8.2, clearly one line won’t do it. What is better than one line? I can think of two things:

1. Two lines.
2. A curve.

These two are examples of neural networks. Let me show you why the first one, a classifier using two lines, is a neural network.

### 8.1.1      **Solution - If one line is not enough, use two lines to classify your**

## dataset

Let's explore the classifier that uses two lines to split the dataset in Figure 8.1. There are many ways to draw two lines to split this dataset, and I've drawn one in Figure 8.2. I've called the two lines Line 1 and Line 2.

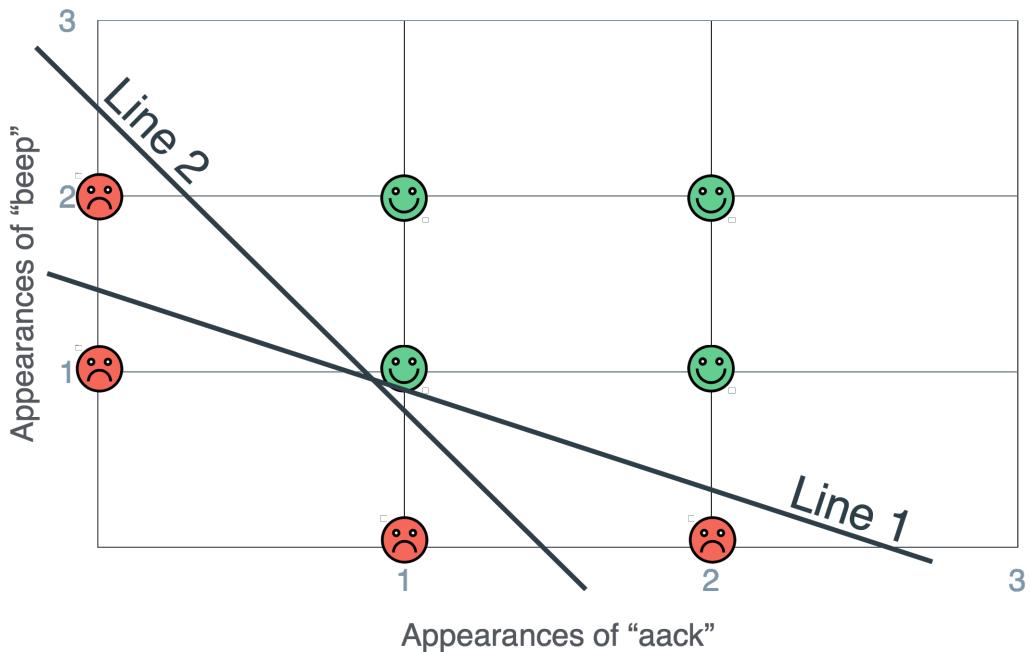


Figure 8.3. The happy and the sad points in our dataset cannot be divided by one line. However, drawing two lines separates them well. Combining linear classifiers this way is the basis for neural networks.

We can now simply define our classifier as follows:

**Classifier:** For this problem, a sentence is classified as happy if its corresponding point is above the two lines in Figure 8.X. If it is below at least one of the lines, it is classified as sad.

Now, let's throw in some math. Can you help me think of two equations for these lines? Many equations would work, but I've thought about the following two (where #aack is the number of times the word 'aack' appears in the sentence, and #beep is the number of time 'beep' appears.):

- **Line 1:**  $6 * (\#aack) + 10 * (\#beep) = 15$
- **Line 2:**  $10 * (\#aack) + 6 * (\#beep) = 15$

**HOW DID I FIND THESE EQUATIONS?** Notice that Line 1 passes through the points (0, 1.5) and (2.5, 0). Therefore, the slope, defined as the change in the horizontal axis divided by the change in the vertical axis is precisely  $-2.5/1.5 = -5/3$ . The y-intercept, namely, the height at which the line crosses the vertical axis is 1.5. Therefore, the equation of this line is  $\#beep = -5/3 * (\#aack) + 1.5$ . By manipulating this equation, we get  $6*(\#aack) + 10*(\#beep) = 15$ . We can take a similar approach to find the equation for Line 2.

Therefore, our classifier becomes the following:

**Classifier:** A sentence is classified as happy if both of the following two inequalities hold.

- **Inequality 1:**  $6*(\#aack) + 10*(\#beep) \geq 15$
- **Inequality 2:**  $10*(\#aack) + 6*(\#beep) \geq 15$

If at least one of them fails, then the sentence is classified as sad.

As a consistency check, Table 8.2 has the values of each of the two equations. At the right of each equation we check if the equation's value is larger than or equal to 15. The right-most column checks if both values are larger than or equal to 15.

**Table 8.2. The same dataset as in Table 8.1, but with some new columns corresponding to the two linear equations. For each linear equation, we check if each of the data points gives a value larger than or equal to 15. Finally, we check if both data points gave us a value larger than or equal to 15. This is our classifier.**

Sentence	Aack	Beep	Equation 1	$Eq\ 1 \geq 15?$	Equation 2	$Eq\ 2 \geq 15?$	$Both\ eqs.\ \geq 15$
“Aack”	1	0	6	no	10	no	no
“Aack aack”	2	0	12	no	20	yes	no
“Beep”	0	1	10	no	6	no	no
“Beep beep”	0	2	20	yes	12	no	no
“Aack beep”	1	1	16	yes	16	yes	yes
“Aack aack beep”	1	2	26	yes	22	yes	yes
“Beep aack beep”	2	1	22	yes	26	yes	yes

"Beep aack beep aack"	2	2	32	yes	32	yes	yes
--------------------------	---	---	----	-----	----	-----	-----

Note that the right-most column in Table 8.2 (yes/no) coincides with the right-most column in Table 8.1 (happy/sad). This means the classifier managed to classify all the data correctly.

### 8.1.2 Why two lines? Is happiness not linear?

In Chapters 4 and 5 we managed to infer things about the language based on the equations in the classifiers. For example, if the weight of the word 'aack' was positive, we concluded that it was likely a happy word. What about now? Could we infer anything about the language in this classifier that contains two equations?

The way I like to think of two equations is that maybe on the aliens' planet, happiness is not a simple linear thing, but is instead based on two things. In real life, happiness can be based on many things: it can be based on having a happy family life, combined with a fulfilling career, and food on the table. It could be based on having coffee and also a donut. In this case, let's say that the two aspects of happiness are family life and career. Thus, for an alien to be happy it needs *both*: a happy family life and a happy career life.

It turns out that in this case, both family happiness and career happiness are simple linear classifiers, and each one is described by one of the two lines. Let's say that line 1 corresponds to family happiness, and line 2 to career happiness. Thus, we can think of alien happiness as the diagram in figure 8.X. In this diagram, family happiness and career happiness are joined by an AND operator, which checks if both are true. If they are, then the alien is happy. If any of them fails, it is unhappy.

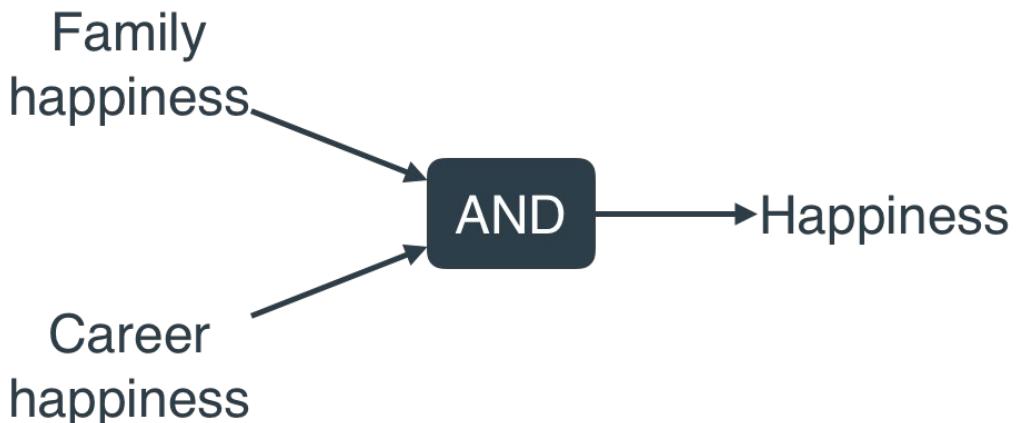


Figure 8.4. The happiness classifier is formed by the family happiness classifier, the career happiness classifier, and an AND operator. The way this works is, if both the family and career happiness classifiers output a 'yes', then so does the happiness classifier. If any of them outputs a 'no', then the happiness classifier also outputs a 'no'.

That is starting to look like a neural network. Just a little bit more math and we'll get to something looking much more like figure 8.1 at the beginning of the chapter. But don't worry, the way we're approaching this will show you step-by-step how we reach that figure, so you understand it better.

### 8.1.3 Perceptrons and how to combine them

In this section I teach you the concept of a perceptron. Actually, this is something we already know. I'll only show you a different image of it. If you recall, in chapter 4 we learned the *perceptron algorithm*, but somehow the word perceptron never came up. The perceptron is simply a way to represent the classifiers we learned in chapters 4 and 5.

In this chapter we have seen two classifiers that are very similar to those in chapter 4: The family happiness and the career happiness classifier. Let's study them more carefully. We'll start with the family happiness classifier. It was given by Line 1, which had the equation

$$6 * (\#aack) + 10 * (\#beep) \geq 15$$

We can rewrite this classifier in the language of scores and thresholds as follows:

#### **Classifier 1 (family happiness)**

##### **Scores:**

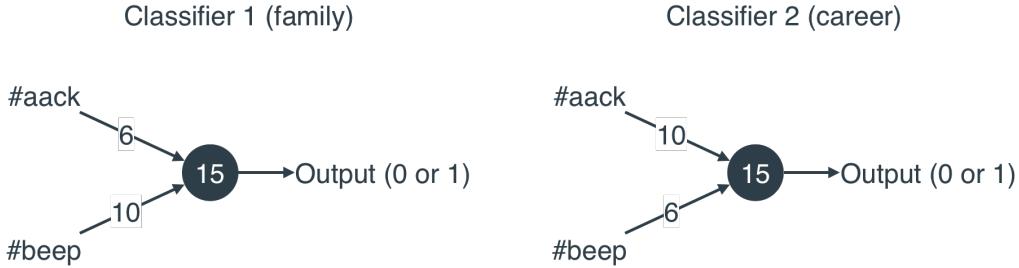
- Aack: 6 points
- Beep: 10 points
- Threshold: 15

##### **Rule:**

Add the scores of all the words.

- If the score is larger than or equal to the threshold of 15, predict that the alien has a happy family life.
- If the score is smaller than the threshold of 15, predict that the alien doesn't have a happy family life.

We can represent this graphically as in Figure 8.x (left), by a node with two inputs and an output. The inputs correspond to the scores of the words, and the output corresponds to the prediction. More specifically, on the edges coming in, we write these scores. Inside the node, we write the value of the threshold.



**Figure 8.5.** In order to build neural networks, we draw linear classifiers like a perceptron. In this image you can see the perceptrons for the family and the career happiness classifiers.

The input to this perceptron is a pair of numbers (a, b), where a is the number of appearances of 'aack' and b is the number of appearances of 'beep' in the corresponding sentence. The output is 0 or 1, depending on the prediction. If the prediction was a happy family life, the output is 1, otherwise it is 0.

In chapter 5 we learned activation functions, in particular the step function, which returns a 1 if the input is positive or zero, and a 0 if the input is negative. In this case, we can write the output of the classifier as the following:

$$\text{Output} = \text{Step}(6 * (\#aack) + 10 * (\#beep) - 15).$$

In a similar way, we can see the career happiness classifier as the following:

### **Classifier 2 (career happiness)**

#### **Scores:**

- Aack: 10 points
- Beep: 6 points
- Threshold: 15

#### **Rule:**

Add the scores of all the words.

- If the score is larger than or equal to the threshold of 15, predict that the alien has a happy career life.
- If the score is smaller than the threshold of 15, predict that the alien doesn't have a happy career life.

The diagram of this perceptron is in the right of figure 8.x, and its output is given by the following formula:

$$\text{Output} = \text{Step}(10 * (\#aack) + 6 * (\#beep) - 15)$$

Now, for the sake of clarity, let's make a table with the outputs of these two classifiers (table 8.3).

**Table 8.3.** The same dataset as in table 8.1, with two more columns corresponding to the results of the family and career happiness classifiers.

Sentence	Aack	Beep	Family Happiness	Career Happiness
“Aack”	1	0	0	0
“Aack aack”	2	0	0	1
“Beep”	0	1	0	0
“Beep beep”	0	2	1	0
“Aack beep”	1	1	1	1
“Aack aack beep”	1	2	1	1
“Beep aack beep”	2	1	1	1
“Beep aack beep aack”	2	2	1	

Now, brace yourself, because here is where the neural network gets built. Let's look at Table 8.3, and completely forget about the first three rows. That only leaves the last two rows, namely, career and family happiness. Let's say that someone gave us that as our original dataset, where the features are family and career happiness, and asked us to predict the label, which is happiness, based on that dataset. Can you predict the label only from those two features? Namely, can you fit a linear classifier to the data in Table 8.4?

**Table 8.4.** The results of the AND operator applied to the results of the family and career happiness classifiers.

Family Happiness (feature)	Career Happiness (feature)	Happiness (label)
0	0	0
0	1	0
0	0	0

1	0	0
1	1	1
1	1	1
1	1	1
1	1	1

That seems like an easy Chapter 4 problem. I can think of many classifiers that work, and here is one of them:

### Classifier 3 (happiness)

#### Scores:

- Family happiness: 1 point
- Career happiness: 1 point
- Threshold: 1.5

#### Rule:

Add the scores of all the words.

- If the score is larger than or equal to the threshold of 1.5, predict that the alien has a happy career life.
- If the score is smaller than the threshold of 1.5, predict that the alien doesn't have a happy career life.

Notice that this classifier works because, since the features are only 0 and 1, the only way to achieve a sum of features that is greater than or equal to 1.5 is when both features are 1. This is the mathematical way to say that the only way to achieve happiness is when you have both family happiness and career happiness.

The formula for the output of this classifier is given by

$$\text{Output} = \text{Step}(1 * (\text{Family happiness}) + 1 * (\text{Career happiness}) - 1.5)$$

Where (Family happiness) and (Career happiness) are the outputs of classifiers 1 and 2, respectively. This perceptron classifier is illustrated by the diagram in Figure 10.x.

## Classifier 3 (happiness)

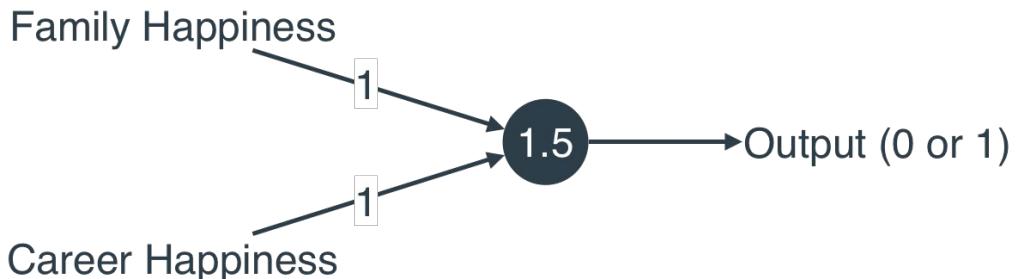


Figure 8.6. Perceptrons can simulate logical operators, and this is very useful for us. In this case, we write the AND operator corresponding to the happiness classifier as a perceptron.

Since the outputs of the family and career classifiers are used as inputs to the happiness classifier, it is natural to plug in the right end of the family and career happiness perceptrons into the happiness perceptron, obtaining the diagram in Figure 8.x.

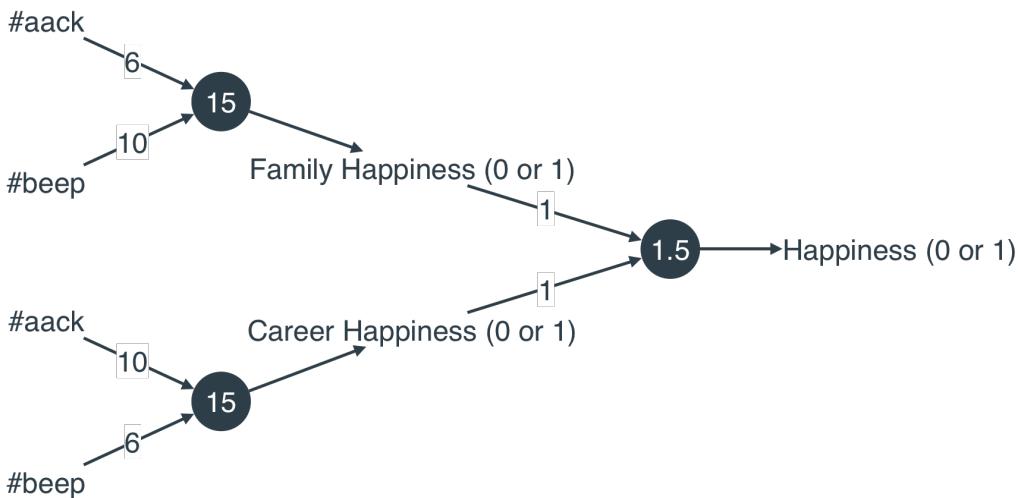


Figure 8.7. Joining perceptrons together is how we start building our neural networks. We do this by plugging in the outputs of the family and career happiness classifiers as inputs into the happiness classifier (the AND operator).

Notice that Figure 8.x has a lot of redundancy. For instance, the two inputs appear repeated at the very left. If we clean up this figure a bit, we obtain Figure 8.x.

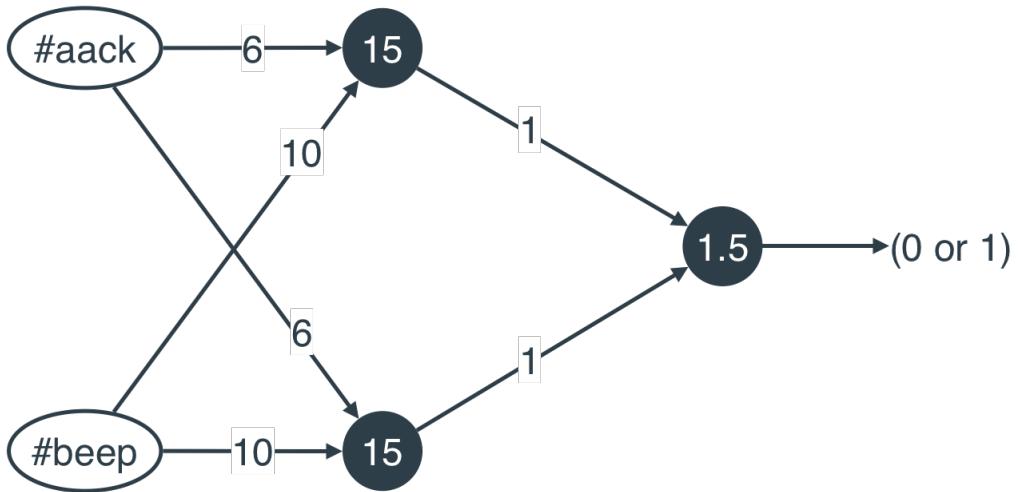


Figure 8.8. We clean up Figure 8.7 a bit, as to not have repeated inputs, and we get our very first neural network!

Now, Figure 8.x looks a lot more like the neural networks we are used to seeing (such as the one in Figure 8.1). Neural networks are also called *multilayer perceptrons*, or MLPs.

Another way I like to see neural networks is by plotting the boundaries in every one of the classifiers, as Figure 8.x shows.

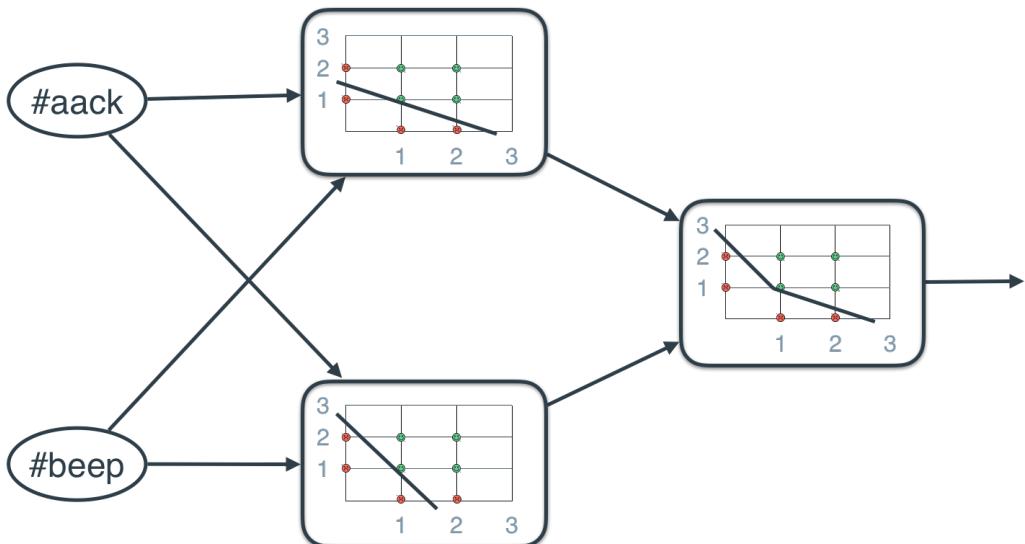


Figure 8.9. Another way to see the perceptrons is by their linear boundary. In this way, the family and career

happiness classifiers look like lines, and the happiness classifier looks like a broken line, formed by the two previous ones.

This was a very small example. In the next section we'll study more about the general architecture of neural networks.

### 8.1.4 From discrete perceptrons to continuous perceptrons - a trick to improve our training

In Chapter 4 we learned discrete perceptrons. Then in Chapter 5, we moved on to continuous ones. The only thing we changed was the activation function, from the step function to the sigmoid function. If you need a refresher, please check out Section 5.11. In the section 8.1.3 we developed perceptrons with step functions, which output a 0 or a 1. We'll call those *discrete perceptrons*. In this section, we'll develop continuous perceptrons, which use the sigmoid function to output any number between 0 and 1. The development is the same, except the equation that defines the output is now the following (for classifier 1, the family happiness classifier):

$$\text{Output} = \sigma(6 \times (\#aack) + 10 \times (\#beep) - 15).$$

Now you may be thinking, what does this output mean? In the previous section, the output was 1 if the classifier thought that the alien had a happy family life, and 0 if it think it didn't. In this case, the classifier outputs a number between 0 and 1 that can be taken as a percentage. This is the percentage of family happiness that the classifier predicts that that alien has.

Let's look at a simple calculation as an example. We'll calculate the output for the sentence "Aack beep", where #aack = 1 and #beep = 1. This output is

$$\text{Output} = \sigma(6 \times (1) + 10 \times (1) - 15) = \sigma(1) = 0.73.$$

The output is 0.73, which means that the family happiness classifier predicts that this alien is (family) happy with a probability of 73% (or that the alien is 73% family happy, however you prefer to interpret it). Notice that the discrete classifier predicted that the alien was happy, and the continuous one predicted that it was happy with a probability of 73%, which is bigger than 50. In general, if a discrete classifier predicts a yes, the same continuous classifier will predict an answer that is larger than or equal to 50%.

Similarly, the equation for classifier 2, the career happiness classifier, is

$$\text{Output} = \sigma(10 \times (\#aack) + 6 \times (\#beep) - 15),$$

and that of classifier 3, the happiness classifier, is

$$\text{Output} = \sigma(1 \times (\text{Career}) + 1 \times (\#beep) - 1.5),$$

where Career and Family are the outputs of classifiers 1 and 2, respectively.

If we fill in the table with these values, we get Table 8.5. Notice that this table also predicts the data in Table 8.1 correctly, since it assigns probabilities lower than 50% for sad aliens (the first 4), and higher than 50% for the happy aliens (the last 4).

**Table 8.5.** A modification of the results in Tables 8.3 and 8.4. In here, we use the sigmoid activation function to get values between 0 and 1 as our outputs.

Sentence	Aack	Beep	Career Happiness	Family Happiness	Happiness
“Aack”	1	0	0.000	0.007	0.183
“Aack aack”	2	0	0.047	0.993	0.387
“Beep”	0	1	0.007	0.000	0.183
“Beep beep”	0	2	0.993	0.047	0.387
“Aack beep”	1	1	0.731	0.731	0.491
“Aack aack beep”	1	2	1.000	0.999	0.622
“Beep aack beep”	2	1	0.999	1.000	0.622
“Beep aack beep aack”	2	2	1.000	1.000	0.622

Notice that this classifier is not perfect. The sentence “Aack beep” is happy, but this classifier gave it a score of 0.491, which is less than 0.5, which equals 50%. This simply means that as a continuous multilayer perceptron, this is not the correct one. We may want to decrease the threshold of 1.5 in the final perceptron to some smaller value for the network to correctly classify this point.

In the same way that the plot of the boundary of the discrete multilayer perceptron was two lines, we can still plot the boundaries of continuous multilayer perceptrons. The output function is quite complicated, since it has sigmoid functions inside other sigmoid functions, but we can still plot its contours (if you are curious, check out the plot in the Github repo in [www.github.com/luisquiserrano/manning](http://www.github.com/luisquiserrano/manning)). In Figure 8.x, we can compare the boundaries of the discrete and the continuous multilayer perceptrons we’ve fit to this dataset.

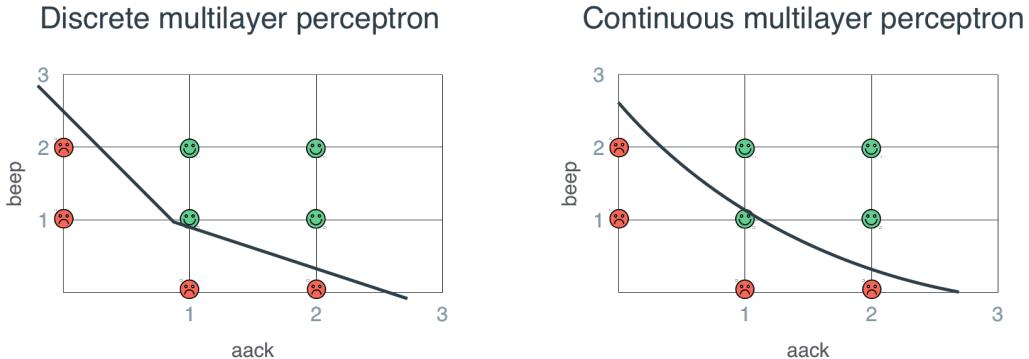


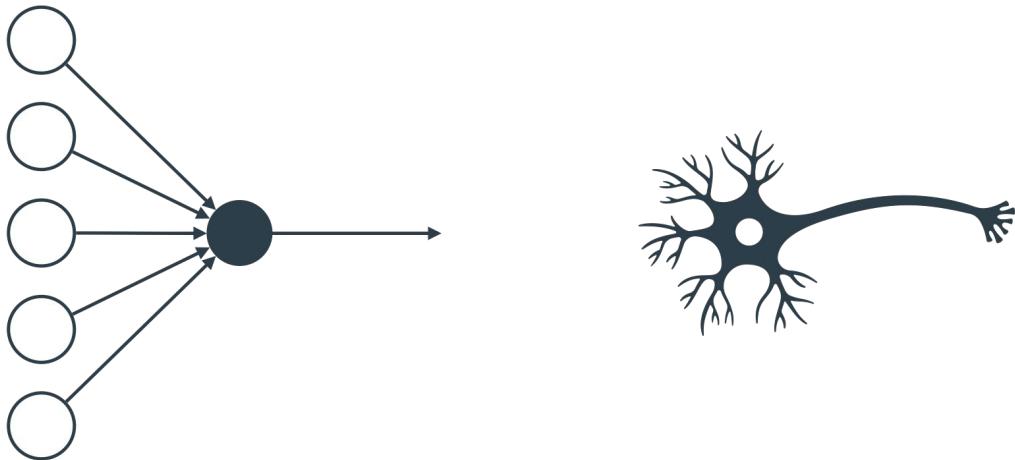
Figure 8.10. The difference between a discrete and a continuous multilayer perceptron. The boundary of a discrete MLP looks like a broken line, whereas the boundary of a continuous MLP looks like a smooth curve.

So that's it! Our first neural network. Of course, neural networks can get much more complex, and that's what we learn in the next section.

In real life, we never actually use discrete multilayer perceptrons. The reason will become clear later, but in short, they are impossible to train. Continuous multilayer perceptrons provide functions with much nicer derivatives, which makes the training easier. Thus, from now on, when I say neural network or multilayer perceptron, we'll assume that it is continuous and uses the sigmoid activation function. Later in the chapter we'll learn some more activation functions, and when that happens, I'll specify which one we use.

## 8.2 The general scenario - Neural networks

First things first. Why are these objects called neural networks? The idea is that in a vague way, neural networks resemble how the brain works. If we look at a perceptron, it resembles a neuron (Figure 8.x). A general perceptron will have several inputs and one output, and the output is 0 or 1 depending on what the inputs are. This is meant to resemble the way a neuron, or human brain cell, works, as the neuron decides to transmit an electrical impulse or not, based on its inputs.



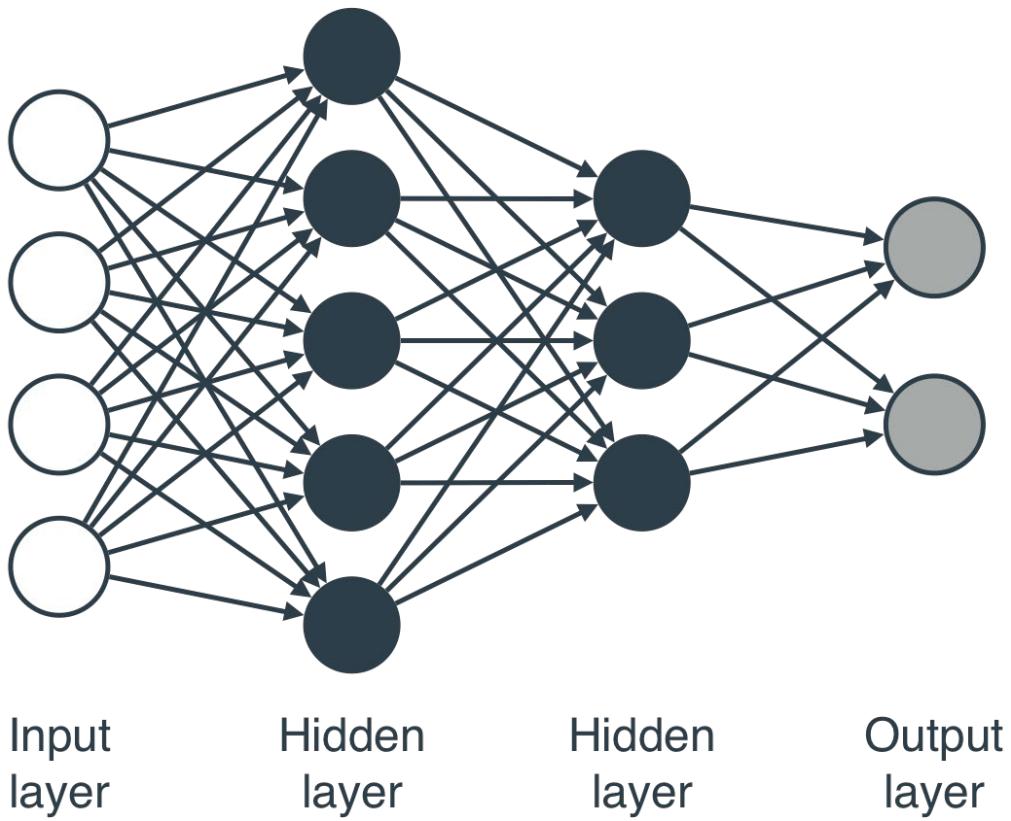
**Figure 8.11.** Perceptrons are inspired by neurons in the brain, as neurons receive the inputs of other neurons in the form of signals, process them, and output another signal.

### 8.2.1 The architecture of a neural network

The neural network we built previously had three layers. The first layer was the input layer, where we entered the features, namely, #aack and #beep. The next layer is the hidden layer, which helps our training. The hidden layers get their name from the fact that we never get to see their outputs, as all we see is the output of the final layer. The last layer is the output layer, which tells us if the sentence is happy or not.

In real life, neural networks tend to be much larger, and the way we build them is by adding many more hidden layers. Figure 8.x shows a larger neural network with two hidden layers. The size of a hidden layer is simply the number of nodes in it. The architecture of this neural network is as follows:

- An input layer of size 4.
- A hidden layer of size 5.
- A hidden layer of size 3.
- An output layer of size 2.



**Figure 8.12.** The architecture of a neural network consists of an input layer, several hidden layers, and an output layer. The input layer is where we input our data, the hidden layers process it, and the output layer is where the predictions come out.

What does it mean to have a hidden layer of size 2 or more? This is a great property of neural networks. We are used to classifiers that return one output, or answer, such as a ‘yes’ or a ‘no’. However, in neural networks we can output as many answers as we want. For example, if we want to train a neural network that will recognize images and tell us if the image contains a dog, cat, bird, or aardvark, we simply make sure the output layer has 4 nodes, one for each animal.

Now, how does this relate to the boundaries that we saw in section 8.1.4? The way I like to picture neural networks in my head is as in Figure 8.x. This network has four layers, an input, an output, and two hidden layers. The first layer is the input. The second layer (or first hidden layer) is formed by linear models, namely, perceptrons. The third layer (or second hidden layer) is a superposition of the linear models in the second layer, and thus, they look like curves. These superpositions depend on the weights and biases coming in from the previous layer. The fourth layer (output) is also formed as a superposition of the nodes in the previous layer, so it looks

like a more complicated curve. Thus, if our dataset is complicated, we can always find a neural network with enough layers and nodes that will form a boundary that splits our data well.

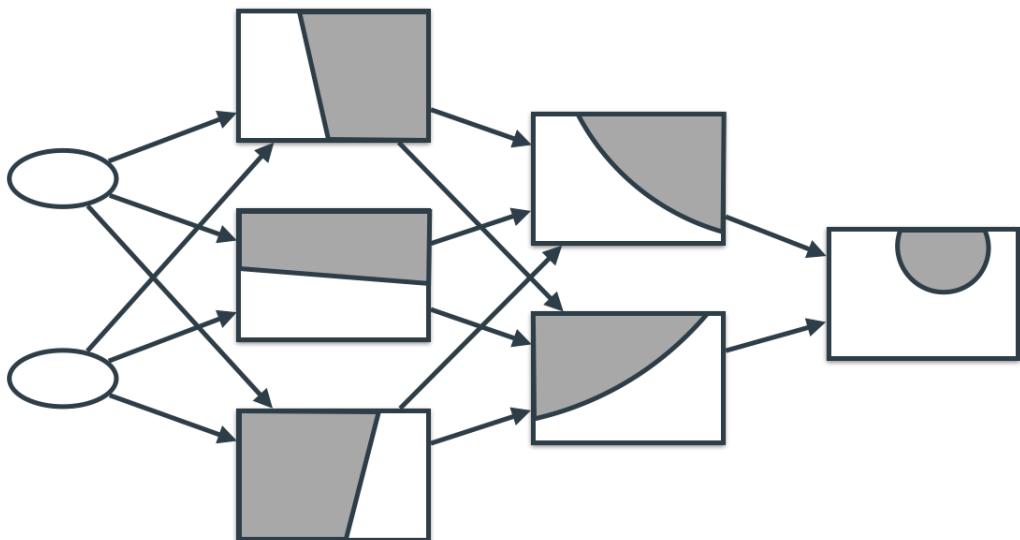


Figure 8.13. A mental picture of neural networks. The first hidden layer is formed by linear classifiers. As we move to the right in the neural network, these linear classifiers are combined to form more complex classifiers with more complicated boundaries.

In real life, there are neural networks with many layers and many many nodes. However, you can always picture them as drawing some boundaries that get more and more complicated as we move across the layers.

### 8.2.2 Bias vs Threshold - Two equivalent ways of describing the constant term in the perceptron

In Section 4.11, we learned the difference between bias and threshold. To refresh your memory, let me show you an example. In Classifier 1, our weights are 6 and 10, and our threshold is 15. This means that for a sentence with the word 'aack' appearing  $a$  times and the word 'beep' appearing  $b$  times, the score is given by  $6a + 10b$ . This is then compared with the threshold, and if the score is larger than or equal to the threshold, then the perceptron outputs a 1. Otherwise, the perceptron outputs a 0. To summarize, these are the parameters of the classifier:

- Score of 'aack': 6
- Score of 'beep': 10
- Threshold: 15

Let's say we want the threshold to always be zero, by convention. In that case, we need to subtract 15 from the score, and compare this score with a zero, in order to get the same effect. We call this -15 the bias. Now, the score is  $6a + 10b - 15$ . We can write the classifier as follows:

- Score of 'aack': 6
- Score of 'beep': 10
- Bias: 15

We can illustrate the bias in a perceptron using a slightly different diagram, as you can see in Figure 8.x. In here, we consider the bias as another input to the perceptron. We have used as the input to this perceptron the variables  $x_1$  and  $x_2$  (instead of #aack and #beep), and as you can see, the input corresponding to the bias is simply a 1.

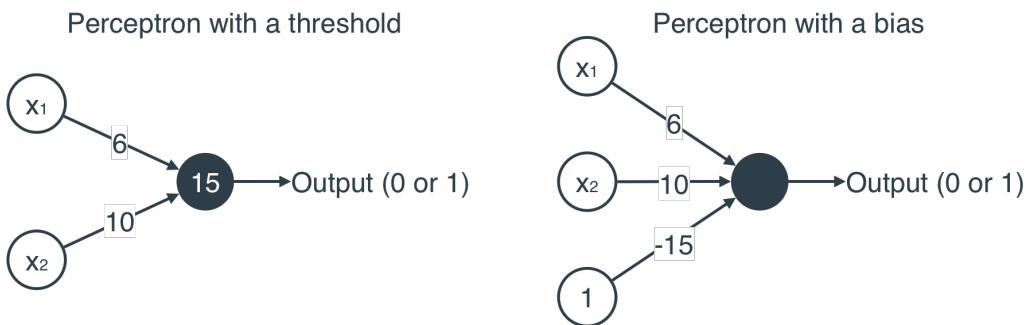


Figure 8.14.

Left: A perceptron with a threshold. The value of the threshold goes inside the node.

Right: The same perceptron with a bias unit. The value of the bias comes in as the weight of the bias node.

Both of these notations are used in the literature, but the one in the right is more common.

These two notations are common in the literature, but as you read papers in deep learning, you'll find that the notation at the right of Figure 8.x is more common.

### 8.3 Training neural networks

We've seen how neural networks look in general, and that they're not as mysterious as they sound like. Now you may be asking, how do I train one of these monsters? Well, in the theory, the process is not terribly complicated (although it may be computationally expensive, but there are ways to speed it up). In this section we'll take a closer look at how to train neural networks. The training is not very different from other algorithms such as linear or logistic regression: First

we define an error function, then we start with random weights, and finally we iterate over these weights many times in order to reduce the error function.

### 8.3.1 Error function - A way to measure how our neural network is performing

The training process in most machine learning models makes use of an error function which constantly informs us how the model is doing at every step. This is the case for neural networks as well. The error function we use here is actually very similar from the one we used for logistic regression in chapter 5; in fact, it is the same log loss function we defined in Section 5.11. This function is defined as follows:

- If the label is 0:
  - $\text{log loss} = -\ln(1 - \text{prediction})$
- If the label is 1:
  - $\text{log loss} = -\ln(\text{prediction})$ .

These can be summarized into one formula as:

$$\text{log loss} = (-\text{label}) \ln(\text{prediction}) - (1-\text{label}) \ln(1 - \text{prediction})$$

Why is this a good loss function? Well, all we need for a loss function to be good is that if the prediction is close to the label, the loss is small, and if the prediction is far from the label, the error (loss) is large. Here are some examples:

- **Example 1:** Label close to the prediction, so the error is small.

Label = 0

Prediction = 0.1

$$\text{log loss} = -\ln(1-0.1) = -\ln(0.9) = -0.105$$

- **Example 2:** Label far from the prediction, so the error is large.

Label = 0

Prediction = 0.9

$$\text{log loss} = -\ln(1-0.9) = -\ln(0.1) = -2.303$$

- **Example 3:** Label far from the prediction, so the error is large.

Label = 1

Prediction = 0.1

$$\text{log loss} = -\ln(0.1) = -2.303$$

- **Example 4:** Label close to the prediction, so the error is small.

Label = 1

Prediction = 0.9

$$\text{log loss} = -\ln(0.9) = -0.105$$

### 8.3.2 Backpropagation - The key step in reducing the error function in order to train the neural network

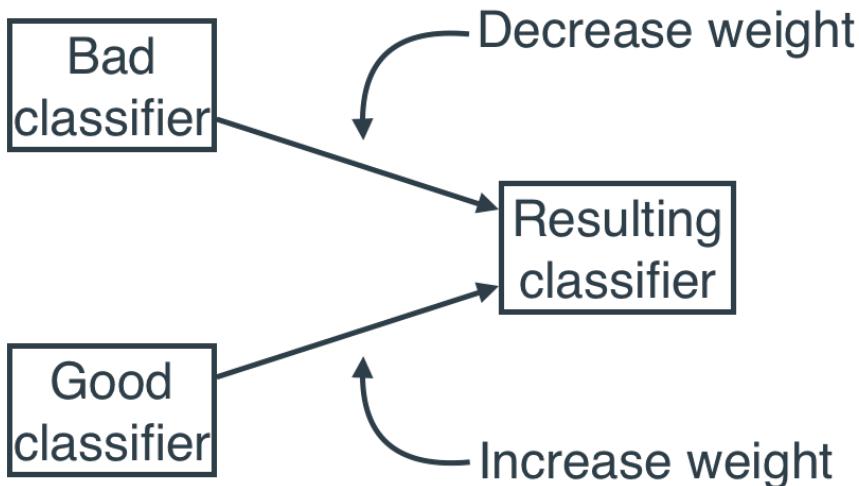
In this section I will show you the most important process of the training of a neural network, called backpropagation. Recall that in Chapters 3, 4, and 5 (linear regression, perceptron algorithm, and logistic regression), we used gradient descent to train our models. To refresh your memory, these were the main steps of the gradient descent method:

**Gradient descent:**

1. Start with random weights.
2. Repeat many times:
  - a) Calculate the loss function.
  - b) Take a small step in the direction of the gradient in order to decrease the loss function by a small amount.
3. The weights you obtain correspond to a neural network that (hopefully) fits the data well.

When we talk about the gradient of the loss function, we are thinking of a derivative. The loss function of a neural network is complicated, since it involves the logarithm of the prediction, and the prediction itself is a complicated function (it contains several linear equations and sigmoids). Furthermore, we need to calculate the derivative with respect to many variables, one corresponding to each of the weights and biases of the neural network. Calculating these derivatives is out of the scope of this book, but if you want to work out the math, I encourage you to look for 'backpropagation' in any machine learning book or tutorial, and you'll find the process.

However, can we think of a visual way to understand backpropagation? In Chapters 3, 4, and 5, we managed to explain gradient descent in a simple manner, such as moving a line closer to a point or farther from it. With neural networks, this is more complicated, but we can still find a mental picture. Every perceptron on the neural network receives some input and processes some output. This input comes from other classifiers. Some of the classifiers may classify our data well, others may classify it poorly. The output of each of these classifiers gets multiplied by a weight (these are the weights that we train during the backpropagation process). The backpropagation process simply increases the weights of the classifiers that train our data well, and decreases the weights of those who train our data poorly.



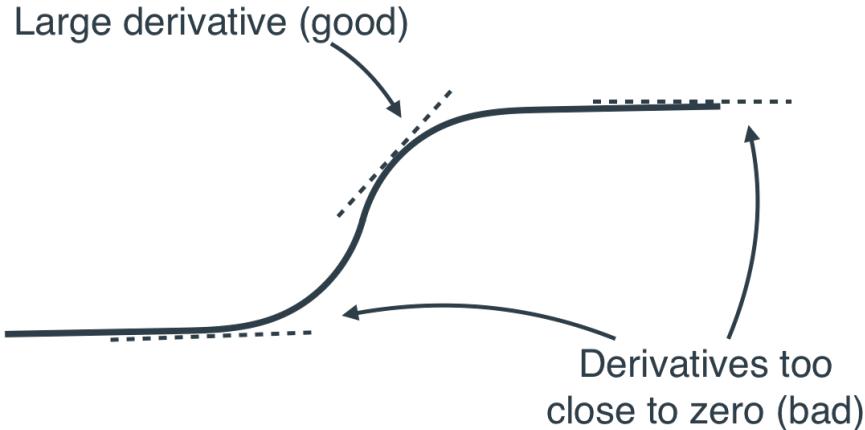
**Figure 8.15.** A mental picture of backpropagation. At each step of the training process, the weights of the edges get updated. If a classifier is good, its weight gets increased by a small amount, and if it is bad, its weight gets decreased.

What about the bias? I see the bias as a classifier too, only one that returns a 1 as a prediction for each of the points. Therefore this one can also be considered a good or bad classifier, depending on the labels of the data, and its weights can be trained accordingly.

### 8.3.3 Potential problems with neural networks - From overfitting to vanishing gradients

In practice, neural networks work very well. But as you can imagine, due to their complexity, many problems arise with their training. Luckily, we can put a solution to the most pressing ones. One problem that neural networks have is overfitting, as really big architectures can potentially memorize our data without generalizing it well. Another problem they can have is vanishing gradients, in which the values used to update the weights get so small, that training gets stuck. In Sections 8.2.3 and 8.2.4 I'll explain these problems more in detail and show you some solutions to these problems.

Another serious problem that neural networks show is vanishing gradients. Notice that the sigmoid function is very flat on the ends, which means that the derivatives (tangents to the curve) are too flat. This means their slopes are very close to zero.



**Figure 8.16.** The sigmoid function is flat at the ends, which means that for large positive and negative values, its derivative is very small, hampering the training.

During the backpropagation process, we compose many of these sigmoid functions (which means we plug in the output of a sigmoid function as the input into another sigmoid function repeatedly). As you can imagine, this composition make the derivatives very close to zero, which means the steps taken during backpropagation are tiny. Recall that the gradient descent process is like descending from a mountain. If you try to descend from a mountain taking very tiny steps, the process will take a very very long time.

There are several solutions to the vanishing gradient problem, and so far one of the most effective ones is to try different activation functions. We'll elaborate on this in Section 8.2.5.

### 8.3.4 Techniques for training your neural network - Dropout, regularization

As I mentioned before, neural networks are very prone to overfitting. In this section I'll teach you some techniques to decrease the amount of overfitting during the training of neural networks.

The first question you may have is: How do I pick the correct architecture? This is a very difficult question, and there is no concrete answer. The rule of thumb is to err on the side of picking a larger architecture than you may need, and then to apply techniques to reduce the amount of overfitting that your network may have. In some way it is like picking a pair of pants, where the only choices you may have are too small or too big. If you pick too small, there is not much you can do. On the other hand, if you pick too big, you can wear a belt to make them fit better. It's not ideal, but it's all we have for now. Picking the correct architecture based on the dataset is an area of research, and you may be able to contribute to it!

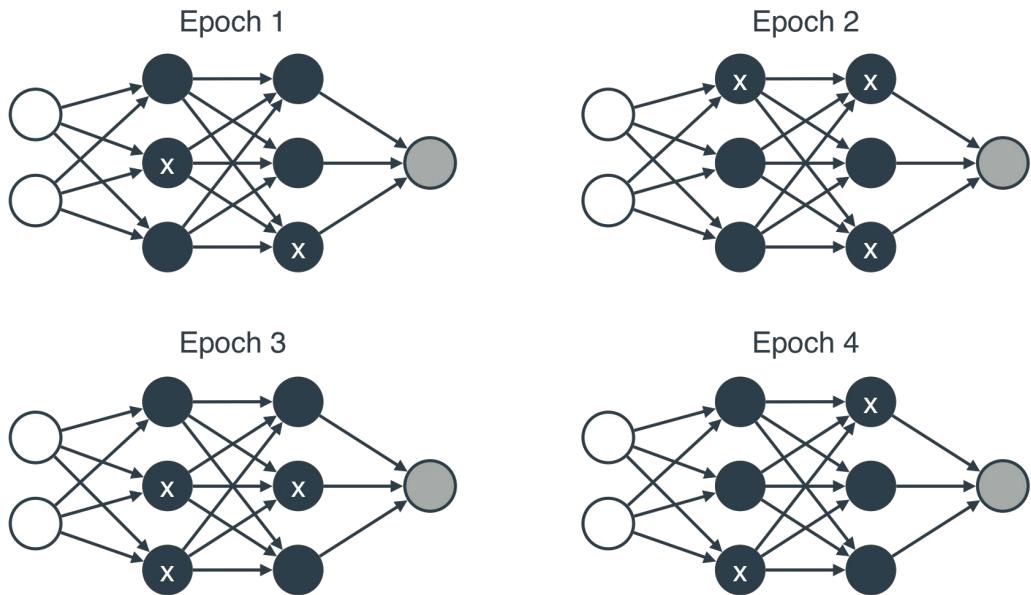
### **REGULARIZATION - A WAY TO REDUCE OVERFITTING BY PUNISHING HIGH WEIGHTS**

As we've learned in this book, L1 and L2 regularization are techniques that we can use to decrease overfitting in many algorithms such as linear and logistic regression, and neural networks are not the exception. The way one applies regularization in neural networks is the same as one would apply it in linear and logistic regression, by adding an extra term to the loss function, so we won't elaborate it any further in this chapter.

### **DROPOUT - A WAY TO MAKE SURE A FEW STRONG NODES ARE NOT DOMINATING THE TRAINING**

Dropout is a very interesting technique used to reduce overfitting in neural networks, and to understand it I like to think of the following analogy. Let's say that we are right-handed, and we like to go to the gym. After some time we start noticing that after a while in the gym, our right bicep is growing a lot, but our left one is not growing at all. We then start paying more attention to our training, and realize that since we are right-handed, we tend to always pick up the weights with the right arm, and not allow the left arm to do much exercise. We decide that enough is enough, so we take a drastic measure. Sometimes when we go to the gym, we'll tie our right hand to our back, and force ourselves to do our routine without the right arm. After this, we start seeing that the left arm starts to grow, as desired. Then, in order to work both arms, this is what we do: Every day before the gym, we flip two coins, one for each arm. If the left coin falls in heads, we tie the left arm to our back, and if the right arm falls in heads, we tie the right arm to our back. Some days we'll work with both arms, some day with only one, and some days with none (those are leg days, perhaps). The randomness of the coins will make sure that on average, we are working both arms mostly equally.

Dropout uses this, except instead of arms, we train the weights in the neural network. One problem neural networks have is that if a neuron has very large weights, that neuron dominates the prediction step and drowns the smaller neurons around it, not allowing them to train properly. The dropout process attaches a small probability  $p$  to each of the neurons. Thus, in each epoch of our training process, it removes that neuron with probability  $p$ , and trains the network with the remaining ones. This means that in every epoch, the neural network removes some of its neurons and trains without them, forcing the others one to train. Dropout is used only on the hidden layers, not on the input or output layers. In Figure 8.x the process is illustrated, where throughout 4 epochs of training, some neurons are removed. Note that since each one is removed with a probability, it is not guaranteed that the same number of neurons are removed during each epoch.



**Figure 8.17.** The dropout process. At different epochs, we pick random nodes to remove from the training, in order to give all the nodes an opportunity to update their weights and not have a single node dominating the training.

Dropout has had great success in the practice, and I encourage you to use it every time you train a neural network. The packages that we use for training neural networks make it very easy to use, as you'll see later in this chapter.

### 8.3.5      **Different activation functions - Sigmoid, hyperbolic tangent (tanh), and the rectified linear unit (ReLU)**

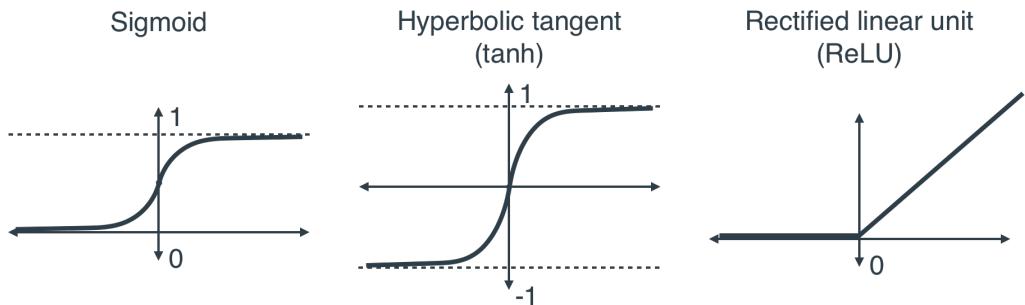
As we saw in Section 8.2.3, the sigmoid function is a bit too flat, which causes problems with vanishing gradients. A solution to this problem is to use different activation functions. In this section I will show you two different activation functions that will be crucial to improve our training process: The hyperbolic tangent (tanh) and the rectified linear unit (ReLU)

#### **HYPERBOLIC TANGENT (TANH)**

The hyperbolic tangent function tends to work better than the sigmoid function in the practice, due to its shape. This one is given by the following formula:

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

Tanh is a bit less flat than sigmoid, but it still has a similar shape, as you can see in Figure 8.x. It provides an improvement over sigmoid, but it still has the vanishing gradient problem.



**Figure 8.18. Three different activation functions.**

Left: The sigmoid function, represented by a greek letter sigma.

Middle: The hyperbolic tangent, or tanh.

Right: The rectified linear unit, or ReLU.

#### **RECTIFIED LINEAR UNIT (RELU)**

A much more popular activation that is commonly used in neural networks is the *Rectified Linear Unit*, or ReLU. This one is very simple, if the input is negative, the output is zero, while if the input is positive, the output is the same input. In other words, if  $x \geq 0$ , the output is  $x$ , and if  $x \leq 0$ , the output is 0.

Relu doesn't have the vanishing gradient problem, since the derivative of it is 1 when the input is positive, and thus, very large neural networks tend to use it very often.

#### **8.3.6 More than one input? No problem, the softmax function is here to help**

So far, neural networks have solved binary classification problems, namely, problems in which our outputs consist of two classes. However, if our data has more than 2 labels, we can do a simple modification to our neural network architecture to allow more than two classes. In this section I show you this modification, which involves adding more neurons in the output layer and slightly modifying the activation function.

Let's look at a small example. We want to build a neural network that recognizes images in a dataset which contains images of dogs, cats, and birds. Our goal is for our network to output information on how likely an image is to be of a dog, cat, or bird. The first question we may ask is, how do we want this output to look? Since we want probabilities, an answer that makes sense is the following. The neural network would output three numbers which add to 1. The first is the probability that the image is of a dog, the second is the probability that it's a cat, and the third is the probability that it's a bird. Thus, if we input the image of a cat, we'd like the neural network to output something like (0.02, 0.95, 0.03), where the second number is the largest.

The key to get the neural network to output this is in the activation function used in the final layer. We can build an architecture which has three neurons in the final layer, each of the outputting a score, which is a real number. Say, for example, that it outputs the numbers 1, 2, and 3. This means that the neural network has given the dog a score of 1, the cat a score of 2, and the bird a score of 3. Let's record that.

- Score(dog) = 1
- Score(cat) = 2
- Score(bird) = 3.

Thus, the image seems to be more likely to be of a bird. However, what are the probabilities? One simple technique to turn numbers into probabilities is to divide by their sum. In this case, the sum is 6, so by dividing everything by 6, we get the following:

- Probability(dog) =  $\frac{1}{6}$
- Probability(cat) =  $\frac{2}{6} = \frac{1}{3}$
- Probability(bird) =  $\frac{3}{6} = \frac{1}{2}$ .

That's not a bad idea, but what if the scores instead were 1, 1, and -2? In this case, their sum is 0, and we know we can't divide by 0. Can we slightly modify the original idea so that this won't happen? Ideally, we'd like to remove negative scores, but still keep an order in them in such a way that high scores correspond to high probabilities, and low scores to low probabilities. Is there any function which respects this order, yet always outputs a positive number? I can think of one, the exponential function! What if we raise e to the power of the scores?

- $e^{\text{Score(dog)}} = e^1 = 2.718$
- $e^{\text{Score(cat)}} = e^2 = 7.389$
- $e^{\text{Score(bird)}} = e^3 = 20.086$ .

Now if we divide the three of them by their sum, which is 30.193, we get the following probabilities:

- Probability(dog) = 0.090
- Probability(cat) = 0.245
- Probability(bird) = 0.665.

Note that the choice of the base e was arbitrary. We could have also picked 2 or 10. But e is always a good choice for a base of an exponential, since the derivative of the function  $e^x$  with respect to x is precisely  $e^x$ .

This function we have just defined is called the *softmax function*. Note that there is nothing special with the fact that we had three inputs. If we had 100 inputs, we can still do the same, namely, raise e to the power of each of the inputs, calculated the sum of all these results, and then divided each one of them by the sum. Let's define this softmax function in a more formal way.

## SOFTMAX FUNCTION

Given n scores,  $a_1, a_2, \dots, a_n$ , the softmax function will output the n numbers ( $p_1, p_2, \dots, p_n$ ) which add to 1, and correspond to probabilities, where

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

You may ask, what if we used the softmax function for two outputs? Let's say that our dataset is still a dataset of images, only that now it contains pictures of dogs, and things that are not a dog. We assign a label of 1 to the dogs, and 0 to everything else. Let's say that the output of the neural network (before the final activation function) is some number a. For example, say  $a=2$ . That is the score the neural network assigns to the image, and it should be high if the neural network thinks the image is a dog, and low if it thinks it is not. Artificially, let's assign a score of 0 to the label 'not dog'. We have the following:

- Score(dog) = 2
- Score(not dog) = 0

Now, we calculate the softmax.

- Probability(dog) =  $\frac{e^2}{e^2 + e^0}$
- Probability(not dog) =  $\frac{e^0}{e^2 + e^0}$

Remembering that  $e^0 = 1$ , we get that the probability that the image is a dog is  $\frac{e^2}{e^2 + 1}$ . Multiplying the numerator and the denominator by  $e^2$ , we get that

$$P(\text{dog}) = \frac{1}{1 + e^{-2}} = \sigma(2).$$

The softmax turns precisely into the sigmoid function! Therefore, we can see the softmax as a multi-variable generalization of the sigmoid function.

### 8.3.7 Hyperparameters - what we fine tune to improve our training

Like most machine learning algorithms, neural networks utilize many hyperparameters that we can fine tune to get them to work better. These hyperparameters determine how we do our training, namely, how long do we want the process to go, at what speed, and how do we choose to enter our data into the model. Some of the most important hyperparameters in neural networks are

- Learning rate: the size of the step that we use during our training.
- Number of epochs: The number of steps we use for our training.
- Batch vs mini-batch vs stochastic gradient descent: How many points at a time enter the training process. Namely, do we enter the points one by one, in batches, or all at the same time?

### **LEARNING RATE - THE LENGTH OF THE STEP THAT WE USE DURING OUR TRAINING**

Just like in linear or logistic regression, neural networks use the learning rate as the size of each step that it takes during the training process. This is a very important hyperparameter, because a learning rate that is too big will give very large steps to get to the solution, but it may accidentally miss it. A very small learning rate may be more accurate, but it may take too long to get to the solution.

The learning rate in neural networks works as follows. During the backpropagation process, we calculate the derivative of the loss function with respect to every parameter in the neural network. This gives us a value for every parameter, that we must subtract from it in order to reduce the loss. However, as it is common in machine learning, it is much better to give many small steps rather than a few big ones, and this is why we multiply these values by the learning rate. The values for learning rate in neural are normally very similar to those used in linear or logistic regression, and good values can be found using methods such as grid search. However, finding a good learning rate is an area of research itself. A good rule of thumb to have is that when the training is not going well, decreasing the learning rate can be very helpful.

### **NUMBER OF EPOCHS - THE NUMBER OF STEPS WE USE FOR OUR TRAINING**

The number of epochs is simply the number of times we loop through our data during the training algorithm. Choosing this parameter well is also very important, since we need the model to run for long enough to provide good results, but we also have limits in our time and computational power. Normally, the more computational power we have, the more epochs we should use. However, there are other ways to tell when to stop the training, such as the following:

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

### **BATCH VS MINI-BATCH VS STOCHASTIC GRADIENT DESCENT - HOW MANY POINTS AT A TIME ENTER THE TRAINING PROCESS**

The training process of a neural network can be very slow if our dataset is very large. Luckily, there are tricks to speed this process up. In the same way that we eat a large sandwich by taking small bites, we can train a model on a very large dataset by dividing it into many batches (chosen at random), and applying backpropagation in each one of these batches separately. The size of these batches is also a very important hyperparameter used when training neural networks.

When we train a small network with a small dataset, it is no problem to run all our training data through the network at every epoch. This is called *batch gradient descent*. However, if we have a very large dataset and a complicated architecture, this can be very expensive. A method that speeds this up while not harming the training very much is *stochastic gradient descent*. In stochastic gradient descent, we simply input the data to the neural network one data point at a time, and we update the weights every time.

There are pros and cons to both. Batch gradient descent is more exact, since each step that we take in the training process considers all the data, so the steps are very accurate. Stochastic gradient descent is much faster, but since each step doesn't use the whole dataset, the steps are more chaotic. Moreover, if the data point that we use happens to be an outlier, then this will entail a misstep in our training.

However, there is a happy medium which seems to enjoy the benefits of both batch and stochastic gradient descent, called *mini-batch gradient descent*. This method consists of taking small batches of data, inputting them in the network, and updating the loss function after each mini-batch. Given a good size of the mini-batch, this can make our training fast and accurate. The size of the mini-batch is also a hyperparameter, and it can be picked by experimenting with different ones until our training is good.

### **8.3.8 Can neural networks predict values instead of classes? Yes we can! - Neural networks for regression**

So far in this chapter we have seen neural networks as a classification algorithm. However, neural networks can also be used for regression. The transition from classification to regression is very simple, and it only consists of looking into the output layer.

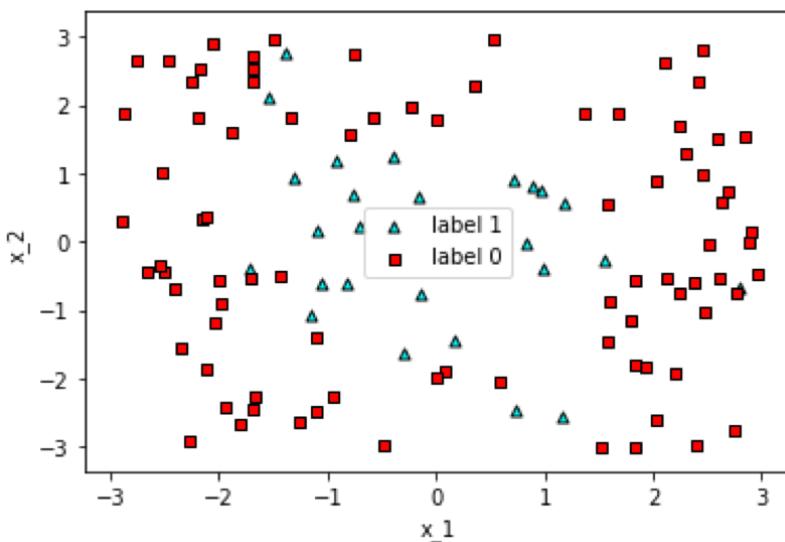
Recall that in classification models, we want our output to be a number between 0 and 1, and in regression models we want our output to be any number. In neural networks, the output layer normally contains a sigmoid layer or a softmax layer, to turn our outputs into numbers between 0 and 1. If we were to remove this function out of the layer, and simply return the scores that were obtained before applying the activation function, then our output can be any number. Thus, we get a regression neural network!

## **8.4 How to code a neural network in Keras**

Now that we've learned how neural networks operate, it's time to train a real neural network! In this section I will show you how to code a neural network in Keras. We will start by loading a sample dataset with two classes. Then we will pre-process this dataset, build a model architecture, and train the model. Finally, we will use the model to make predictions on this dataset.

This code, together with some of its helper functions, is all in the repo for this book, at <http://www.github.com/luisquiserrano/manning>, under Chapter 8.

The dataset we'll use is plotted in Figure 8.x. Notice that this dataset has the triangles (label 1) in the center, and the squares (label 0) outside. This dataset is in the .csv file called 'one\_circle.csv'.



**Figure 8.19.** Neural networks are great for non-linearly separable sets. In order to test this, we'll train a neural network on this circular dataset.

There are many very good packages to code neural networks. Some of the most popular are Keras, TensorFlow, and Pytorch. In this chapter I'll teach you Keras, but I encourage you to check out all of them, as they are very useful and powerful.

Before we train the model, let's look at some random rows in our data. The input will be called X, with features  $x_1$  and  $x_2$ , and the output is called y. Table 8.6 has some sample datapoints. The dataset has 110 rows.

**Table 8.6.**

$x_1$	$x_2$	y
-0.759416	2.753240	0
-1.885278	1.629527	0
...	...	...
0.729767	-2.479655	1
-1.715920	-0.393404	1

### 8.4.1 Categorizing our data - a way to turn categorical features into numbers

Preprocessing data is one of the most important parts of the work of a data scientist. There is a particular preprocessing step which is recommended when training neural networks, called categorizing the data, which I'll show you in this section.

Notice that the output layer has size 2, instead of 1. This is a bit counterintuitive, since we are only trying to predict one label. However, it is good practice to use two outputs in this kind of problems. The first output is the probability that the label is a 0, and the second one is the probability that it is a 1 (so they must add to 1). We could obtain the same results with a layer of size 1 and a sigmoid activation function, and the output will simply be the probability that the label is a 1.

In order for the input to match the output, we also need to turn our labels into two columns, by categorizing (or one-hot encoding) our data. Notice that the labels are one column where every entry is 0 or 1. We'd like to one-hot encode this data, in such a way that the labels are two columns. The labels of these two columns are called  $y_0$  and  $y_1$ , and they are the following.

- If  $y = 0$ , then  $y_0 = 1$  and  $y_1 = 0$ .
- If  $y = 1$ , then  $y_0 = 0$  and  $y_1 = 1$ .

This is easily done with the following command.

```
from tensorflow.keras.utils import to_categorical
y = np.array(to_categorical(y, 2))
```

### 8.4.2 The architecture of a neural network that we'll use to train this dataset

In this section we build the architecture of the neural network for this dataset. We'll use the following four-layer architecture:

- Input layer
  - Size: 2
- First hidden layer
  - Size: 128
  - Activation function: ReLU
- Second hidden layer
  - Size: 64
  - Activation function: ReLU
- Output layer (size = 2)
  - Size: 2
  - Activation function: Softmax

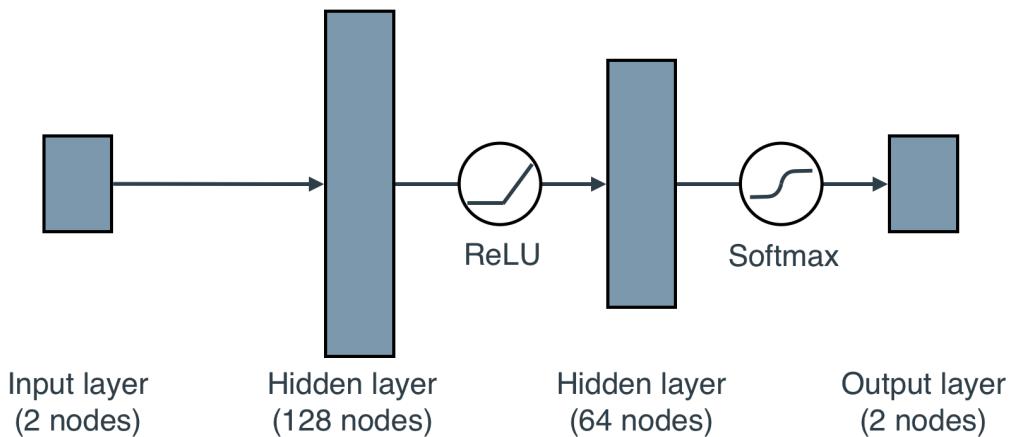


Figure 8.20. The architecture that we will use to classify our dataset. It contains two hidden layers, one of 128 and one of 64 nodes. The activation function between them is a ReLU, and the final activation function is a softmax.

Why this architecture? Why not more layers, or less, or a different number of nodes? There is no answer to that. I normally use layers of size equal to a power of two, such as 64, 128, 256, or 512, and architectures of 2 or 3 layers are common for small datasets like this one. You normally want an architecture that is big enough to handle your data, but not that bit that it uses too many computational resources. I encourage you to try to model this same dataset with different architectures to see what results you get!

Furthermore, we'll add dropout layers in between our hidden layers, to prevent overfitting.

#### 8.4.3 Defining the model in Keras - Number of layers, size of each layer, and activation functions

The next step in training our model is to define the architecture that we'll use. This comprises the following:

- The number of layers.
- The size (number of nodes) of each layer.
- The activation functions used at each layer.
- Other options, such as using dropout in that particular layer during the training.

This can all be done in Keras in only a few lines of code. The first thing we have to do is some useful imports.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation
```

Now, on to define the model with the architecture that we have defined in Section 8.3.1. First, we define the model with the following line::

```
model = Sequential()
```

Now, we proceed to add the hidden layers and the output layer, as follows:

```
model.add(Dense(128, activation='relu', input_shape=(2,))) #A
model.add(Dropout(.2)) #B
model.add(Dense(64, activation='relu')) #C
model.add(Dropout(.1))
model.add(Dense(2, activation='softmax')) #D
```

#A Adding the first hidden layer

#B Adding dropout

#C Adding the second hidden layer

#D Adding the output layer

Once the model is defined, we need to compile it with the following line of code.

```
model.compile(loss = 'categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

What do these all mean? Let me elaborate.

- loss = 'categorical\_crossentropy': This is the loss function, which we have defined as the log loss. Since our labels have more than one column, we need to use the multivariate version for the log loss function, and that is *categorical cross-entropy*.
- optimizer = 'adam': Packages like Keras have many built-in tricks that help us train a model in an optimal way. I won't get into details, but it's always a good idea to add an optimizer to our training. Some of the best ones are Adam, SGD, RMSProp, Adagrad, etc. I encourage you to try this same training with other optimizers, and see how they do!
- metrics = ['accuracy']: As the training goes, we'll get reports on how the model is doing at each epoch. This flag allows us to decide what metrics we want to see during the training, and we've picked accuracy.

When we run the code, we get a summary of our architecture and number of parameters, as follows:

```
Model: "sequential_5"
```

Layer (type)	Output Shape	Param #
<hr/>		
dense_10 (Dense)	(None, 128)	384
<hr/>		
dropout_6 (Dropout)	(None, 128)	0
<hr/>		
dense_11 (Dense)	(None, 64)	8256
<hr/>		
dropout_7 (Dropout)	(None, 64)	0
<hr/>		
dense_12 (Dense)	(None, 2)	130
<hr/>		
Total params: 8,770		
Trainable params: 8,770		
Non-trainable params: 0		

Each row in the previous output is a layer (dropout layers are treated as separate layers for description purposes). The columns correspond to the type of the layer, the shape (number of nodes), and the number of parameters, which is precisely the number of weights plus the number of biases.

#### 8.4.4 Training the model in Keras

In the previous section we defined the model, and now we get to train it. For training, only one simple line of code suffices:

```
model.fit(X, categorized_y, epochs=200, batch_size=10)
```

Let's examine each of the inputs to this fit function.

- X and categorized\_y: The features and labels, respectively.
- epochs: The number of times we run backpropagation on our whole dataset. Here we do it 200 times.
- batch\_size: The length of the batches that we use to train our model. Here we are introducing our data to the model in batches of 10. For a small case dataset like this one, we don't need to input it in batches, but in this example we are doing it for exposure.

As the model trains, it outputs some information at each epoch, namely, the loss (error function) and the accuracy. For contrast, notice how the first epoch has a high loss and a low accuracy, while the last epoch has much better results in both of them.

```
Epoch 1/200
11/11 [=====] - 0s 1ms/step - loss: 0.5906 - accuracy: 0.6273
```

```
Epoch 200/200
11/11 [=====] - 0s 2ms/step - loss: 0.1953 - accuracy: 0.9091
```

The final accuracy of the model is 0.9091, which is pretty good. Now, let's plot the boundary with our plot\_model function to have a visual of how the neural network performed (Fig 8.x).

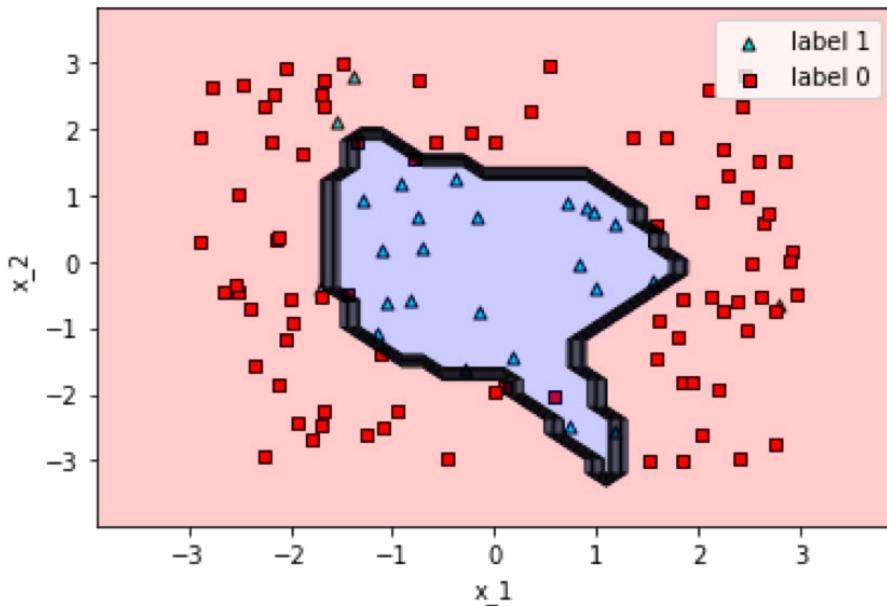


Figure 8.21. The boundary of our neural network classifier.

Note that it managed to capture the data pretty well, encircling the triangles and leaving the squares outside. It has some mistakes, due to noisy data, which is ok. The rigged boundary hints to small levels of overfitting, but in general it seems like a good model.

## 8.5 Other more complicated architectures and some sci-fi applications

Neural networks are useful in many applications, perhaps more so than any other machine learning algorithm currently. One of the most important qualities of neural networks is their versatility. We can modify the architectures in very interesting ways in order to better fit our data and solve our problem. If you want to find out more about these architectures, I highly recommend the book *Grokkering Deep Learning* by Andrew Trask (Manning, 2019).

### 8.5.1 How neural networks see - Image recognition

Neural networks are great with images, and there are many applications where one can use this, such as the following:

- Image recognition: The input is an image, and the output is the label on the image. Some famous datasets used for image recognition are the following:
  - MNIST: Handwritten digits in 28 by 28 gray scale images.

- CIFAR-10: Color images, with 10 labels such as airplane, automobile, etc., in 32 by 32 images.
- CIFAR-100: Similar to CIFAR-10, but with 100 labels such as aquatic mammals, flowers, etc.
- Semantic segmentation: The input is an image, and the output is not only the labels of the things found in the image, but also the location of them. Normally, the neural network will output this location as a bounded rectangle in the image.

You may be wondering, how do we input an image into a neural network? The way we do this is simply by assigning a value of 0 to black pixels and a value of 1 to white pixels. Gray pixels get any value between 0 and 1, depending on their shade. We then turn the image into a long vector, and input that vector into a neural network, as Figure 8.x illustrates.

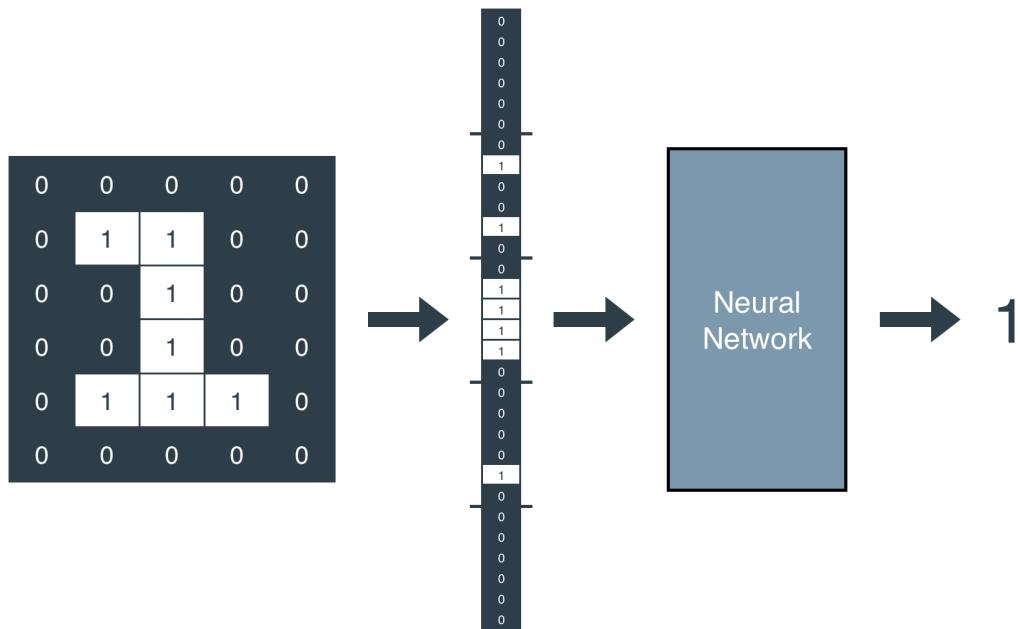


Figure 8.22. Neural networks are great for image recognition. The simplest way we can input an image into a neural network is by reading the values of the pixels in our image as a long vector, and entering this vector into our neural network.

This method works well for simple images, such as handwritten digits as in the MNIST dataset. However, for more complicated images such as pictures, faces, etc., the neural network won't do very well. This is because turning the image into a long vector loses a lot of information. For these complicated images we need different architectures, and this is where convolutional neural networks come to help us.

The details of convolutional neural networks are out of the scope of this book, but imagine the following process. You have a large image that you want to process. You take a smaller window, say 5 by 5, or 7 by 7 pixels, and you swipe it through your large image. Every time you pass it, you apply a formula called a convolution. Thus, you end with a slightly smaller filtered image, which in some way summarizes the previous one. That is a convolutional layer. Your neural network then consists of several of these convolutional layers, followed by some fully connected layers.

When it comes to complicated images, one normally wouldn't go about training a neural network from scratch. A very useful technique called *transfer learning* consists of starting with a pre-trained network, and using your data to tweak some of its parameters (usually the last layer), tends to work very well and at a low computational cost. Networks such as InceptionV3, Imagenet, ResNet, and VGG, have been trained by companies and research groups with large computational power, so I highly recommend you to use them.

### **8.5.2        How neural networks talk - Natural language processing**

One of the most fascinating applications of neural networks is when we can get them to talk to us. This involves listening to what we say or reading what we write, analyzing it, and being able to respond or take action. The ability for computers to understand and process language is called natural language processing. Neural networks have had a lot of success in natural language processing. As you can imagine, there are many aspects to it, and here are some of their main applications:

- Sentiment analysis: Looking at a sentence and predicting if it is positive or negative. Depending on the labels, we can make a model to predict happy/sad sentences, spam/ham emails, or anything we'd like. The problem we saw at the beginning of this chapter was a sentiment analysis problem, as it involved predicting the mood of the aliens based on what they said.
- Machine translation: Translating sentences from various languages into others.
- Speech recognition: Decoding human voice and turning it into text.
- Text summarization: Summarizing large texts into a few paragraphs.
- Chatbots: A system that can talk to humans and answer questions. These are not yet perfected, but there are very useful chatbots that operate in specific topics, such as customer support, etc.

The most useful architectures that work well for processing texts are *recurrent neural networks*, and some more advanced versions of them called long short-term memory networks (LSTM) and gated recurrent units (GRU). Again, these architectures are out of the scope of this book, but to give you an idea of what they are, imagine a neural network where the output gets plugged back into the network as part of the inputs. In this way, neural networks have a memory, and when trained properly, this memory can help them make sense of the topic in the text.

### 8.5.3 How neural networks generate faces that look real - Generative adversarial networks

In my opinion, the most fascinating among all the current applications of neural networks is in generation. So far, neural networks (and most other ML models in this book) have worked well in predictive machine learning, namely, being able to answer questions successfully. For example, a machine learning model can answer questions such as "how much is that?", or "is this A or B?". However, lately there have been many advances in a new area called *generative machine learning*. Generative machine learning is the area of machine learning that teaches the computer how to create things, rather than simply answer questions. Actions such as painting a painting, writing a song, writing a story, these actions represent a much higher level of understanding of the world.

Many recent advances have been made in generative machine learning, and without doubt one of the most important one has been *generative adversarial networks*, or GANs. Generative adversarial networks (GAN) has had some fascinating results when it comes to image generation. GANs consist of two competing networks, the generator and the discriminator. The generator tries to generate real-looking faces, while the discriminator tries to tell the real images and the fake images apart. During the training process, we feed real images to the discriminator, as well as fake images generated by the generator. Fascinatingly, this process results in a generator that can generate some very real-looking faces. In fact, they look so real, that humans sometimes have a hard time telling them apart. If you'd like to test yourself against the generator, go to [www.whichfaceisreal.com](http://www.whichfaceisreal.com).

## 8.6 Summary

- Neural networks are a very powerful algorithm used for classification and regression. A neural network consists of a set of perceptrons organized in layers, where the output of one layer serves as input to the next layer. Their complexity allows them to achieve great success in applications that are very difficult for other machine learning models.
- Neural networks have cutting edge applications in many areas, including image recognition and text processing.
- The basic building block of a neural network is the perceptron. A perceptron receives several values as inputs, and outputs one value. The way the perceptron outputs a value is by multiplying the inputs by weights, adding a bias, and applying an activation function.
- Popular activation functions include sigmoid, hyperbolic tangent, softmax, and the rectified linear unit (ReLU). The goal of the sigmoid and the hyperbolic tangent is to squish our input into a small interval, so the answer can be taken as a category. These two functions have flat derivatives when the input is large, which causes problems with vanishing gradients. The ReLU function doesn't have this problem, and so it has been used widely in between hidden layers to improve the training process.
- Neural networks have a very complex structure, which makes them hard to train. The process we use to train them is called backpropagation, which has shown great success.

Backpropagation consists of taking the derivative of the loss function and finding all the partial derivatives with respect to all the weights of the model. These derivatives are then used to update the weights of the model iteratively in order to improve its performance.

- Neural networks are prone to overfitting and other problems such as vanishing gradients, but there are techniques such as regularization and dropout which help us reduce these problems.
- There are very useful packages to train neural networks, such as Keras, TensorFlow, and PyTorch. These packages make it very easy for us to train neural networks, as we only have to define the architecture of the model, the error functions, and they take care of the training. Furthermore, they have many built-in cutting-edge optimizers that we can take advantage of.

# 9

## *Finding boundaries with style: Support vector machines and the kernel method*

### This chapter covers

- What is a support vector machine?
- What does it mean for a linear classifier to fit well between the points?
- A new linear classifier which consists of two lines, and its new error function.
- Tradeoff between good classification and a good fit: the C parameter.
- Using the kernel method to build non-linear classifiers.
- Types of kernels: polynomial and radial basis function (rbf) kernel.
- Coding SVMs and the kernel method in sklearn.

In Chapters 4 and 5, we learned about linear classifiers. In two dimensions, these are simply defined by a line that best separates a dataset of points with two labels. However, you may have noticed that many different lines can separate a dataset, and this raises the question: How do you know which is the best line? In figure 9.1 I show you three different classifiers that separate this dataset. Which one do you prefer, classifier 1, 2, or 3?

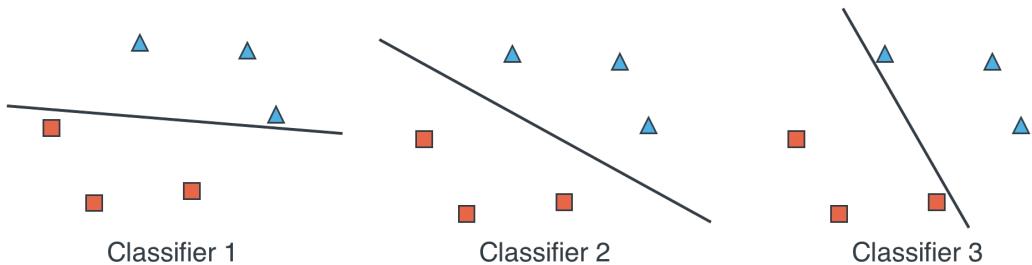


Figure 9.1. Three classifiers that classify our data set correctly. Which one do we prefer, Classifier 1, 2, or 3?

If you said Classifier 2, we agree. All three lines separate the dataset well, but the second line is better placed, since as you can see, the first and the third line are very close to some of the points, while the second line is not close to any points. If we were to wiggle the three lines around a little bit, the first and the third may go over one of the points, misclassifying them in the process, while the second one will still classify them all correctly.

This is where Support Vector Machines (SVMs for short) come into play. The trick that SVMs use is to use two parallel lines, and try to space them up apart as much as possible while still classifying the points correctly. In Figure 9.2 we can see the two parallel lines for the three classifiers. It is clear that the lines in Classifier 2 are farther from each other, which makes this classifier the best one.

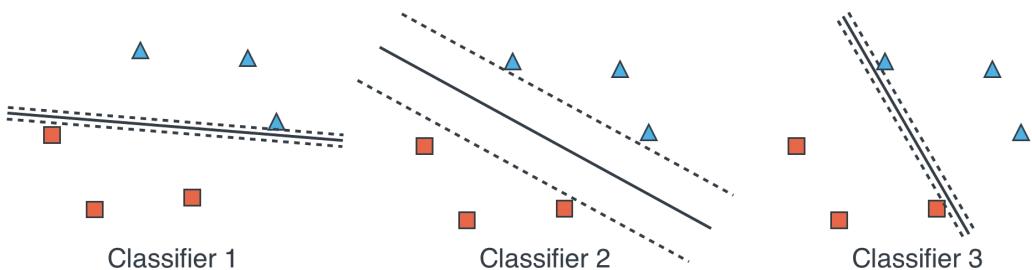


Figure 9.2. We draw our classifier as two parallel lines, as far apart from each other as possible. We can see that Classifier 2 is the one where the parallel lines are farther from each other. This means that the main line in Classifier 2 is the one best located between the points.

How do we build such a classifier? In a very similar way as before, only with a slightly different error function, and a slightly different iterative step.

## 9.1 Using a new error function to build better classifiers

We need to build an error function that will create a classifier consisting of two lines, as spaced apart as possible. When I think of building an error function, I always ask myself: "What do I

want to achieve?”. The error function should penalize any model that doesn’t achieve those things. Here, we want to achieve two things:

- We want the two lines to classify the points as best as possible.
- We want the two lines to be as far away from each other as possible.

Since we want two things, our SVM error function should be the sum of two error functions. One error function penalizes points that are misclassified, and the second penalizes lines that are too close to each other. Therefore, our error function should look like this:

Error = Classification Error + Distance Error.

In the next two sections we develop each one of these two error functions separately.

### 9.1.1 Classification error function - trying to classify the points correctly

One difference between SVMs and other linear classifiers is that SVMs use two parallel lines instead of one. Luckily, two parallel lines have very similar equations, they have the same weights, but a different bias. Thus, in our SVM, we will 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 9.3 shows the three parallel lines with the following equations:

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

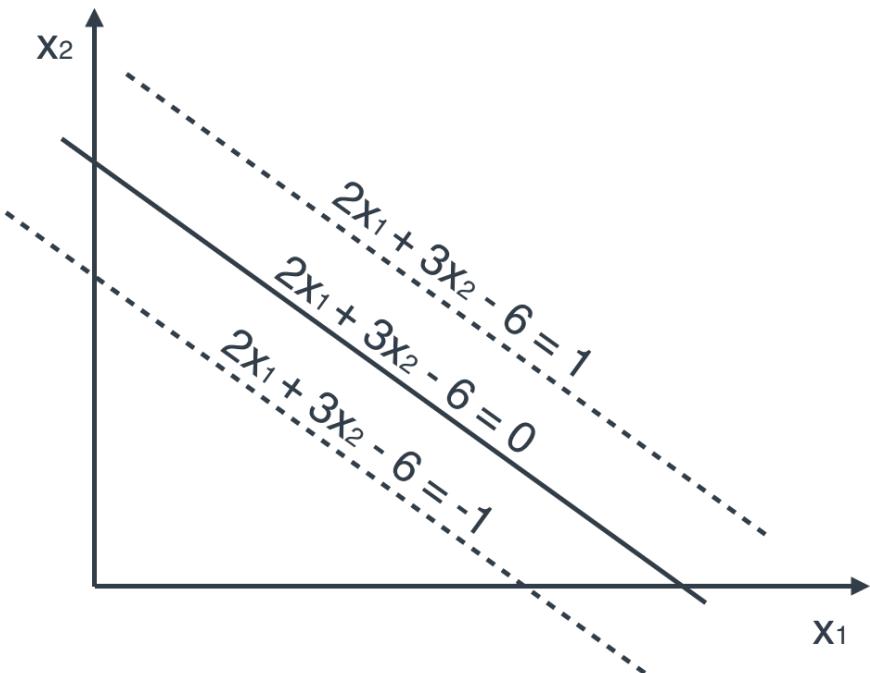


Figure 9.3. Our main line  $L$  is the one in the middle. We build the two parallel lines  $L+$  and  $L-$  by slightly changing the equation of  $L$ .

Our classifier will consist now of the lines  $L+$  and  $L-$ . The goal of this classifier is to have as few points as possible in between the two lines, and to have the two lines separate the points as best as possible. In order to measure this, we simply need to think of the classifier as two different classifiers, one for  $L+$  and one for  $L-$ . The classification function is defined as the sum of the error functions for both classifiers.

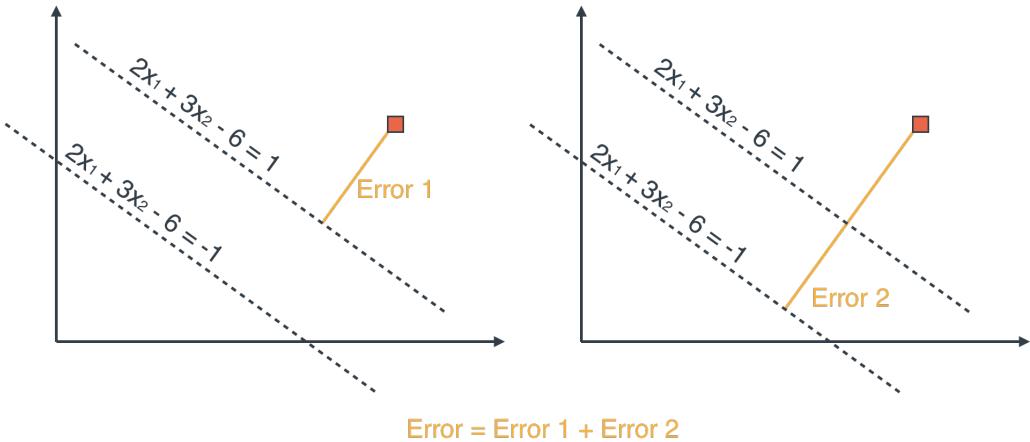


Figure 9.4. Now that our classifier consists of two lines, the error of a misclassified point is measured with respect to both lines. We then add the two errors to obtain the classification error.

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 the classifier does not classify it correctly.

### 9.1.2 Distance error function - trying to space our two lines as far apart as possible

Now that we have created an error function that takes care of classification errors, we need to build one that takes care of the distance between the two lines. In this section we build a surprisingly simple error function which is large when the two lines are close, and small when the lines are far.

This error function is so simple that you have already seen it before; it is the regularization term. More specifically, if our lines have equations  $ax_1 + bx_2 = 1$  and  $ax_1 + bx_2 = -1$ , then the error function is  $a^2 + b^2$ . Why so? We'll make use of the following fact. The perpendicular distance between the two lines is precisely  $\frac{2}{\sqrt{a^2 + b^2}}$ , as illustrated in Figure 9.5.

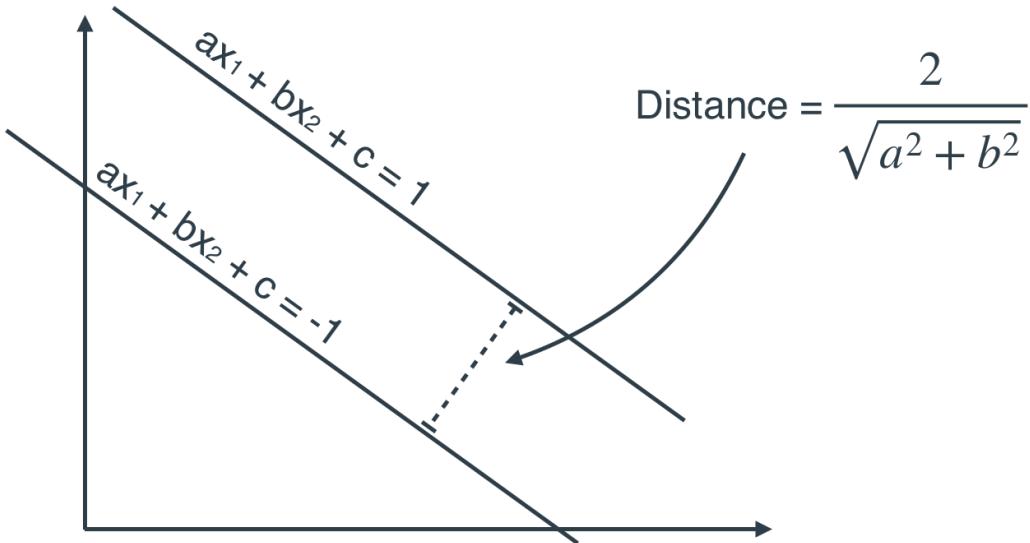


Figure 9.5. The distance between the two parallel lines can be calculated based on the equations of the lines.

Knowing this, notice that

- When  $a^2 + b^2$  is large,  $\frac{2}{\sqrt{a^2 + b^2}}$  is small.
- When  $a^2 + b^2$  is small,  $\frac{2}{\sqrt{a^2 + b^2}}$  is large.

Since we want the lines to be as far apart as possible, this term  $a^2 + b^2$  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 9.6 we can see two examples of classifiers, one with a large error function and one with a small error function. Recall that a classifier's job is to make a prediction of a label based on the features. In this case, the line is a classifier because it predicts the points on one side of the line as having a positive label, and the points on the other side as having a negative label. These two classifiers have the following equations:

- Classifier 1:
  - Line 1:  $3x_1 + 4x_2 + 5 = 1$
  - Line 2:  $3x_1 + 4x_2 + 5 = -1$
- Classifier 2:
  - Line 1:  $30x_1 + 40x_2 + 50 = 1$
  - Line 2:  $30x_1 + 40x_2 + 50 = -1$

Let's calculate the distance error function of each one.

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

Notice also from Figure 9.6, that the lines are much closer in classifier 2 than in classifier 1, which makes classifier 1 a much better classifier (from the distance perspective). The distance between the lines in classifier 1 is  $\frac{2}{\sqrt{3^2 + 4^2}} = 0.4$ , whereas in classifier 2 it is  $\frac{2}{\sqrt{30^2 + 40^2}} = 0.04$ .

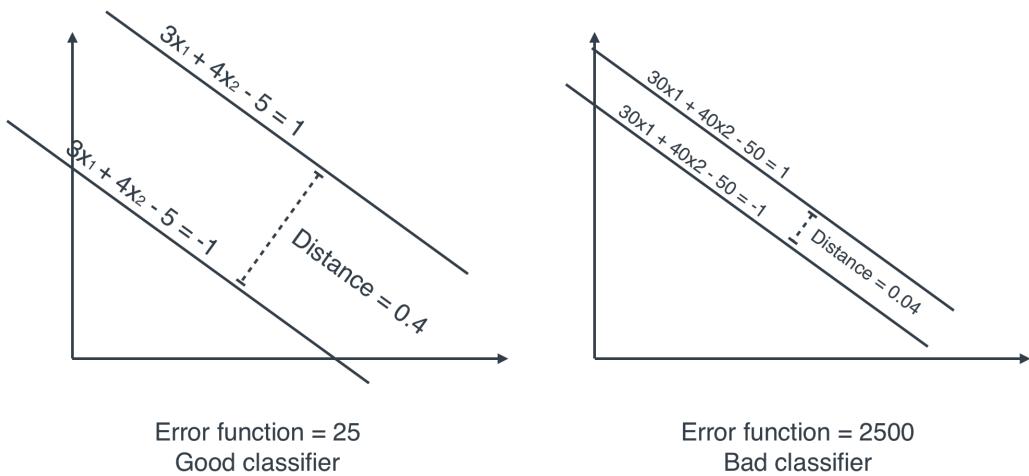


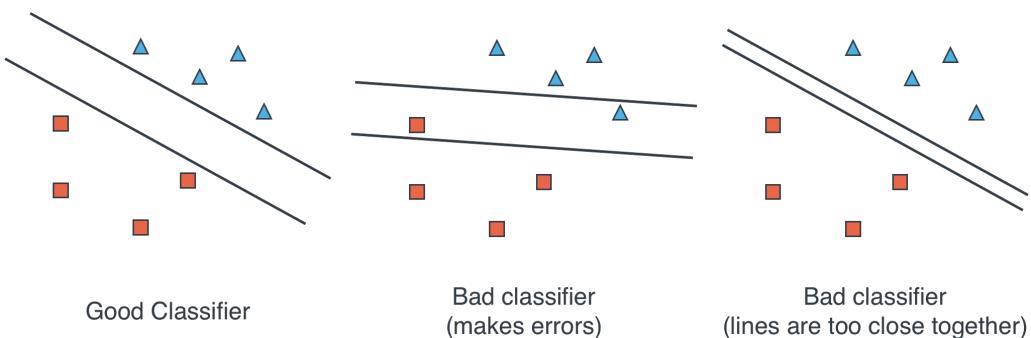
Figure 9.6. Left: A classifier where the lines are at distance 0.4 apart, with an error of 25. Right: A classifier where the lines are at distance 0.04 apart, with an error of 2500. Notice that in this comparison, the classifier in the left is much better than the one in the right, since it has a smaller error. This larger error is bad because it results in two lines that are much closer together.

### 9.1.3 Adding the two error functions to obtain the error function

In order to obtain an error function, we simply add the classification error function and the distance error function, so we get the following formula.

$$\text{Error} = \text{Classification Error} + \text{Distance Error}$$

A good classifier must then try to make very few classification errors, as well as try to keep the lines as far apart as possible.



**Figure 9.7.** Left: A good classifier, which consists of two well spaced lines and classifies all points correctly. Middle: A bad classifier which misclassifies two points. Right: A bad classifier which consists of two lines that are too close together.

In Figure 9.7 we can see three classifiers for the same dataset. The one on the left is a good classifier, since it classifies the data well, and the lines are far apart, reducing the likelihood of errors. The classifier in the middle makes some errors (since there is a triangle underneath the top line, and a square over the bottom line), so it is not a good classifier. The classifier in the right classifies the points correctly, but the lines are too close together, so it is also not a good classifier.

#### 9.1.4 Using a dial to decide how we want our model: The C parameter

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 9.8 we have two classifiers for the same data. The one on the left makes three errors, but the lines are far apart. The one on the right makes no errors, but the lines are too close together. Which one do we prefer?

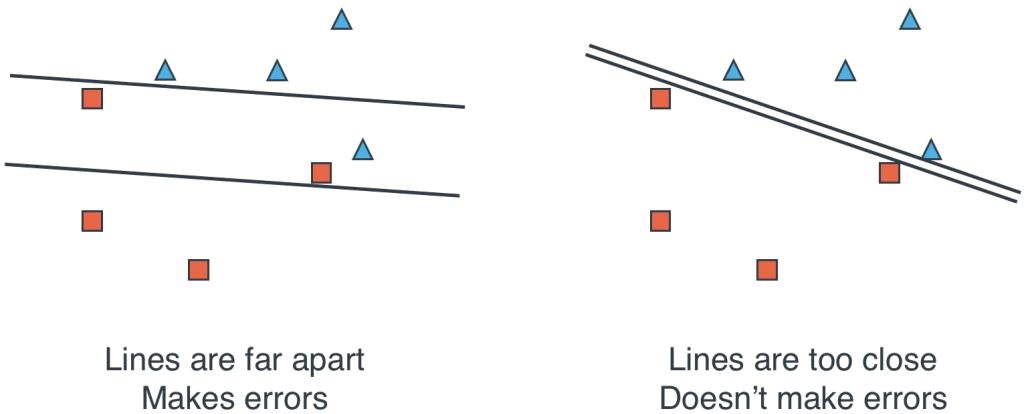


Figure 9.8. Both of these classifiers have one pro and one con. The one in the left has the lines well spaced (pro), but it misclassifies some points (con). The one in the right has the lines too close together (con), but it classifies all the points correctly (pro).

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

Error formula =  $C * (\text{Classification Error}) + (\text{Distance Error})$ .

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

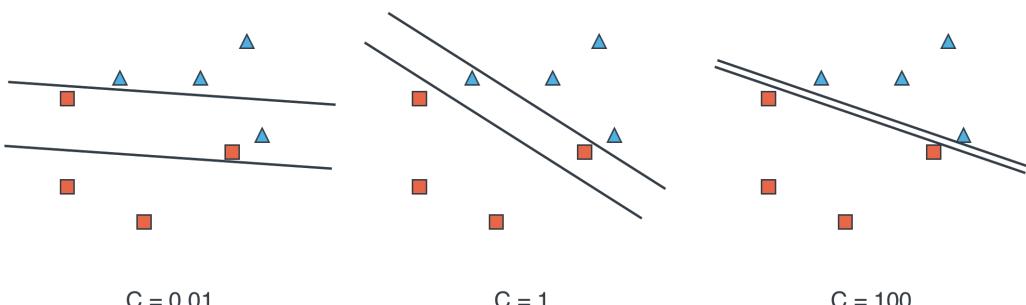


Figure 9.9. Different values of C toggle between a classifier with well spaced lines, and one that classifies points correctly. The classifier on the left has a small value of C (0.01), and the lines are well spaced, but it makes mistakes. The classifier on the right has a large value of C (100), and it classifies points correctly, but the lines are too close together. The classifier in the middle has a medium value of C (1), is a good middle ground

between the other two.

In Figure 9.9 we can see three classifiers. One with a large value of C which classifies all points correctly, one with a small value of C which 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 train using grid search, and our knowledge of the problem we're solving, the data, and the model.

Now we are ready to start coding Support Vector Machines.

## 9.2 Coding support vector machines in sklearn

In sklearn, coding an SVM is very simple, and in this section we learn how to do it. We also learn how to use the C parameter in our code.

### 9.2.1 Coding a simple SVM

Here is an example, which as usual, you can find in the book's Github repo at [www.github.com/luisquiserrano/manning](https://www.github.com/luisquiserrano/manning). We use the dataset called 'linear.csv', which we load and plot (Figure 9.10) as follows:

```
df = pd.read_csv('linear.csv')
X = np.array(df[['x_1', 'x_2']])
y = np.array(df['y']).astype(int)
plot_points(X,y)
```

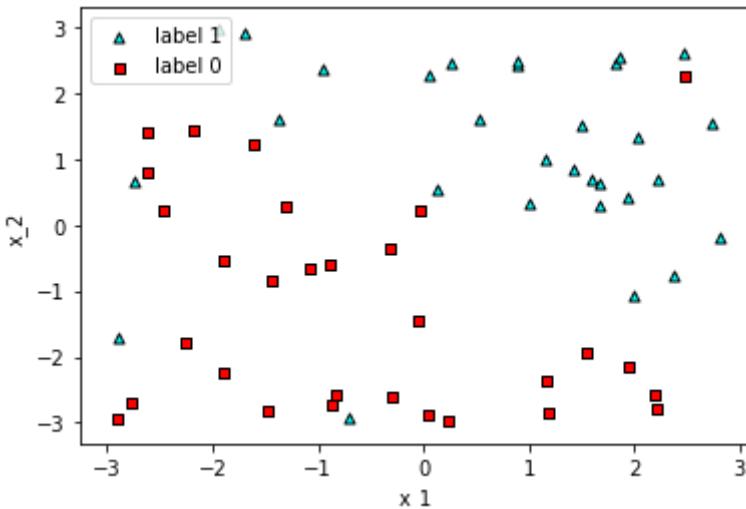


Figure 9.10. An linearly separable dataset, with noise.

We first import from the svm package in sklearn.

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

Then, we proceed to define our model, fit it, and plot it (Figure 9.11). We'll also print the accuracy. In this case, we get an accuracy of 0.9.

```
svm_linear = SVC(kernel='linear')
svm_linear.fit(X,y)
print("Accuracy:", svm_linear.score(X, y))
plot_model(X,y,svm_linear)
```

Accuracy: 0.9333333333333333

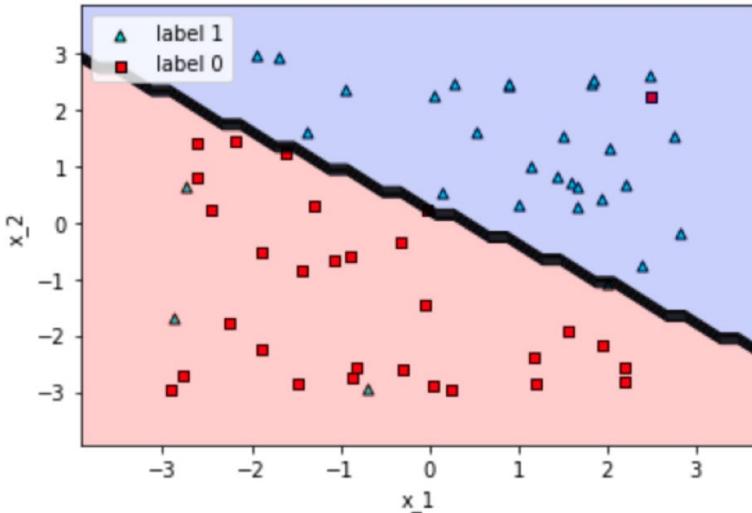


Figure 9.11. The plot of the SVM classifier we've built in sklearn consists of a line. Notice that the accuracy is 0.933.

Notice that the classifier drew a line that is nicely spaced from the points.

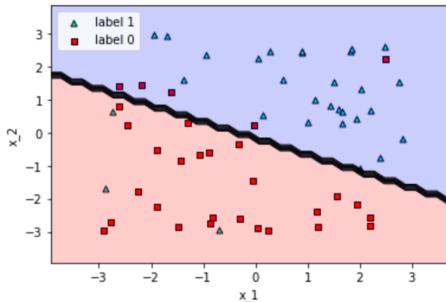
## 9.2.2 Introducing the C parameter

In sklearn, the C parameter can be easily introduced 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.

```
# C = 0.01
svm_c_001 = SVC(kernel='linear', C=0.01)
svm_c_001.fit(X,y)
print("C = 0.1")
print("Accuracy:", svm_c_001.score(X, y))
plot_model(X,y,svm_c_001)
```

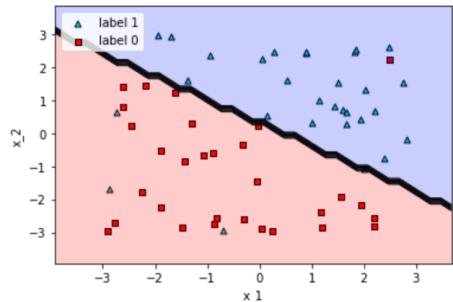
```
# C = 100
svm_c_100 = SVC(kernel='linear', C=100)
svm_c_100.fit(X,y)
print("C = 100")
print("Accuracy:", svm_c_100.score(X, y))
plot_model(X,y,svm_c_100)
```

Accuracy: 0.8666666666666666



C = 0.01

Accuracy: 0.9166666666666666



C = 100

**Figure 9.12.** 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.

If you notice carefully, you 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. You can see this in its low accuracy (0.86), and also in the squares that it mistakenly classifies as triangles in the upper left. 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, and its accuracy is high.

### 9.3 Going from lines to circles, parabolas, etc. - The kernel method

So far in this chapter we've learned a new method to build linear classifiers, such as a line, a plane, etc. In many aspects, SVMs are an improvement over other linear classifiers, since they try hard to find better boundaries. However, this is not the only advantage they have. As we've seen in other chapters of this book, not every dataset is linearly separable, and many times we need to build non-linear classifiers to capture the complexity of the data. In this section we study a powerful method associated to SVMs called the kernel method, which helps us build non-linear classifiers.

If we have a dataset, and find out 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 separable. If we do this in a clever way, we may then be able to remove these columns, and still end up with a classifier (although this one won't be linear anymore).

There is a geometric way to see this. Imagine that your dataset is in two dimensions, which means, your input has two columns. If you add a third column, now you have a three-dimensional dataset. This is like if the points in your paper all of a sudden start flying into space, at different heights. Maybe there is a hope that now, the points are separable by a plane. This is the kernel method, and it is illustrated in Figure 9.13.

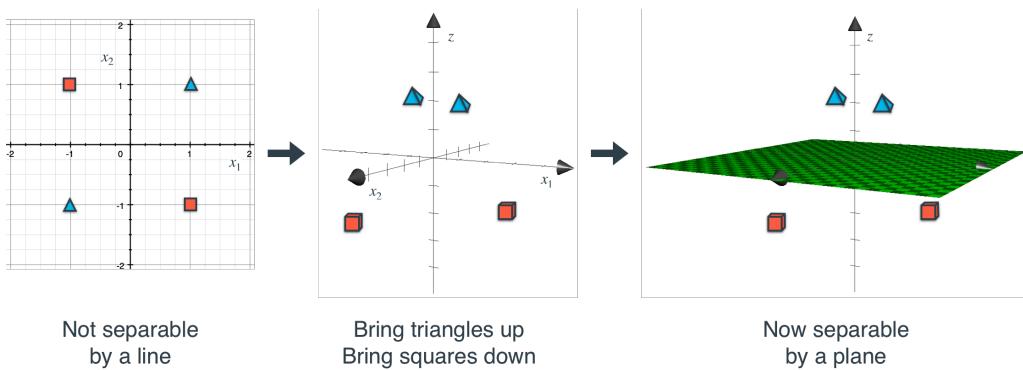


Figure 9.13. Left: The set is not separable by a line. Middle: We look at it in three dimensions, and proceed to raise the two triangles and lower the two squares. Right: Our new dataset is now separable by a plane.

#### **WHY IS IT CALLED THE KERNEL METHOD?**

It is called the kernel method because it uses a *kernel function* in order to embed our dataset in higher dimensions. Depending on what kernel function we use, we get a different embedding. In this chapter we learn two kernels: one using polynomial functions (the *polynomial kernel*), and one using radial basis functions (the *radial basis function*, or *rbf kernel*).

#### **9.3.1 Using polynomial equations (circles, parabolas, hyperbolas, etc.) to our benefit - The polynomial kernel**

Support vector machines use lines to separate the data (or planes, or hyperplanes, depending on the number of dimensions). The kernel method simply consists of adding polynomial equations to the mix, such as circles, parabolas, hyperbolae, etc. In order to see this more clearly, let's look at two examples.

##### **EXAMPLE 1: A CIRCULAR DATA SET**

For our first example, let's try to classify the dataset in Table 9.1.

**Table 9.1.** A small dataset.

$x_1$	$x_2$	$y$
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

When we plot this dataset, we obtain Figure 9.14, where the points with label 0 are drawn as squares, and those with label 1 are drawn as triangles. Notice that there is no line that can split the triangles and the squares.

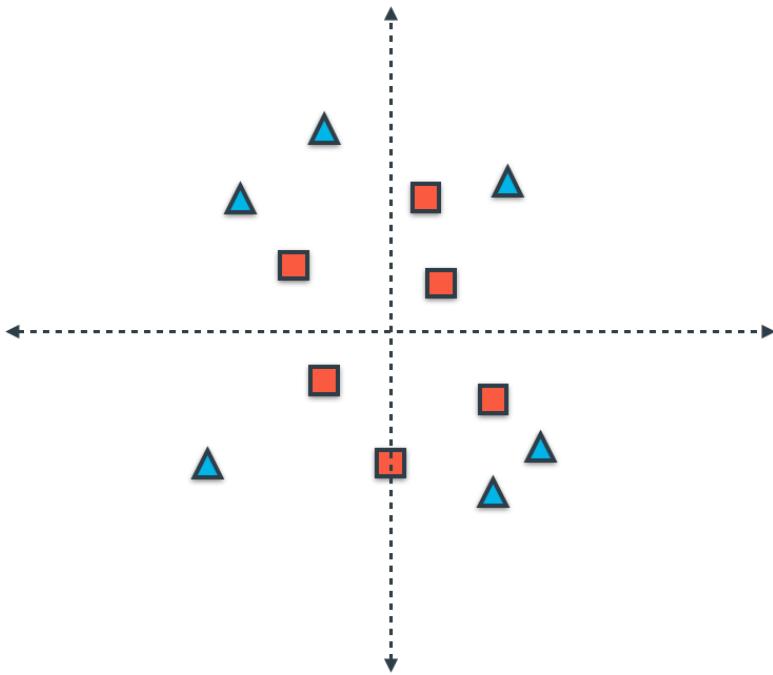
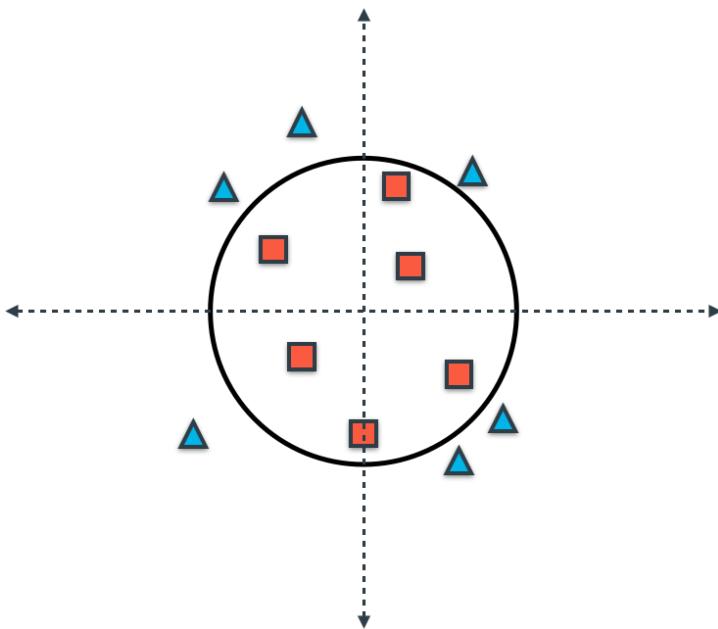


Figure 9.14. Plot of the dataset in Table 9.1. Note that it is not separable by a line. Therefore, this dataset is a good candidate for the kernel method.

When we look at the plot in Figure 9.14, it is clear that a line won't be able to separate the squares from the triangles. However, a circle would (Figure 9.15). Now the question is, if a support vector machine can only draw linear boundaries, how do we draw this circle?



**Figure 9.15.** The kernel method gives us a classifier with a circular boundary, which separates these points well..

In order to draw this boundary, let's think. What is a characteristic that separates the squares from the triangles? Well, it seems that the triangles are farther from the origin than the circles. The formula that measures the distance to the origin is simply the square root of the sum of squares of the two coordinates. If these coordinates are  $x_1$  and  $x_2$ , then this distance is simply  $\sqrt{x_1^2 + x_2^2}$ . Let's forget about the square root, and only think of  $x_1^2 + x_2^2$ . Let's add a column on Table 9.1 with this value and see what happens. The result can be seen in Table 9.2.

**Table 9.2.** We have added one more column to Table 9.1. 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 9.2, 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, since the variables are raised to a power greater than one. In fact, this is precisely the equation of a circle.

The geometric way I like to imagine this is as in Figure 9.16. Our original set lives in the plane, and it is impossible to separate 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 get raised a small amount, since they are close to the origin, and the triangles get raised a large amount, since they are far from the origin. Now the squares and triangles are far away from each other, and therefore, we can separate them with the plane of 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  $x^2 + y^2 = 1$ , which is our desired classifier. Notice that it is not linear, since it has quadratic terms, but this is ok.

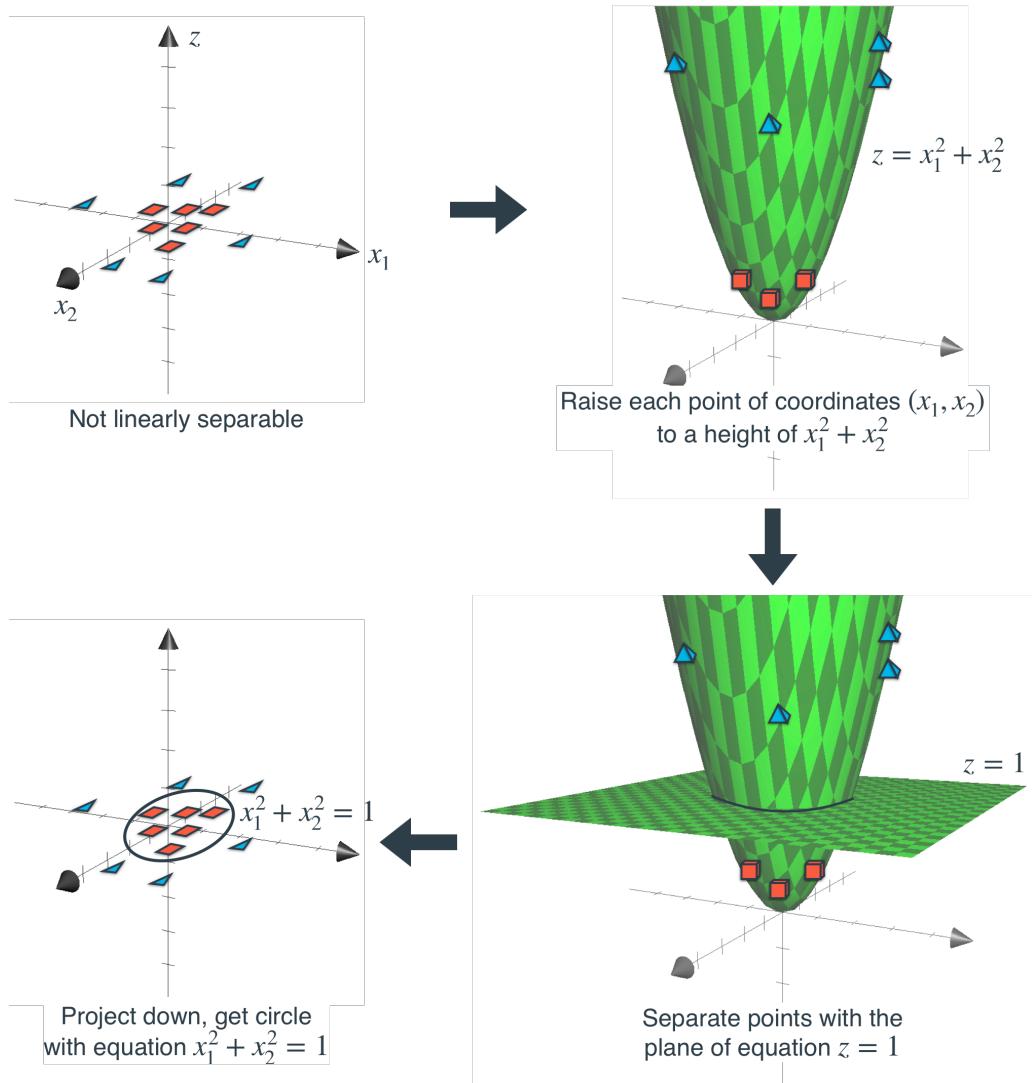


Figure 9.16. The kernel method. Step 1: We start a dataset that is not linearly separable. Step 2: Then we raise each point by a distance which is the square of its distance to the origin. This creates a paraboloid. Step 3: Now the triangles are high, while the squares are low. We proceed to separate them with a plane. Step 4. We project everything down. The intersection between the paraboloid and the plane gives a circle. The projection of this circle gives us the circular boundary of our classifier.

### **EXAMPLE 2: THE AND OPERATOR ON A RECOMMENDATION SYSTEM**

But circles are not the only figure we can draw. Let's look at a more real-world example. We have an online store, and we have a dataset of eight customers. Our online store only sells one product. Some customers buy it and some don't. When a customer leaves the page, they are asked to fill in a feedback through a form (and somehow, all of the customers filled it in). The two questions in the feedback form are:

- Did you like the product?
- Did you like the price of the product?

We end up with the dataset in Table 9.3 that tells us if the customer liked the product, if they liked the price, and if they bought the product or not. In this dataset, the two inputs are  $x_1$  and  $x_2$ , which correspond to the customer liking the product and the price. The output is  $y$ , corresponding to the customer buying the product or not..

**Table 9.3. A table of customers. For each customer, we record if they liked the product, if they liked the price, and if they bought the product.**

$x_1$ (Liked the product)	$x_2$ (Liked the price)	$y$ (Bought the product)
0	0	0
0	0	0
0	1	0
0	1	0
1	0	0
1	0	0
1	1	1
1	1	1

At first glance, the dataset looks like a simple AND operator, since the customers who bought the product are those who liked the product AND also liked the price. But can we model the AND operator as a polynomial equation in the same fashion as we did in the last example? Well, since all the inputs are 0 or 1, the AND operator is simply the product (since it's only 1 if the two factors are 1, and otherwise 0). Let's add the column corresponding to the product of  $x_1$  and  $x_2$ .

**Table 9.4.** We have added a column to Table 9.3, which consists of the product of the first two columns. Since the first two columns can be seen as yes/no answers, this new column can also be seen as the result of the first column AND the second column.

$x_1$ (Liked the product)	$x_2$ (Liked the price)	$x_1x_2$ (Liked the price and the product)	$y$ (Bought the product)
0	0	0	0
0	0	0	0
0	1	0	0
0	1	0	0
1	0	0	0
1	0	0	0
1	1	1	1
1	1	1	1

Notice that the column corresponding to the  $x_1x_2$  is exactly the column of labels. We can now see that a good classifier for this data is the one with the following equation:  $x_1x_2 = 0.5$ . When we plot this dataset and the equation, we get the graph in Figure 9.17., which precisely corresponds to the graph of a hyperbola.

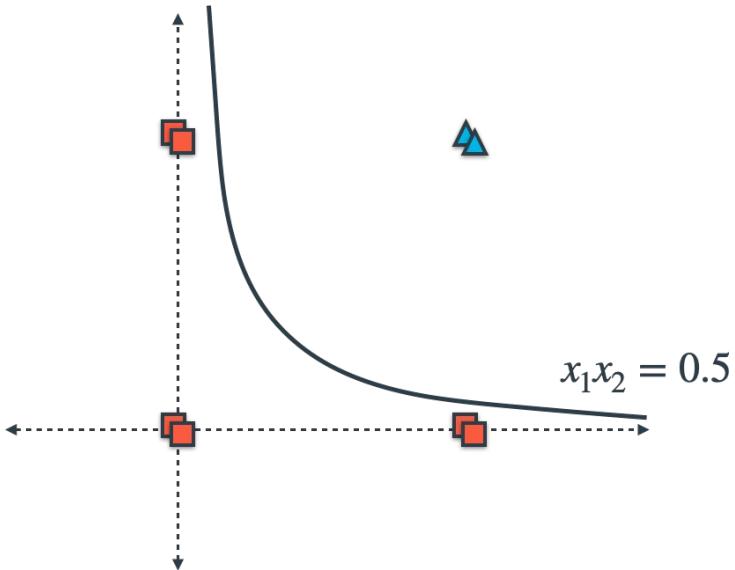


Figure 9.17. The plot of our dataset in Table 9.3. Notice that each point appears twice.

I encourage you to imagine how this would look as in three dimensions, in the way we worked out the previous example. It will look similar to Figure 9.13, except in here, we raise the triangles a height of 1, and leave the squares in the floor. Then we cut with a plane at height 0.5, and then we project down to get the desired curve.

#### **GOING BEYOND QUADRATIC EQUATIONS - THE POLYNOMIAL KERNEL**

In both of the previous examples, we had a dataset that was not separable by a linear classifier (in this case, a line). In both cases, a new expression came to our rescue. In the first example, this expression was  $x_1^2 + x_2^2$ , since that value is small for points near the origin and large for points far from the origin. In the second example, the expression was  $x_1x_2$ , since that corresponds to the AND operator (when the variables are either 0 or 1).

You may be wondering, 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 very simple, we 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 simply call these our new variables, and we treat them as if they had no relation with  $x_1$  and  $x_2$ . We can even call them  $x_3$ ,  $x_4$ , and  $x_5$  to avoid confusion. Thus, Table 9.3 now becomes Table 9.5.

**Table 9.5. We have added 3 more rows to Table 9.5, one corresponding to each of the monomials of degree 2 on the two variables.**

$x_1$	$x_2$	$x_3 = x_1^2$	$x_4 = x_1x_2$	$x_5 = x_2^2$	$y$
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

We can now build a support vector machine that classifies this enhanced dataset. I encourage you to build such a classifier using Sklearn, Turi Create, or the package of your choice. Here's a particular equation of a classifier that works (although many different ones will work for this dataset):

$$0x_1 + 0x_2 + 1x_3 + 0x_4 + 1x_5 - 1 = 0.$$

Remembering that  $x_3 = x_1^2$  and  $x_5 = x_2^2$ , we get the desired equation of the circle:

$$x_1^2 + x_2^2 = 1.$$

If we want to visualize this process geometrically, like we've done with the previous ones, it gets a little more complicated. Our nice 2-dimensional dataset became a 5-dimensional dataset. In this one, the points labelled 0 and 1 are now far away, and can be separable by a 4-dimensional hyperplane. When we project this down to 2 dimensions, we get the desired circle.

The polynomial kernel, in this case, the kernel of the variables  $x_1$  and  $x_2$ , is precisely the set  $\{x_1, x_2, x_1^2, x_1x_2, x_2^2\}$ . Since 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 about a polynomial kernel of higher degree, say,  $k$ ? This is defined as the set of monomials in the given set of variables, of degree less than or equal to  $k$ . For example, the degree 3 polynomial kernel on the variables  $x_1$  and  $x_2$  is precisely the set  $\{x_1, x_2, x_1^2, x_1x_2, x_2^2, x_1^3, x_1^2x_2, x_1x_2^2, x_2^3\}$ .

We can also define a polynomial kernel on many more variables in the same way. Here is the degree 3 polynomial kernel in the set of variables  $x_1$ ,  $x_2$ , and  $x_3$ :  $\{x_1, x_2, x_3, x_1^2, x_1x_2, x_1x_3, x_2^2, x_2x_3, x_3^2\}$ .

Here is a question for you: How many elements does the polynomial kernel of degree 4 have, on the variables  $x_1$ ,  $x_2$ , and  $x_3$ ? Feel free to think about it yourself before reading the answer. The answer is 35. There are 4 of the form  $x_a^4$ , 12 of the form  $x_a^3x_b$ , 6 of the form  $x_a^2x_b^2$ , 12 of the form  $x_a^2x_bx_c$  and 1 of the form  $x_a x_b x_c x_d$ .

### 9.3.2 Using bumps in higher dimensions to our benefit - The radial basis function (rbf) kernel

Imagine if you had a point in the plane, and the plane was like a blanket (as illustrated in Figure 9.18). Then you pinch the blanket at that point, and raise it. This is how a *radial basis function* looks like. It is called a radial basis function 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. The *radial basis function kernel* (also called rbf kernel) is precisely the set of all radial basis functions for every point in the plane.

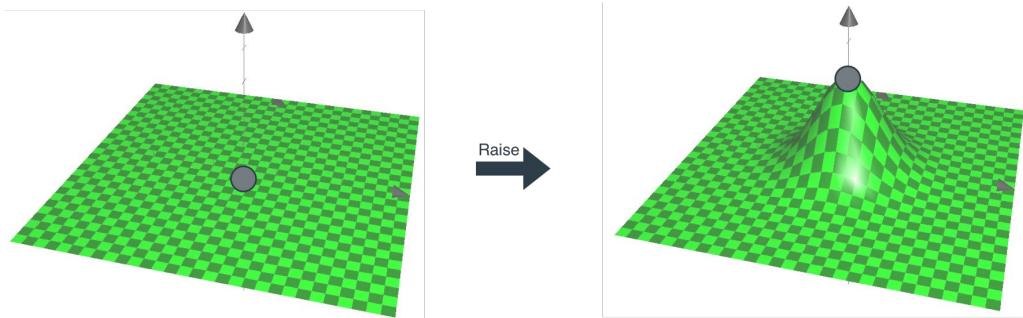


Figure 9.18. A radial basis function consists of raising the plane at a particular point.

Now, how do we use this as a classifier? Imagine the following: You have the dataset at the left of Figure 9.19, 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 3-dimensional plot at the right of Figure 9.19.

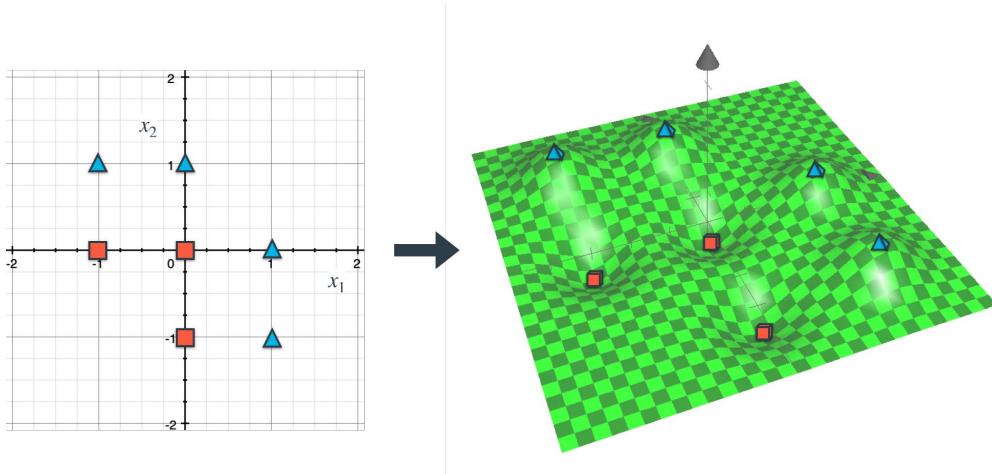


Figure 9.19. Left: A dataset in the plane. Right: We have used the radial basis functions to raise each of the triangles and lower each of the squares. Notice that now we can separate the dataset by a plane, which means our new dataset is linearly separable.

In order to create the classifier, we simply 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 with the sea. The curve will correspond to the coastline, namely, where the water and the land meet. This gives us the curve in the left of Figure 9.20.

We then project everything back to the plane, and obtain our desired classifier, shown in the right of Figure 9.20.

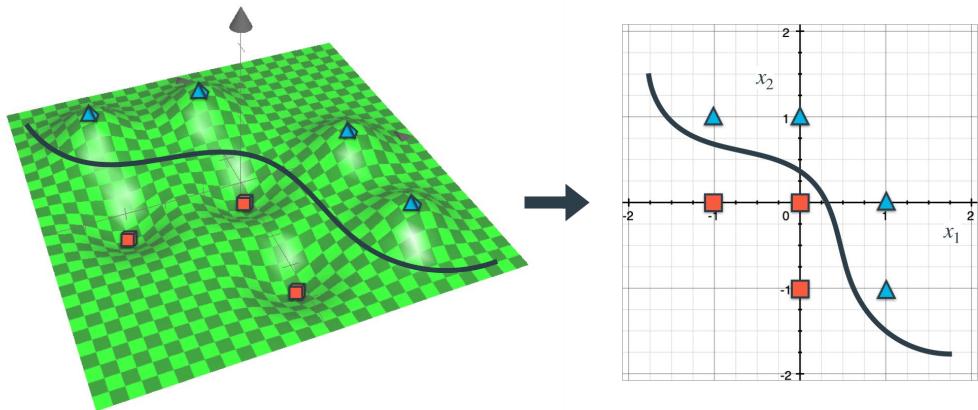


Figure 9.20. Left: If we look at the points at height 0, they form a curve. If we think of the high points as land and the low points as the sea, this curve is the coastline. Right: When we project (flatten) the points back to the plane, the coastline is now our classifier that separates the triangles from the squares.

That is the idea behind the rbf kernel. Of course, we have to develop the math, which we will in the next few sections. But in principle, if you can imagine lifting and pushing down a blanket, and then building a classifier by looking at the boundary of the points that lie at height 0, then you understand what an rbf kernel is.

#### **A MORE IN DEPTH LOOK AT RADIAL BASIS FUNCTIONS**

Radial basis functions can exist in any number of variables. I will show you the ones for a small number of variables first, and you will quickly see the trend. For one variable, one of the simplest radial basis function has the formula  $y = e^{-x^2}$ . This looks like a bump over the line (Figure 9.21.). You may notice that it looks a lot like a Gaussian, or a normal distribution, and this is correct.

The normal distribution is very similar, but it has a slightly different formula (it is scaled by  $\frac{1}{\sqrt{2\pi}}$  so that the area underneath it is 1).

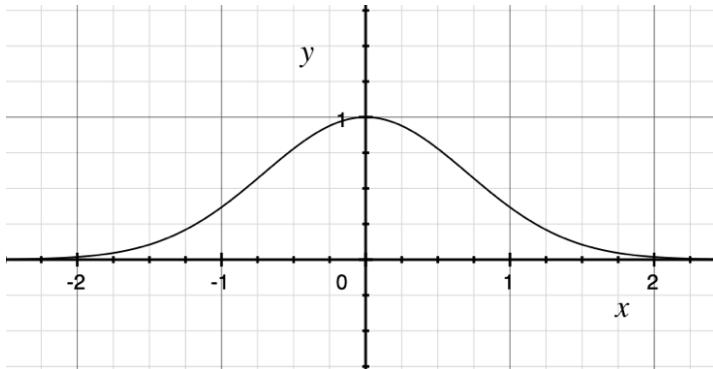


Figure 9.21. An example of a radial basis function. It looks a lot like a Gaussian (normal) distribution.

Notice that this bump happens at 0. If we wanted it to appear at any different point, say  $p$ , we simply translate the formula, and get  $y = e^{-(x-p)^2}$ . Thus, if I want to obtain 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  $z = e^{-(x^2+y^2)}$ , and it looks like the plot in Figure 9.22. Again, you may notice that it looks a lot like a multivariate normal distribution, and it does. It is again, a scaled version of the multivariate normal distribution.

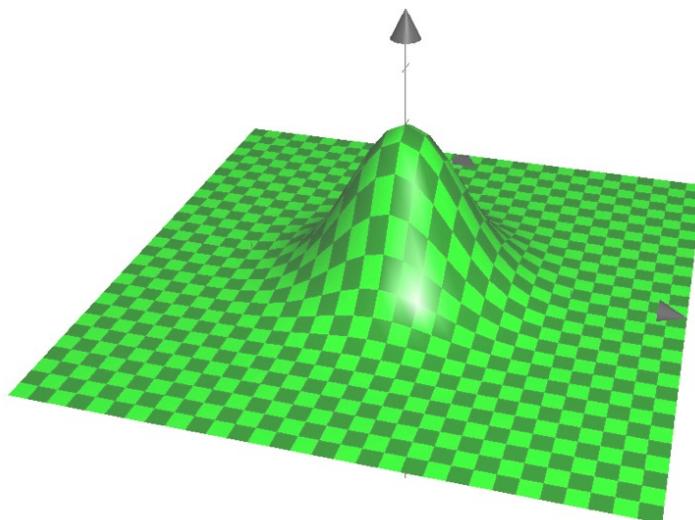


Figure 9.22. A radial basis function on two variables. It again looks a lot like a Gaussian distribution.

Again, this bump happens exactly at the point (0,0). If we wanted it to appear at any different point, say (p,1), we simply translate the formula, and get  $y = e^{-[(x-p)^2 + (y-q)^2]}$ . Thus, if I want to obtain the radial basis function centered at the point (2,-3), the formula is precisely  $y = e^{-[(x-2)^2 + (y+3)^2]}$ .

For n variables, the formula for the basic radial basis function is  $y = e^{-(x_1^2 + \dots + x_n^2)}$ . I can't draw a plot in n+1 dimensions, but we can imagine it in our heads. However, since the algorithm that I will present to you is purely mathematical, the computer has no trouble running it in as many number of variables as we want.

And as usual, this n-dimensional bump is centered at zero, but if we wanted it centered at the point  $(p_1, \dots, p_n)$ , the formula is  $y = e^{-[(x_1 - p_1)^2 + \dots + (x_n - p_n)^2]}$ .

#### **A MEASURE OF HOW CLOSE POINTS ARE - SIMILARITY**

In order 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 by, and not similar if they are far away (Figure 9.23). Similarity is then a number assigned to every pair of points which is high if the points are close, and low if they are far away. 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. We'll get to how to determine the similarity value in just a bit.

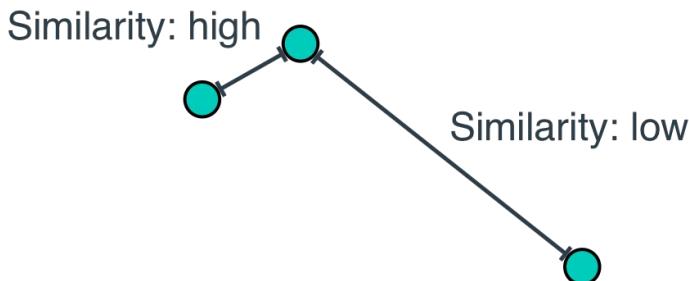


Figure 9.23. Two points that are close by are defined to have high similarity. Two points that are far away are defined to have low similarity.

Now we need to find a formula for similarity. As you can see, similarity grows inversely as distance. Thus, many formulas for similarity would work, as long as the similarity increases while the distance decreases. Since we are using exponential functions a lot in this section, let's define it as follows. For points p and q, the similarity between p and q is:

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

That looks like a complicated formula for similarity, but there is a very nice way to see 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 9.24., we can see this for one variable, but you can imagine it in any number of variables by using the blanket analogy.

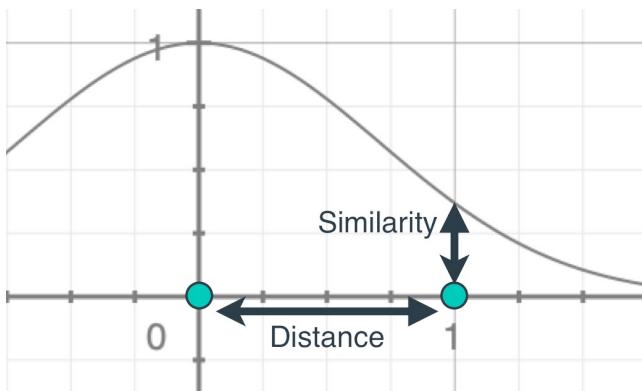


Figure 9.24. The similarity is defined as the height of a point in the radial basis function, where the input is the distance. Note that the higher the distance, the lower the similarity, and vice versa.

### 9.3.3 Training an SVM with the rbf kernel

Ok, we now have all the tools to train an SVM using the rbf kernel. Let's look again at the example at the beginning of this section. The dataset is in Table 9.6, and the figure in Figure 9.19.

Table 9.6. A simple dataset in two dimensions. 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

This is the same dataset that it is plotted in Figure 9.19, and for convenience I've repeated it right here.

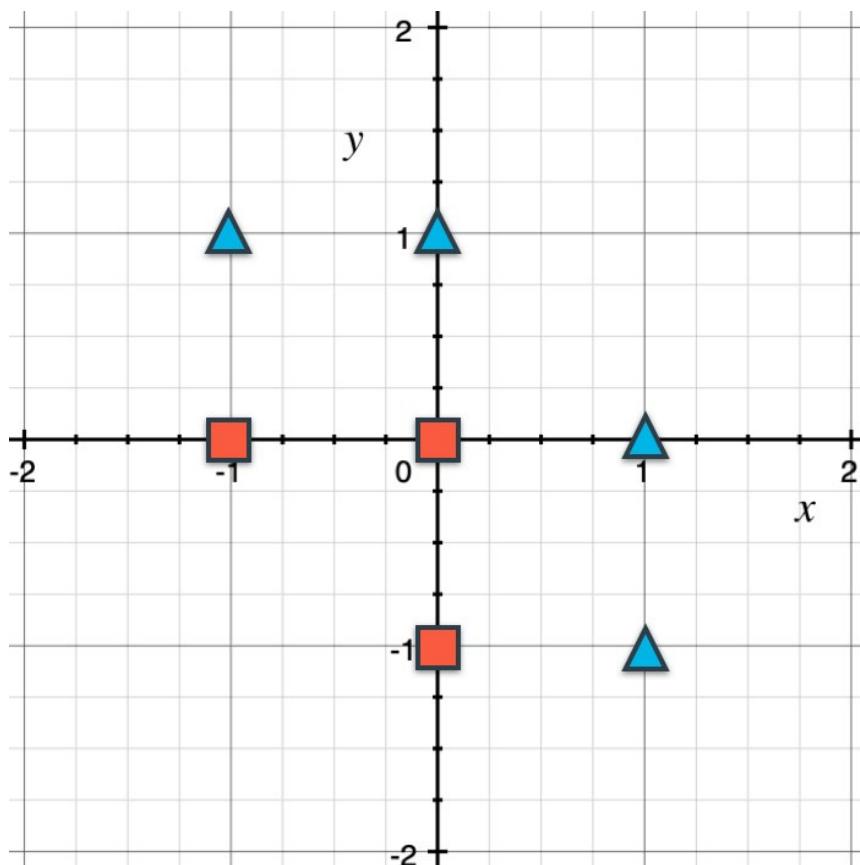


Figure 9.25. The plot of the dataset in Table 9.6, where the points with label 0 are squares, and those with label 1 are triangles. Notice that the squares and triangles cannot be separated with a line. We'll use an SVM with an rbf kernel to separate them with a curved boundary.

Notice that in the first column of Table 9.6., we have added a number for every point. This is not part of the data, it is only for convenience. We will now add 7 columns to this table. The columns will be the similarities with respect to every point. For example, for point 1, we add a similarity column named "Sim 1". 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, point 6. The distance between point 1 and point 6, by the pythagorean theorem is:

$$\text{distance}(\text{point 1}, \text{point 6}) = \sqrt{(0+1)^2 + (0-1)^2} = \sqrt{2}.$$

Therefore, the similarity is precisely:

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

I invite you to fill in a few more values in this table by yourself to convince yourself that this is the case. The result is in Table 9.7.

**Table 9.7. We have added 7 similarity columns to the dataset in Table 9.6. Each one records the similarities with all the other 6 points.**

Point	$x_1$	$x_2$	Sim 1	Sim 2	Sim 3	Sim 4	Sim 5	Sim 6	Sim 7	$y$
1	0	0	1	0.135	0.135	0.135	0.135	0.368	0.368	0
2	-1	0	0.135	1	0.368	0.368	0.018	0.135	0.018	0
3	0	-1	0.135	0.368	1	0.018	0.135	0.007	0.368	0
4	0	1	0.135	0.368	0.018	1	0.135	0.368	0.007	1
5	1	0	0.135	0.018	0.135	0.135	1	0.007	0.368	1
6	-1	1	0.368	0.135	0.007	0.368	0.007	1	0	1
7	1	-1	0.368	0.018	0.368	0.007	0.368	0	1	1

Notice a couple of things:

1. The table is symmetric, since the similarity between p and q is the same as the similarity between q and p (since it only depends on the distance between p and q).

- The similarity between points 6 and 7 appears as 0, but in reality it is not; it only rounds to zero. The distance between points 6 and 7 is  $\sqrt{2^2 + 2^2} = \sqrt{8}$ , so their similarity is  $e^8 = 0.00033546262$ , which rounds to zero, since we are rounding to 3 significant figures.

Now, on to building our classifier! Notice that for the data on the small Table 9.6, no linear classifier works (since the points can't be split by a line), but on the much larger Table 9.7. that has a lot more features, we can fit such a classifier. We proceed to fit an SVM to this data. Many SVMs can classify this dataset correctly. I'll show you one that works. This classifier has the following weights:

- The weights of  $x_1$  and  $x_2$  are  $w_1$  and  $w_2$ .
- The weight of Sim p is  $v_p$ , for  $p = 1, 2, \dots, 7$ .
- The bias is  $b$ .

A classifier that works well is the following:

- $w_1 = 0$
- $w_2 = 0$
- $v_1 = -1$
- $v_2 = -1$
- $v_3 = -1$
- $v_4 = 1$
- $v_5 = 1$
- $v_6 = 1$
- $v_7 = 1$
- $b = 0$

What I did to find that classifier was no science. I simply added a label -1 to the columns corresponding to the points labelled 0, and a +1 to the columns corresponding to the points labelled 1. To check that this works, simply take Table 9.7, add the values of the columns Sim 4, Sim 5, Sim 6, and Sim 7, then subtract the values of the columns Sim 1, Sim 2 and Sim 3. You'll notice that you get a negative number in the first three rows, and a positive one in the last four rows. We can now simply add a threshold of 0, and we have a classifier which classifies this dataset correctly, since the points labeled 1 get a positive score, and the points labeled 0 get a negative score.

Of course, that may not be the best classifier. Our SVM may be able to find a different one. But the point is, we used the radial basis functions to add many more columns to our table, based on similarities between the points, and this helped us build a linear classifier (in a much higher dimensional space), which we then projected into our plane to get the classifier we wanted. We can see the resulting curved classifier in Figure 9.26.

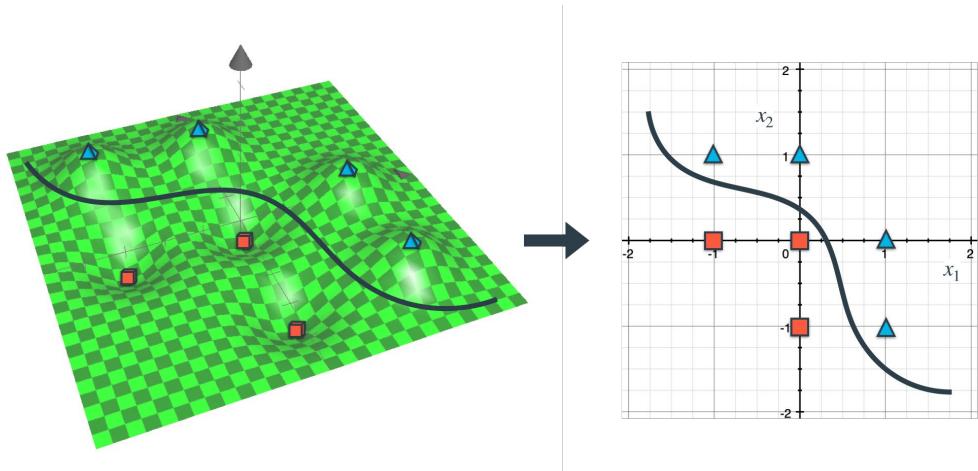


Figure 9.26. In this dataset, we have raised each triangle and lowered each square. Then we have drawn a plane at height 0, which separates the squares and the triangles. The plane intersects the surface in a curved boundary. We then project everything back down to two dimensions, and this curved boundary is the one that separates our triangles from our squares. The boundary is drawn at the right.

#### **OVERFITTING AND UNDERFITTING WITH THE RBF KERNEL - THE GAMMA PARAMETER**

At the beginning of this section, we mentioned that there were many different radial basis functions, 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 9.27. In practice, the wideness of our radial basis functions is something we want to tune (I elaborate on this below). In order to tune the wideness of the surface, we use a parameter called the *gamma parameter*. When the gamma parameter is small, the curve formed is very wide, and when gamma is large, the curve is very narrow.

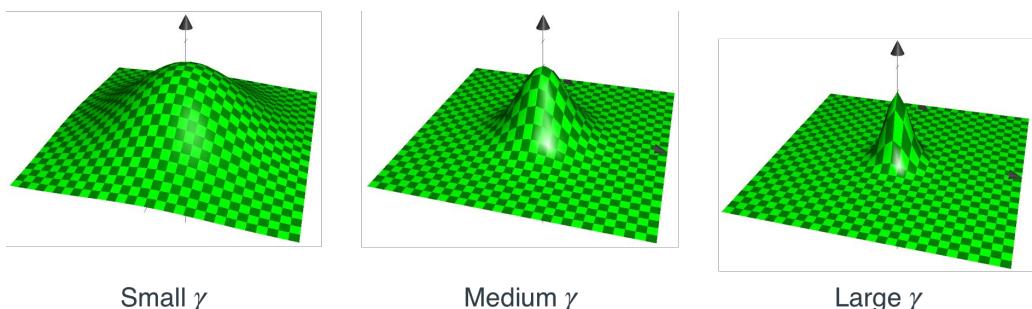
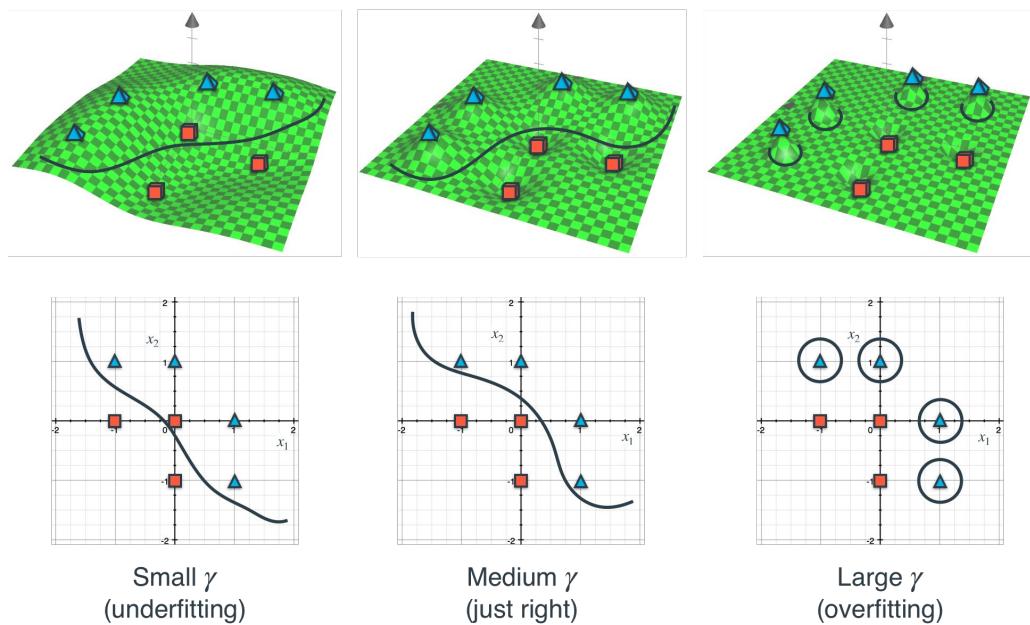


Figure 9.27. The gamma parameter determines how wide the surface is. Notice that for small values of gamma, the surface is very wide, and for large values of gamma, the surface is very narrow.

This gamma is a hyperparameter. Recall that hyperparameters are the specifications that we use to train our model. The way to tune this hyperparameter is using methods that we've seen before, such as grid search. As you can imagine, 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.



**Figure 9.28. Three SVM classifiers with an rbf kernel and different values of gamma.** Notice that the classifier on the left (small value of gamma) underfits, since it doesn't classify all the points well. The classifier on the right (large value of gamma) overfits, as it only manages to draw a small circle around each of the triangles, while classifying everything else as a square. The classifier in the middle is good, as it draws a boundary that is simple enough, yet classifies the points correctly.

Notice that for a very small value of gamma, the model overfits, as the curve is too simple and it doesn't classify our data well. For a large value of gamma, the model vastly overfits, since 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, as it builds a boundary that is simple enough, yet classifies the points correctly. Notice how similar this is from a k-nearest neighbor model. The only difference is that now, instead of looking at what the closest points are, we look at similarity between points, and even far away points can still be considered similar, just at a very small scale.

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, to get the following equation. In the general case, we get the following:

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

Don't worry very much about learning this formula. Just know that in the same way that we can make bumps in the plane, this is possible to do in higher dimensions as well, with that formula. And as usual, there is a way to code this and make it work, which is what we do in the next section.

### 9.3.4 Coding the kernel method

In order to train an SVM in sklearn with a particular kernel, all we do is add the kernel as a parameter when we define the SVM. In this section I show you how to train SVM classifiers with the polynomial and the rbf kernels.

#### **CODING THE POLYNOMIAL KERNEL TO CLASSIFY A CIRCULAR DATASET**

In this subsection I show you how to code the polynomial kernel in sklearn. For this, we use the following circular dataset called 'one\_circle.csv' (Figure 9.27).

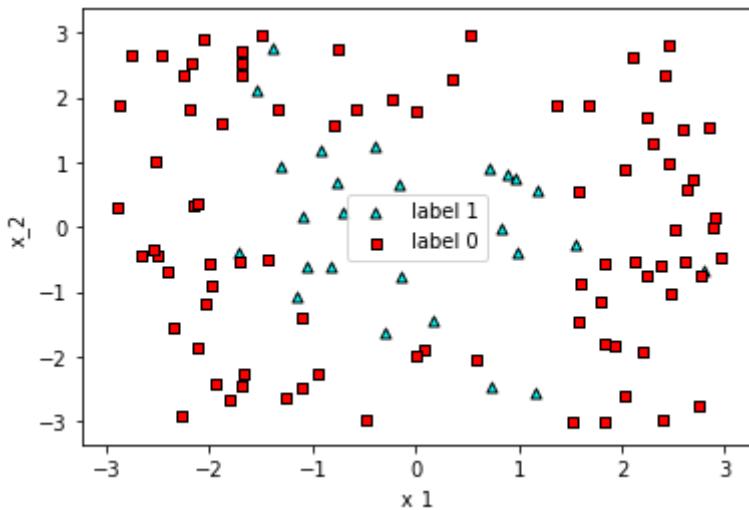


Figure 9.29. A circular dataset, with some noise. We will use an SVM with the polynomial kernel to classify this dataset.

Notice that this is indeed a circular dataset (with some noise). We train an SVM classifier where we specify the ‘kernel’ parameter to be ‘polynomial’, and the ‘degree’ parameter to be 2. The reason we want the degree to be 2 is because we want to train a quadratic kernel.

```
svm_degree_2 = SVC(kernel='poly', degree=2)
svm_degree_2.fit(X,y)
```

**Degree = 2**  
**Accuracy: 0.8909090909090909**

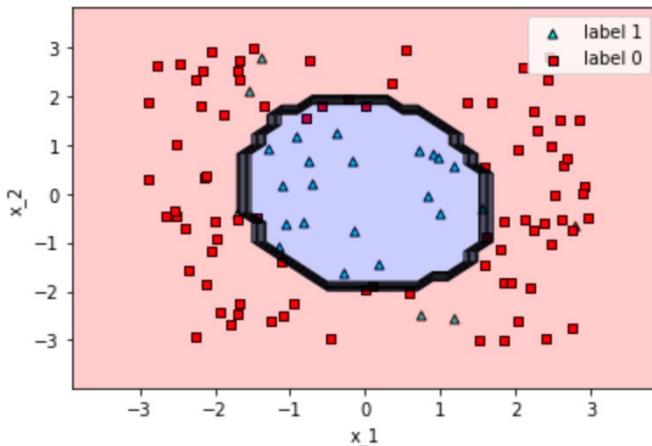


Figure 9.30. An SVM classifier with a polynomial kernel of degree 2. Notice that it draws a circular region, which is given by the polynomial kernel.

Notice that it manages to build a circular region to bound the dataset.

#### **CODING THE RBF KERNEL TO CLASSIFY A DATASET FORMED BY TWO INTERSECTING CIRCLES**

Ok, we’ve drawn a circle, but let’s get more complicated. In this subsection I’ll show you how to code several SVMs with the rbf kernel in order to classify a dataset that has the shape of two intersecting circles. This dataset is called ‘two\_circles.csv’ and it is illustrated in Figure

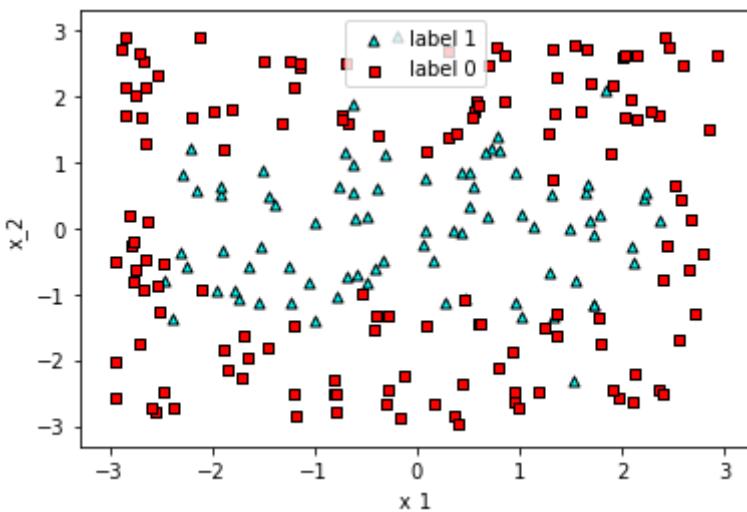


Figure 9.31. A dataset consisting of two intersecting circles, with noise. We will use an SVM with the rbf kernel to classify this dataset.

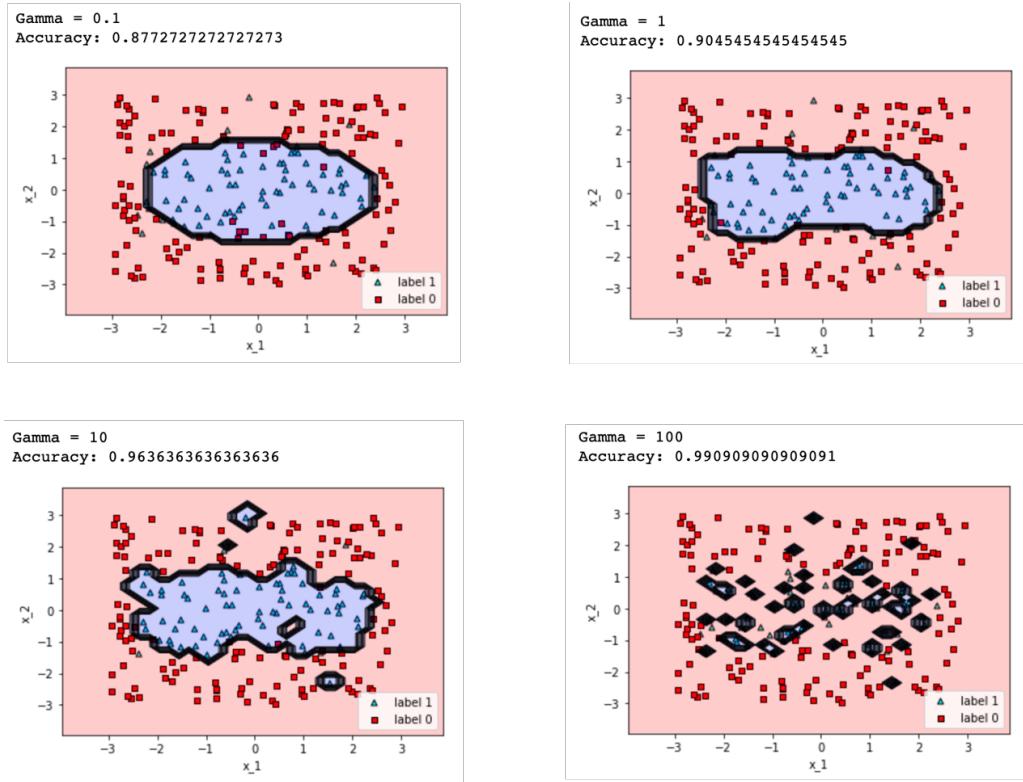
In order to use the rbf kernel, we simply specify ‘kernel = ‘rbf’’. We can also specify a value for gamma. We’ll train four different SVM classifiers, for the following values of gamma: 0.1, 1, 10, and 100.

```
# gamma = 0.1
svm_gamma_01 = SVC(kernel='rbf', gamma=0.1)
svm_gamma_01.fit(X, y)

# gamma = 1
svm_gamma_1 = SVC(kernel='rbf', gamma=1)
svm_gamma_1.fit(X, y)

# gamma = 10
svm_gamma_10 = SVC(kernel='rbf', gamma=10)
svm_gamma_10.fit(X, y)

# gamma = 100
svm_gamma_100 = SVC(kernel='rbf', gamma=100)
svm_gamma_100.fit(X, y)
```



**Figure 9.32.** Four SVM classifiers with an rbf kernel and different values of gamma. Notice that the classifier with gamma = 0.1 underfits, as it doesn't capture the shape of the data. The classifier with gamma = 1 is good, as it classifies the data well but it doesn't get distracted by noise. The classifier with gamma = 10 starts to overfit, as it starts to try to classify noise correctly. The classifier with gamma = 100 vastly overfits, as it simply surrounds every triangle, and classifies everything else as a square.

The original boundary is a union of two circles. Notice that for gamma = 0.01, the model underfits a little, since 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 there is some noise, and it tries to classify it correctly, encircling small pockets of points. By the time we get to gamma=100, we can see some serious overfitting. This classifier only encircles the triangles, and classifies everything else as a square.

## 9.4 Summary

- A support vector machine (SVM) is a classifier that consists of fitting two parallel lines (or hyperplanes), and trying to space them apart as much as possible, while still trying

to classify the data correctly.

- The way to build support vector machines is with an error function that's similar to the log-loss. This new error function has an extra term which is high when the distance between the two parallel lines is small. This way, an SVM separates the data with boundaries that are well spaced from the points.
- The C parameter is used 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 very useful and very powerful method used to build non-linear 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.
- There are several different kernels, such as the polynomial kernel and the rbf kernel. The polynomial kernel allows us to build polynomial regions such as circles, parabolas, hyperbolas, etc. The rbf kernel allows us to build curved regions comprising circles.

# 10

## *Combining models to maximize results: Ensemble learning*

### This chapter covers

- What is ensemble learning.
- Joining several weak classifiers to form a strong classifier.
- Bagging: A method to randomly join several classifiers.
- Boosting: A method to join several classifiers in a smarter way.
- AdaBoost: A very successful example of boosting methods.

After learning many interesting and very useful machine learning classifiers, a good question to ask is "Is there a way to combine them?". Thankfully the answer is yes! In this chapter we learn several ways to build stronger classifiers by combining weaker ones. The methods we learn in this chapter are bagging and boosting. In a nutshell, bagging consists on constructing a few classifiers in a random way and putting 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. One of the most popular examples of boosting is the AdaBoost algorithm (Adaptive Boosting), which we study at the end of the chapter.

### 10.1 With a little help from our friends

Here is the scenario. You have to take an exam that consists of 100 true/false questions on many different topics, such as math, geography, science, history, music, and so on. Luckily, you

are allowed to call your five friends, Alice, Bob, Carlos, Dana, and Emily to help you. What are some techniques that you can use to get their help? Let me show you two techniques that I can think of.

**Technique 1:** You send the exam to each of the five friends, and ask them to fill it in. Then you get the responses, and for each question, you make them vote. For example, if for question 1, three of your friends answered "True" and two answered "False", you answer that question as "True". We may still get some wrong, but if our friends are of the same level of knowledge and intelligence as us, we can imagine that the five of them together are likely to do better than only one of them.

**Technique 2:** We give the exam to Alice, and ask her to answer the questions that she is the most sure about. She answers a few of them. We assume that those answers are good, since we focused on Alice's strengths. Then we pass the remaining questions to Bob, and follow the same procedure with Carlos and Dana. For our last friend, Emily, we just ask her to answer all the remaining ones. This is a good technique as well, especially if our friends are experts in different disciplines.

These are two examples of combining our friends to form a super-friend, who will likely do well in tests. The equivalent scenario in machine learning is when we have several classifiers that classify our data well, but not great, and we'd like to combine them into a super-classifier that classifies our data very well. This discipline is called *ensemble learning*.

We call the set of classifiers *weak learners*, and the super-classifier they form when combined a *strong learner*.

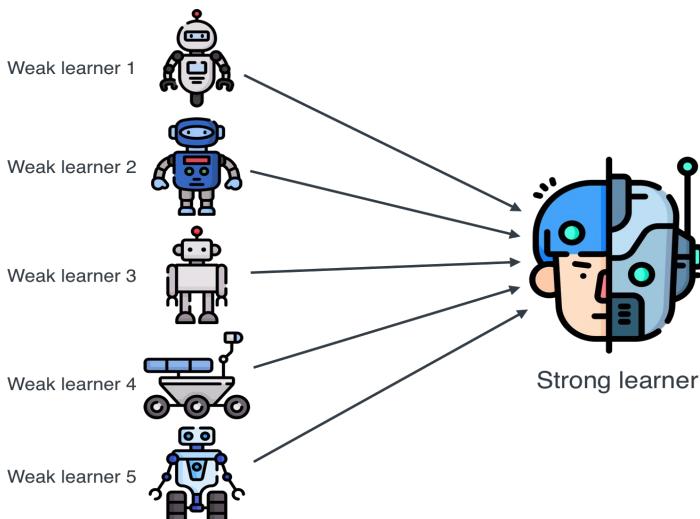


Figure 10.1. Ensemble methods consist of joining several weak learners in order to build a strong learner.

In this chapter we learn two ensemble learning techniques called bagging and boosting, which very much resemble the two previous techniques, respectively. In a nutshell, this is what bagging and boosting do:

**Bagging:** We train a bunch of different classifiers on different random subsets of our data. We join them into a big classifier, which makes predictions by voting. In other words, the resulting classifier predicts what the majority of the other classifiers predict.

A canonical example of a bagging model is *random forests*. Simply put, you pick a random subset of your data and build a decision tree that fits that subset. Then you do the same thing with another random subset of data. You continue in this fashion, building a set of decision trees. Then you build a classifier joining all these decision trees by making them vote. Since the trees were built on random subsets of your data, we call this a random forest.

Why the name ‘bagging’? Well, there is no bag involved; bagging is short for Bootstrap AGGREGatING.

**Boosting:** Boosting is very similar to bagging, except the classifiers are not picked at random. Details vary between algorithms, but in a nutshell, each classifier is picked in a way that focuses on the weaknesses of the previous classifiers. Each classifier is not necessarily strong, but the union of them is. One way to do this is to modify our data by giving more weight to the misclassified points and less to the correctly classified ones, and fit the next classifier to this modified data.

The canonical example of boosting that we learn in this chapter is AdaBoost (ADaptive BOOSTing). I encourage you to further your studies with other algorithms such as gradient boosted trees, and XGBoost (eXtreme Gradient Boosting).

Most ensemble methods in this chapter use decision trees as the weak learners. Many ensemble methods started in order to prevent overfitting in decision trees, and for this reason decision trees tend to be more popular for this kind of approaches. However, as you read this chapter, I encourage you to look at how the strong learners would look if the weak learners were other types of classifiers, such as perceptrons, SVMs, and so on.

## 10.2 Why an ensemble of learners? Why not just one really good learner?

When I suggested combining several different weak learners to form a strong learner, a question may have popped into our minds. We’ve spent an entire book learning how to build strong learners, why all of a sudden do we want to combine them? For example, if we are going to combine a few simple decision trees, why not build one very robust one? Let me illustrate why with a small example.

Let's go back to our spam example in Chapter 7 (Decision Trees). To remind you, the dataset consisted of spam and ham emails, and the features were the number of times the words 'lottery' and 'sale' appeared on the email (Table 10.1).

**Table 10.1.** 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	1	No
3	2	No
3	9	No
1	3	No
2	6	No
4	7	No
1	9	Yes
3	10	Yes
6	5	Yes
7	8	Yes
8	4	Yes
9	6	Yes

This table is plotted in Figure 10.2., where the triangles represent spam emails, and the squares represent ham emails.

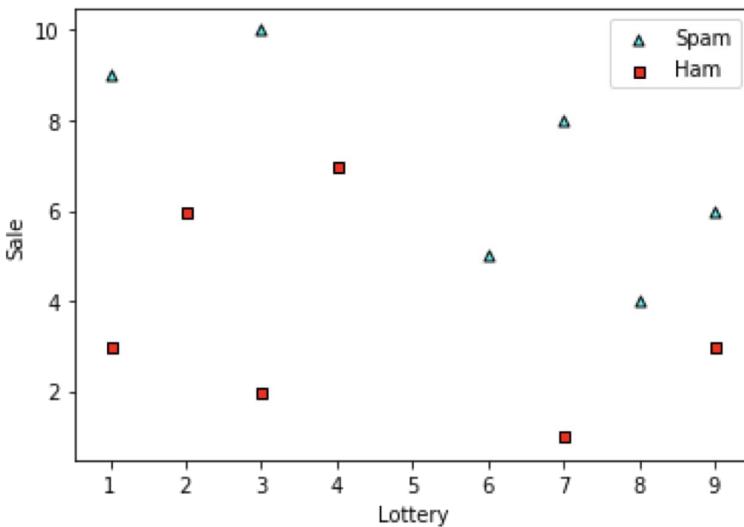


Figure 10.2. The plot of our dataset, where spam emails are triangles and ham emails are squares. In the horizontal axis we have the number of appearances of the word 'lottery', and in the vertical axis, the number of appearances of the word 'sale'..

For this dataset, we trained a decision tree which created a broken linear boundary, as illustrated in Figure 10.3.

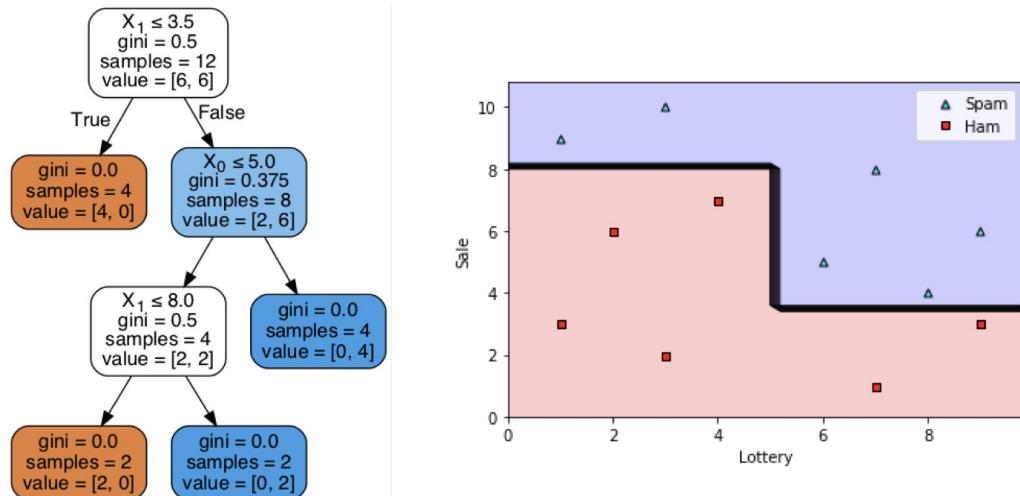
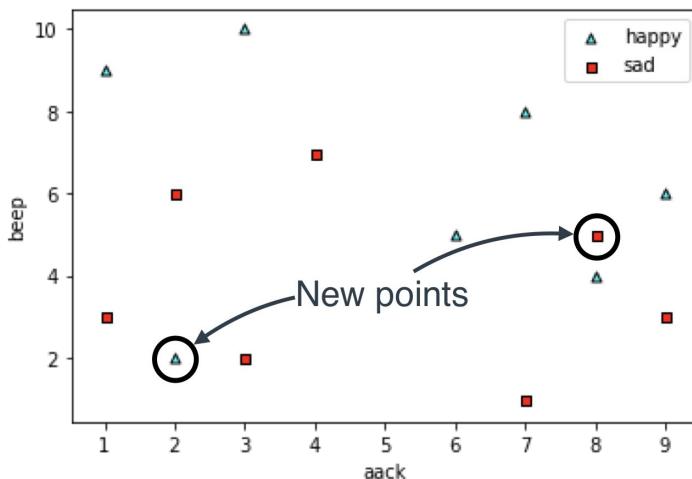


Figure 10.3. Left: A decision tree that classifies our dataset. Right: The boundary defined by this decision tree. Notice that it splits the data very well.

So far, so good. That tree does a great job splitting our data. But what happens if we add the following two more points?

**Table 10.2.** The new data points we are adding to the email dataset.

Buy	Lottery	Spam
8	6	No
2	2	Yes

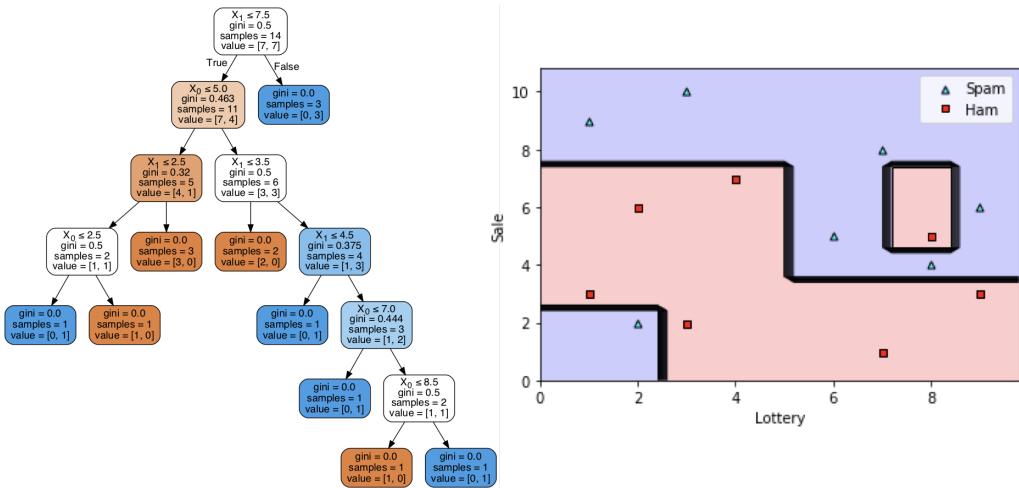


**Figure 10.4.** The plot with the two new data points added.

Now let's try to fit a decision tree to this data. We can use `sklearn`, like we did in Chapter 7, with the following command in `sklearn`:

```
spam_decision_tree = DecisionTreeClassifier()
spam_decision_tree.fit(new_X,new_y)
```

Feel free to look at more details in the repo [www.github.com/luisquiserrano/manning](https://www.github.com/luisquiserrano/manning). The decision tree and the boundary region can be seen in Figure 10.5.



**Figure 10.5.** 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, since the two isolated points would rather be treated as errors.

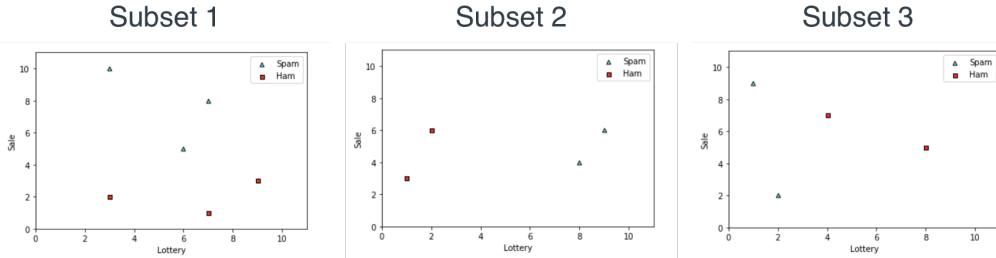
There seems to be some overfitting going on. Those two new points that we added should be considered noise, but instead the tree went out of its way to classify them correctly. As we saw before, decision trees are very prone to overfitting. Could it be that random forests give us a nicer boundary? We'll see in the next section!

## 10.3 Bagging - Joining some classifiers together to build a stronger classifier

Bagging is a technique in which we build a strong learner based on a set of weak learners. The way the strong learner makes a prediction is simply by allowing the weak learners to vote. Whichever prediction gets more votes, is the one the strong learner makes. You can imagine the weak learners to be any type of classifier, but in this chapter, we'll use decision trees. And going by that analogy, a set of decision trees is called a forest. Since there is some randomness involved in building these trees, we'll call it a random forest.

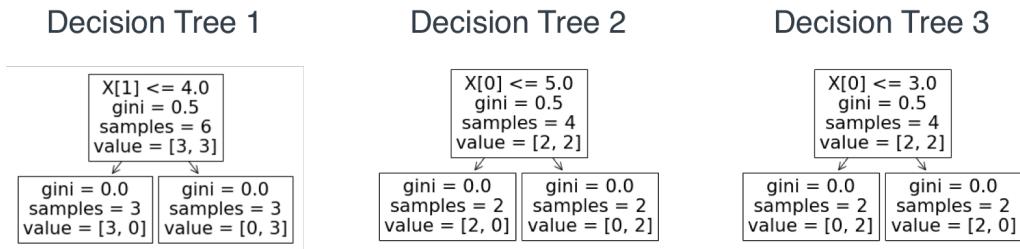
### 10.3.1 Building random forests by joining several trees

Let's try to fit three trees to the data in Figure 10.4. In order to make things computationally easier, we'll split the data into three random (almost) equal subsets, and we'll fit a simple (depth one) tree to each dataset.



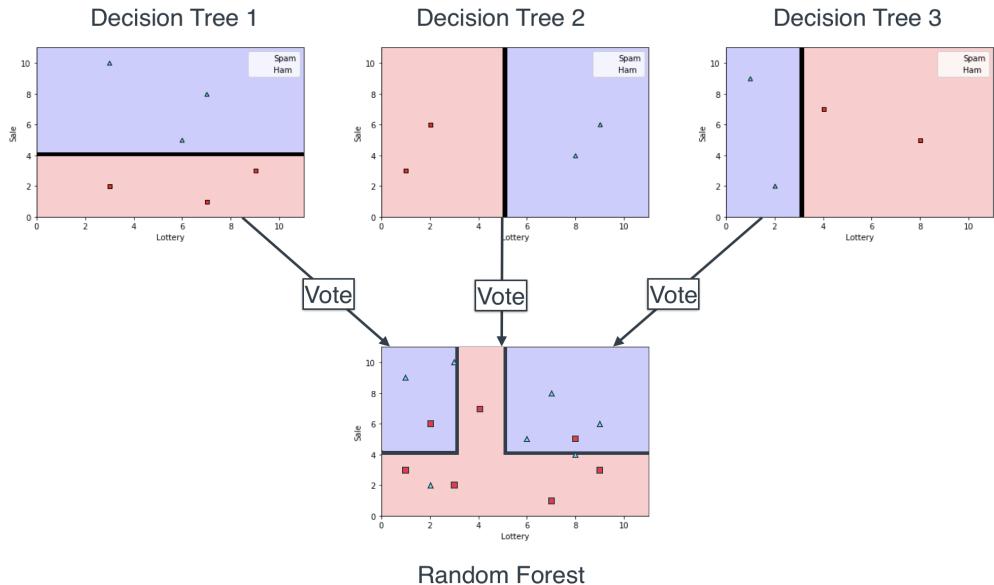
**Figure 10.6.** Splitting our data into three subsets. Notice that the subsets need not be disjoint, and need not cover the entire data set..

Now, we fit a decision tree in each one of them. Using sklearn, we get the three trees in Figure 10.7.



**Figure 10.7.** Three decision trees, each one fitting each of the subsets in Figure 10.6.

But most importantly, the trees define the boundaries in the top of Figure 10.8. And now, we join these three classifiers into one random forest classifier by making them vote. This simply means that when a new email comes in, we check the predictions of the three decision trees. Whichever prediction got two or more votes, whether it's spam or ham, is the prediction that the random forest makes. The boundary of the random forest is illustrated at the bottom of Figure 10.8.



**Figure 10.8.** On top, we can see the three boundaries of the decision trees from Figure 10.7. On the bottom, we can see how the three decision trees vote, to obtain the boundary of the corresponding random forest..

Notice that the random forest is a good classifier, as it classifies most of the points correctly, but it allows a few mistakes in order to not overfit the data.

### 10.3.2 Coding a random forest in sklearn

Now you may be thinking “that way you partitioned the data was a bit convenient, what if you don’t get such nice subsets?” You are right, let’s allow sklearn to build a random forest, and see how it compares. We will build one with five decision trees, or ‘estimators’. The command is the following:

```
from sklearn.ensemble import RandomForestClassifier
random_forest_model = RandomForestClassifier(random_state=0, n_estimators=5)
random_forest_model.fit(new_X,new_y)
random_forest_model.score(new_X,new_y)
```

When we plot the boundary defined by this random forests, we get Figure 10.9.

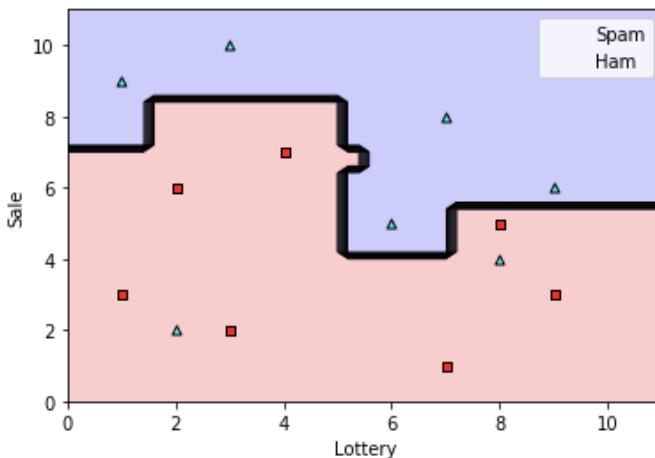


Figure 10.9. The boundary of the random forest obtained with sklearn.

Which seems like a nice boundary. But just out of curiosity, let's plot the boundaries of the estimator trees. They are in Figure 10.10.

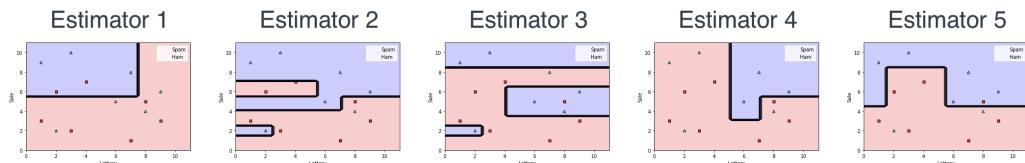


Figure 10.10. The 5 estimators (learners) obtained from sklearn.

Yikes! These are quite complex models, as a matter of fact, some seem to overfit, like Estimator 2. In the repo I have printed out all the decision trees, and you can see that estimators 2 and 3 have depth 6. But somehow, when we overimpose them together, we get a clean boundary. The beauty of machine learning.

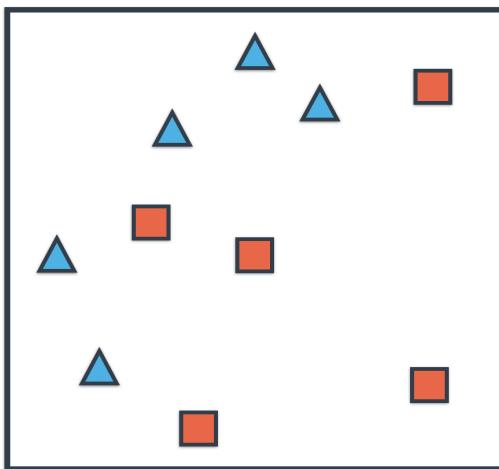
## 10.4 Boosting - Joining classifiers together in a smarter way to get a stronger classifier

Boosting is very similar to bagging, except now we don't select the weak learners at random, but we select them in a more intelligent way. The way we do this is by training each learner to focus on the weaknesses of the previous ones. In other words, each learner tries really hard to correctly classify the points in which the previous classifiers. How do we do this? We start by training the first learner. We now look at which data points did the learner classify correctly. We shrink them by weighting them by a number smaller than 1. Then we look at the data points

that the learner didn't classify correctly. We enlarge them by weighting them by a number larger than 1. Now we have a new weighted dataset, where the errors of the first learner are weighted more heavily than the rest of the points. We now fit a second learner on this new dataset. The second learner, naturally, will try to fit the errors of the previous learner better. After the second learner, we now reweight the data based on which points it classified correctly and incorrectly. We now build a third learner on this new weighted dataset. We continue in this fashion, until we end up with a set of learners, each focusing on the previous learners' weaknesses. As a final step, we make the classifiers vote. The voting can be weighted if we need to. The best way to learn boosting is to look at a very famous example of it: AdaBoost.

AdaBoost (Adaptive Boosting), developed by Freund and Shapire in 1997, is a very powerful boosting algorithm that has produced great results. In AdaBoost, all the weak learners are the simplest possible learner one can have: a decision tree of depth one, or a stump. In our example, a stump is represented by either a vertical or a horizontal line that splits our data. It is the classifier that picks only one of the features, and classifies the data based on if that feature is smaller or larger than some threshold.

I will show you AdaBoost in a small example. We'll try to classify the data in Figure 10.11.



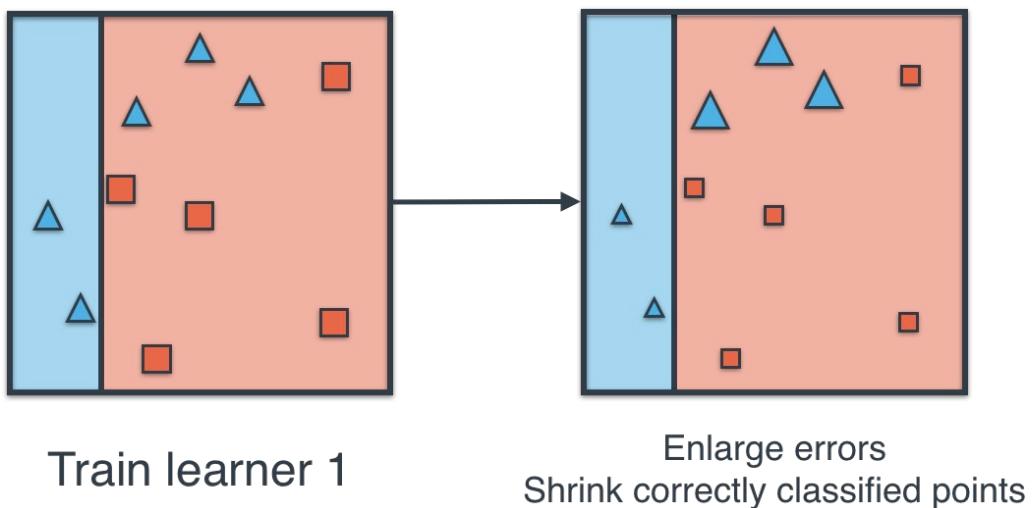
**Figure 10.11.** The data set that we will classify next.

First, we'll do it conceptually, and next I will add some numbers.

#### 10.4.1 A big picture of AdaBoost

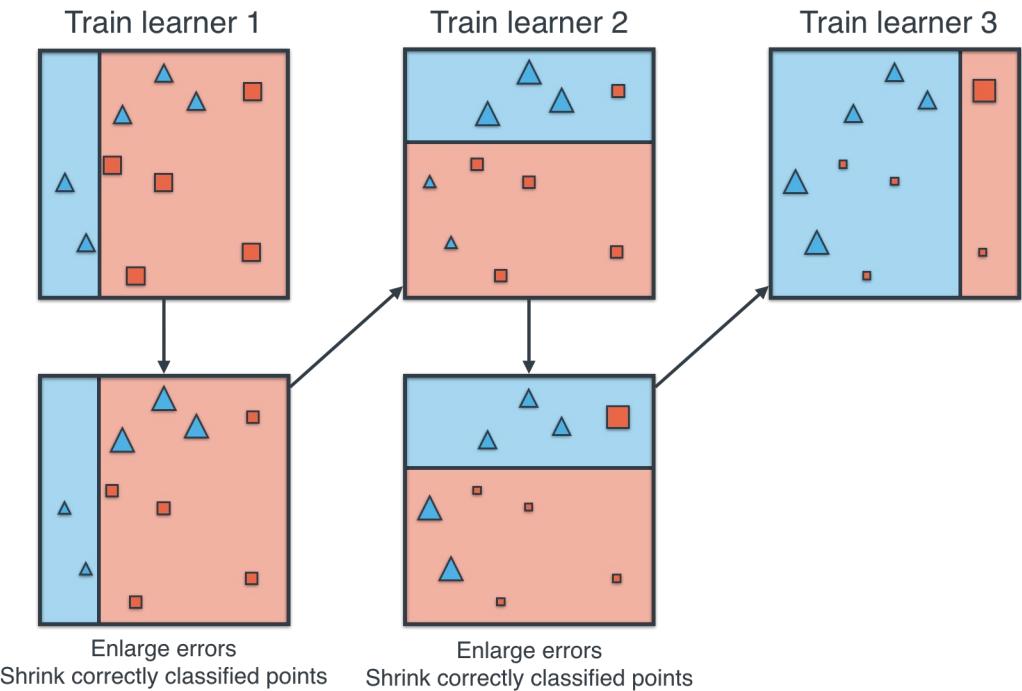
First let's try to fit a decision tree of depth one. That is simply a vertical or horizontal line. There are a few that work, so let's pick the vertical line in the left of Figure 10.12, which correctly classifies the two triangles in the left, and all the squares. That classifier is weak learner 1.

The next step is rescaling our data. We'll enlarge the points that are misclassified, and shrink the ones that are correctly classified. You can imagine this as follows: In the beginning, each point has weight 1. After rescaling the data, some points have weights larger than one, and some have weights smaller than one. In the figures, we enlarge or shrink each data point to illustrate this. The data now looks like the right side of Figure 10.12.



**Figure 10.12.** Left: The first learner. Right: The rescaled dataset, where we have enlarged the misclassified points, and shrunk the correctly classified points..

Now, we simply continue this process. On the rescaled data, we train a second learner. This second learner will be different, because this one tries harder to classify the bigger points (those with larger weight), and doesn't worry so much about the small ones (those with smaller weight). After training this learner, we again enlarge the data accordingly. We repeat this process a third time, and then we decide to stop (we could keep going if we wanted to). The process is illustrated in Figure 10.13.



**Figure 10.13.** The whole AdaBoost process in our data set. First we train the first weak learner (top), next we rescale the points (bottom), and so on as we move to the right.

Now, as a final step, we combine the three learners into one by voting, as illustrated by Figure 10.14.

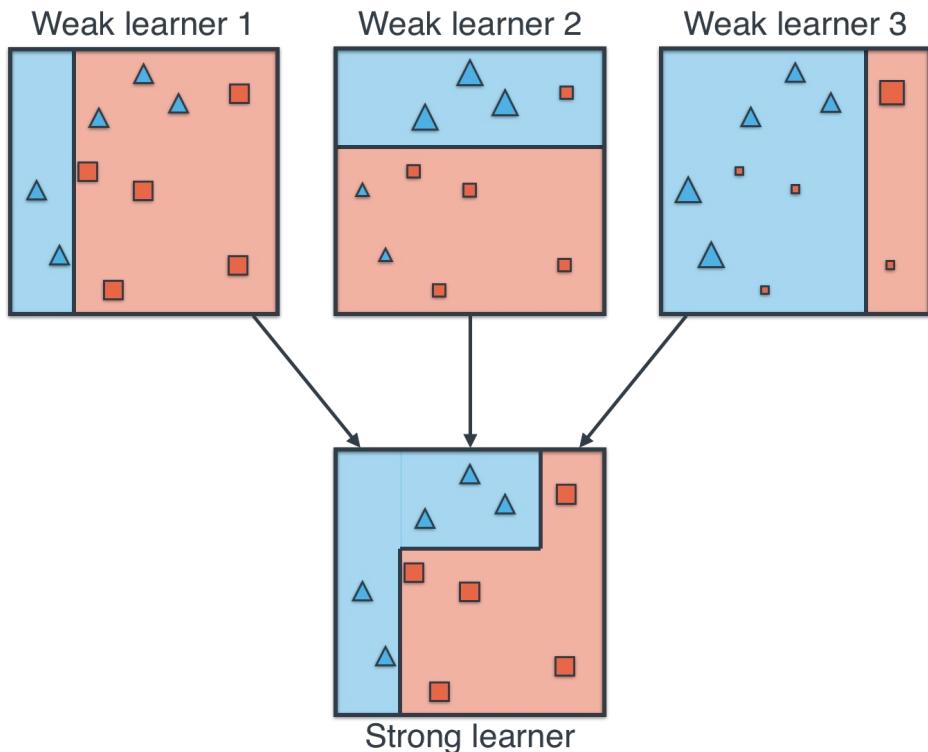


Figure 10.14. Making the three weak learners vote. The resulting strong learner is the one at the bottom.

Ok, that's the big picture explanation of AdaBoost. Let's throw some math at it now.

#### 10.4.2 A detailed (mathematical) picture of AdaBoost

In the previous section I showed you two steps of the algorithm, namely training new learners and rescaling the data. Rescaling data is done in two steps:

1. Picking a number that is larger than 1.
2. Multiplying all the errors by this number.

#### **THE DIFFERENCE BETWEEN PROBABILITY AND THE ODDS RATIO**

In order to come up with the rescaling factor, first let's digress a bit into the difference between probability and odds. Let's say we have a box with three balls, two red and one blue, and we draw one random ball from the box. The probabilities of obtaining a ball of each color are the following:

- $P(\text{red ball}) = \frac{2}{3}$ .
- $P(\text{blue ball}) = \frac{1}{3}$ .

However, one would like to say something along the lines of "It's twice as likely to draw a red ball than to not draw a red ball". For this, we use *odds*. We say that the odds of drawing a red ball are 2, and the odds of drawing a blue ball are  $\frac{1}{2}$ . If the formula we used for probability was

$$P(\text{red ball}) = \frac{\# \text{ red balls}}{\# \text{ balls}},$$

Then the formula for odds (the odds ratio--OR) is

$$\text{OR}(\text{red ball}) = \frac{\# \text{ red balls}}{\# \text{ blue balls}}.$$

**NOTE:** The odds ratio (OR) shouldn't be confused with the OR logical operator.

Notice that since the total number of balls is  $\# \text{balls} = \# \text{red balls} + \# \text{blue balls}$ , then we can conclude that

$$\text{OR}(\text{red ball}) = \frac{\# \text{ red balls}}{\# \text{ balls} - \# \text{red balls}}.$$

From here, we can see that in general, probability and odds are related via the following equation:

$$\text{OR} = \frac{P}{1 - P},$$

where OR is the odds ratio. In the previous example, if the probability of a red ball is  $P(\text{red ball}) = \frac{2}{3}$ , then the odds ratio is:

$$\text{OR}(\text{red ball}) = \frac{P(\text{red ball})}{1 - P(\text{red ball})} = \frac{\frac{2}{3}}{1 - \frac{2}{3}} = 2.$$

The odds ratio is widely used in many areas in particular in science and betting. For our case, we'll be using it to build our rescaling factor.

### CALCULATING THE RESCALING FACTOR

After we train a learner, this learner may make some mistakes. We need to blow up these mistakes, namely, multiply them by a number that is larger than one, so that the next classifier focuses more on them, and less on the points that are well classified. For reasons that we'll see later, we want this factor to be large if our learner is good, and low if our learner is not that good.

Many metrics would work as a rescaling factor, but the one we pick is very related to the odds ratio. In fact, the rescaling factor is the odds that the learner classifies a point correctly.

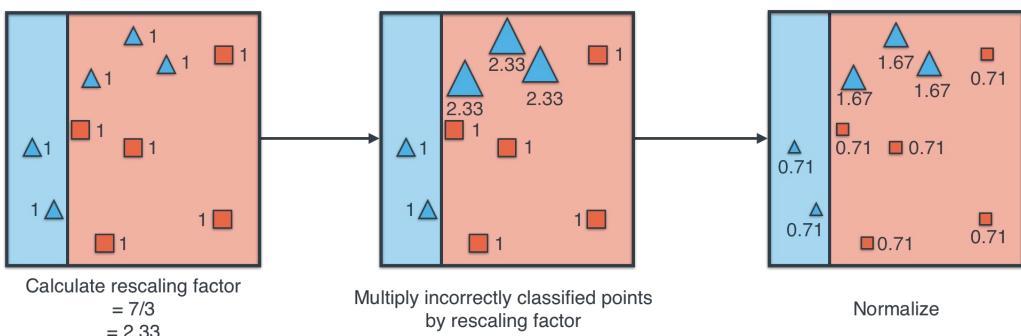
Sounds like a lot of work, but it's not so bad. Let's do it in the example. We begin by assigning to every point a weight of 1. Now, let's look at the accuracy of learner 1. It's correct for 7 points, and wrong for 3 points, so its accuracy is 0.7. But let's actually consider a different number. Let's look at the number of correctly classified points divided by the number of incorrectly classified points. In this case, it is  $7/3 = 2.33$ . That's our rescaling factor. Notice that the better the model, the higher the rescaling factor. Therefore, the following formula works.

$$\text{Rescaling factor} = \frac{\text{Number of correctly classified points}}{\text{Number of incorrectly classified points}}.$$

As a matter of fact, we can do better. Since we'll change the weights of the points during the process, the better way to formulate the rescaling factor is the following.

$$\text{Rescaling factor} = \frac{\text{Sum of weights of correctly classified points}}{\text{Sum of weights of incorrectly classified points}}.$$

Now we look at the three misclassified points in the model, and we multiply their weight by the rescaling factor  $7/3$ . Our weights are now  $7/3, 7/3, 7/3, 1, 1, 1, 1, 1, 1, 1$ . In order for them to ensure they still add to 10, let's divide them all by their sum, which is 14. Our new weights are  $\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, 1/14, 1/14, 1/14, 1/14, 1/14, 1/14, 1/14$ . This rescaling process is illustrated in Figure 10.15.



**Figure 10.15.** We scale the errors by  $7/3$ , and then we normalize everything so the sum of points is 10.

Notice that if we fit a new classifier on this dataset, it will try to classify those three big triangles correctly, since they carry more weight. The best classifier is on the left of Figure 10.16. In the same figure, we have repeated the calculation of the rescaling factor and the normalization.

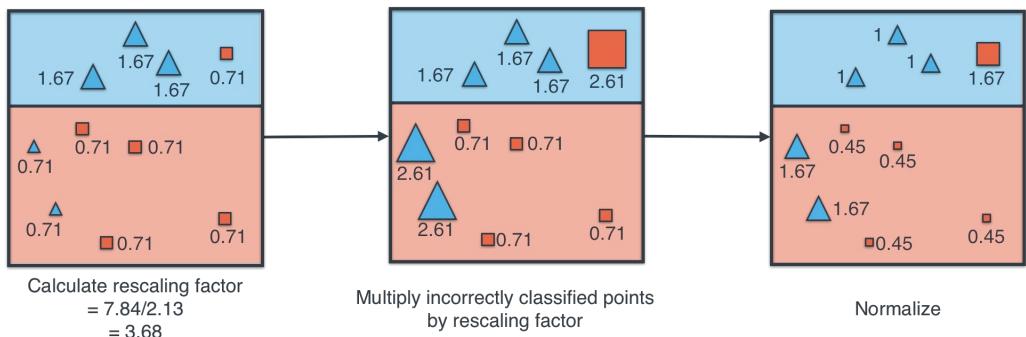
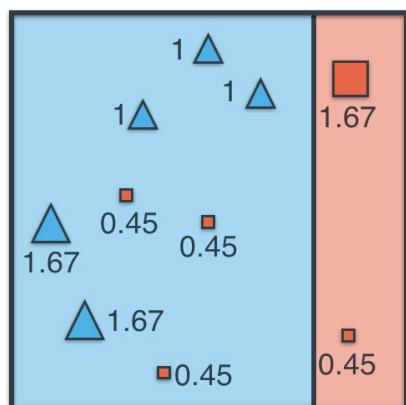


Figure 10.16. Again, we scale the errors by the rescaling factor and normalize.

And as a final step, the best learner for the normalized dataset is simply the vertical line in Figure 10.17. For this one we will calculate the rescaling factor (you'll see why in a bit), but we'll stop training more learners right here.



Calculate rescaling factor  
=  $8.46/1.35 = 6.27$

Figure 10.17. Our last weak learner. We don't need to scale the points anymore, since we are not building any more weak learners.

#### JOINING THE LEARNERS - THE LOGIT

Alright, we now have our three learners! But how do we put them together? In the last section we mentioned voting, but actually what we want is some kind of weighted voting, where the models that are doing very well get more of a say. We have a measure for how good models are, which is the rescaling factor, so what about using that one? For reasons that will be clear

soon, what we actually want is the logarithm of the rescaling factor, also called the *logit*. Allow me to elaborate.

Let's imagine that we have three friends, Truthful Teresa, Unpredictable Umbert, and Lying Lenny. Truthful Teresa almost always says the truth, Lying Lenny almost always lies, and Unpredictable Umbert says the truth exactly half of the time, and lies the other half. Here is a question, out of those three, who is the least useful one?

The way I see it, Truthful Teresa is tremendously reliable. As she almost always says the truth, when she tells us anything, we can be pretty sure that it's true. Among the other two, though, I prefer Lying Lenny. Since he almost always lies, we pretty much have to do the opposite of what he says, and we'll be correct most of the time! Unpredictable Umbert, however, is useless, since we don't know if he's telling the truth or lying.

In that case, if we were to assign a weight to what each friend says, we'd give Truthful Teresa a very high score, Lying Lenny a very high negative score, and Unpredictable Umbert a score of zero.

In machine learning, the equivalent for Truthful Teresa is a model with high accuracy, which correctly predicts points most of the time. The equivalent for Lying Lenny is a model with very low accuracy, and the equivalent to Unpredictable Umbert is a model with accuracy around 50%. Notice that between a model with accuracy around 50% and a model with terrible accuracy, one would actually prefer the one with terrible accuracy (if we are predicting only two classes). Why is this? Because a model that predicts the incorrect class most of the time can be turned into a model that predicts the correct class most of the time, by simply flipping the answer (the equivalent to listening to Lying Lenny and doing the exact opposite). On the other hand, a model that predicts the correct class around 50% of the time, is about as accurate as tossing a fair coin; it gives us no information.

Therefore, we need to find a metric which is high (positive) for high accuracy models, high (negative) for low accuracy models, and around zero for models with 50% accuracy. Here are some examples of what we need:

Truth 99% of the time: Very high score.

Truth 70% of the time: Some positive score.

Truth 50% of the time: Score of 0.

Truth 30% of the time: Negative score.

Truth 1% of the time: Very low negative score.

Maybe odds can help us, let's calculate the odds for each one of these cases.

Truth 99% of the time:  $99/1 = 99$ .

Truth 70% of the time:  $70/30 = 2.33$  (remind you of something?).

Truth 50% of the time:  $50/50 = 1$

Truth 30% of the time:  $30/70 = 0.43$ .

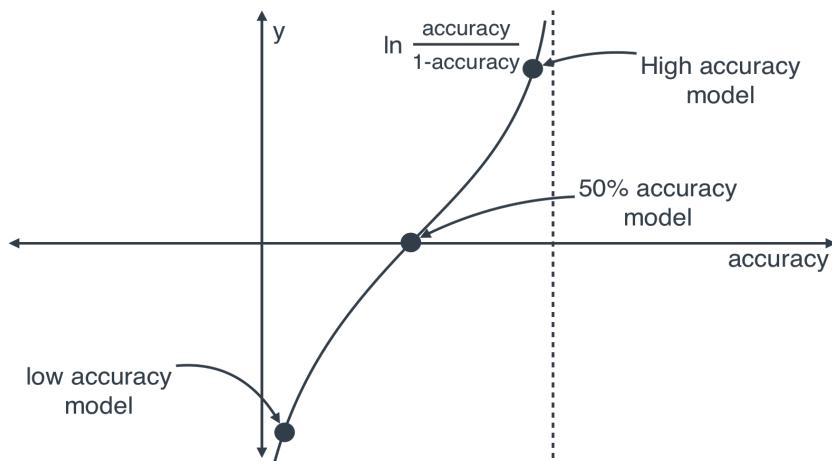
Truth 1% of the time:  $1/99 = 0.01$ .

We need a function that assigns negative scores to the very small numbers, that also assigns 0 to the value 1, and that also assigns positive scores to high numbers? What would this function be? If you said logarithm, that's correct! In Table 10.3 I have calculated the logarithm of each of the values.

**Table 10.3. The accuracy, the odds, and the natural logarithm of the odds, for several different values.**

Accuracy	Odds = #Correct/#Errors	In(Odds)
99%	$99/1 = 99$	4.595
70%	$70/30 = 2.333$	0.8473
50%	$50/50 = 1$	0
30%	$30/70 = 0.4286$	-0.8473
1%	$1/99 = 0.0101$	-4.595

The logarithm of the odds, which is commonly known as logit (short for "logistic unit"), is the weight that we assign to each model in the voting. In Figure 10.18 we can see a plot of this.

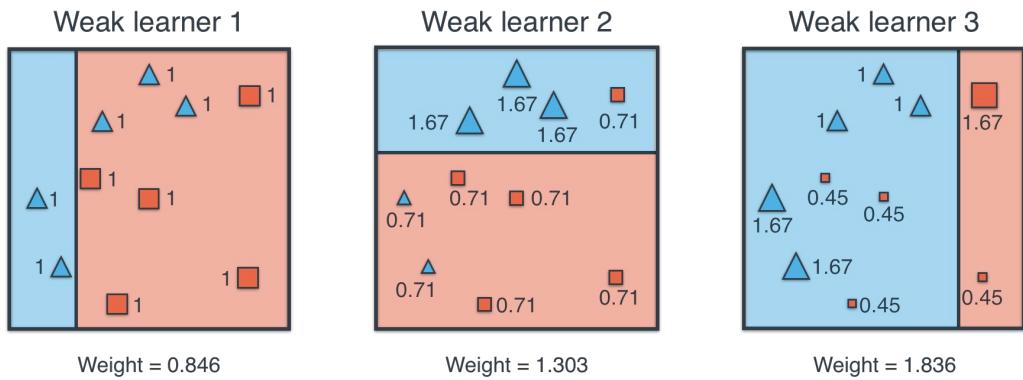


**Figure 10.18. The curve shows the plot of  $\ln(\text{odds})$  with respect to the accuracy. Notice that for small values of**

the accuracy, the  $\ln(\text{odds})$  is a very large negative number, and for higher values of the accuracy, the  $\ln(\text{odds})$  is a very large positive number). When the accuracy is 50% (or 0.5), the  $\ln(\text{odds})$  is precisely zero.

Now we are ready to make the learners vote. Based on the odds, let's calculate the logarithm of the odds of each of the three learners.

- Learner 1:
  - Odds:  $7/3 = 2.33$
  - $\log(\text{odds}) = 0.846$
- Learner 2:
  - Odds:  $7.84/2.13 = 3.68$
  - $\log(\text{odds}) = 1.303$
- Learner 3:
  - Odds =  $8.46/1.35 = 6.27$
  - $\log(\text{odds}) = 1.836$



**Figure 10.19.** We use the  $\ln(\text{odds})$  to calculate the weights of the three weak learners.

The way they vote is illustrated on Figure 10.20. Basically it weights each learner by its weight, and for each point, we add the weight if the learner classified the point as positive, and subtract the weight if it classified it as negative. If the resulting sum is positive, we classify the point positive, and if it is negative, we classify it as negative.

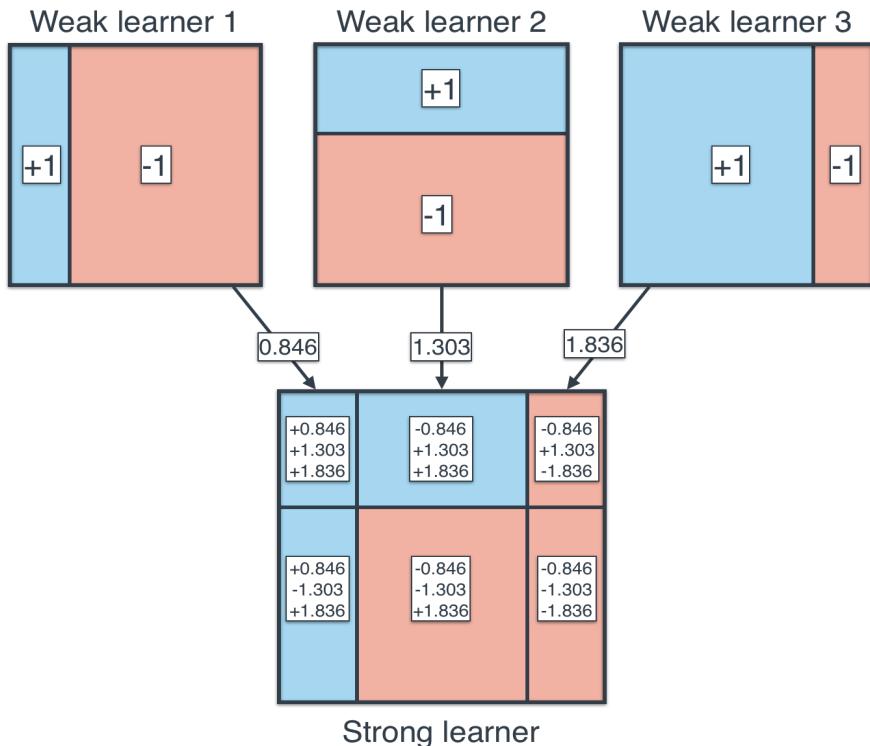


Figure 10.20. We weight each of the weak learners, and make them vote based on this weighting (so the larger the weight, the more voting power that particular learner has).

### 10.4.3 Coding AdaBoost in Sklearn

Let's use our original dataset, and train an AdaBoost classifier. We specify the number of learners as 6 using the parameter `n_estimators`. The model we plot is in Figure 10.21.

```
from sklearn.ensemble import AdaBoostClassifier
adaboost_model = AdaBoostClassifier(random_state=0, n_estimators=6)
adaboost_model.fit(new_X, new_y)
```

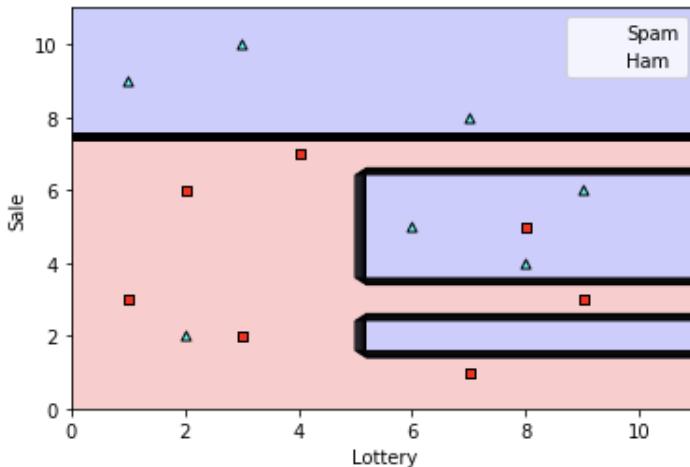


Figure 10.21. The boundary of the AdaBoost strong learner given by sklearn.

We can go a bit farther and actually explore the six learners we've used (Figure 10.22), using the following commands. Furthermore, the command estimator\_weights will help us look at the weights of all the learners.

```
estimators = adaboost_model.estimators_
for estimator in estimators:
    plot_model(new_X, new_y, estimator)
    plt.show()
```

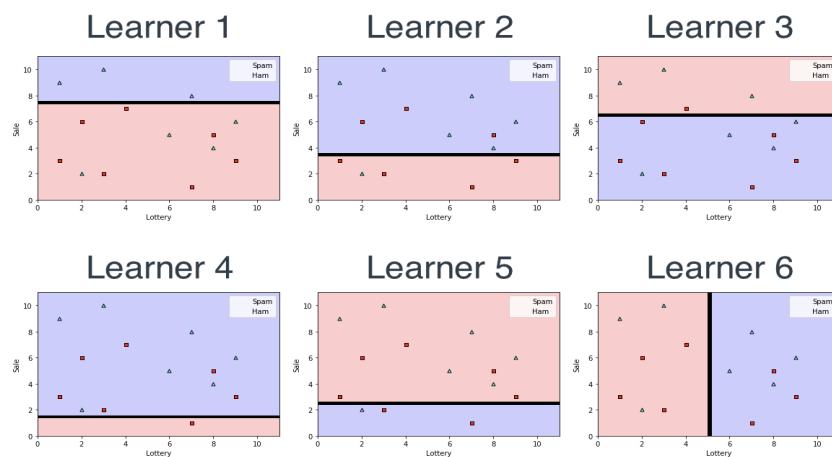


Figure 10.22. The boundaries of the six weak learners given by sklearn.

Notice that when the weak learners in Figure 10.22 vote, we get the strong learner in Figure 10.21.

## 10.5 XGboost - An extreme way to do gradient boosting

After seeing the great results of AdaBoost, it seems that building weak learners in a smart way, rather than randomly, pays off. Noticing that the weak learners in AdaBoost are all tree stumps (trees of depth 1), one can only wonder, what if we built an algorithm similar to AdaBoost, but with deeper trees?

Enter eXtreme Gradient Boosting (XGBoost), a very powerful algorithm which does precisely that, it builds a set of trees in succession, each one designed to focus on the weaknesses of the previous ones. XGBoost works both for regression and classification. In this book I will show you the algorithm in a regression example, but the algorithm for regression is not very different. But before, allow me to introduce some concepts.

### 10.5.1 XGBoost similarity score

Recall that the way we built decision trees in Chapter 7 was to pick a particular metric (accuracy, Gini index, etc.), and every time we split the elements in a node, we do it in a way that optimizes this metric. XGBoost is no different, and the metric we want to optimize here is similarity. Similarity, as the name implies, is simply a measure of how similar elements in a set are. Let's do a small exercise, among the following three sets, which one do you think 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, you are correct. Certainly a set with many elements being the same, is very similar. What about sets 1 and 3? That's hard to tell, right? I would say that Set 3 is more similar than Set 1, since the elements in Set 1 are all vastly different. However, Set 3 only has one element, so in some way it must have less similarity than Set 2, since the element in Set 3 has no one else to be similar with.

In order to quantify similarity, consider the following metric. Given a set  $\{a_1, a_2, \dots, a_n\}$ , the similarity score is the square of the sum of the elements, divided by the number of elements.

Let's calculate similarity for the three sets above:

- Set 1:  $Similarity = \frac{(10 - 10 + 4)^2}{3} = \frac{16}{3}$ .
- Set 2:  $Similarity = \frac{(7 + 7 + 7)^2}{3} = 147$ .
- Set 3:  $Similarity = \frac{7^2}{1} = 49$ .

Note that as we expected, the similarity score of Set 2 is the highest, and the similarity of Set 1 is the lowest. Note that this similarity score gives higher scores to larger sets, but the highest scores it gives is when most of the elements in the set are either positive or negative. Set 1 has a 10 and a -10 that cancel out, which reduces the similarity score vastly.

**NOTE:** This similarity score is not perfect. One can argue that the set {1, 1, 1} is just as similar as {7, 7, 7}, yet the similarity score of {1, 1, 1} is  $\frac{1}{3}$ , which is much less. However, for the purposes of our algorithm, this still works. This only says that the algorithm gives priority to higher numbers, so the values {7, 7, 7} will be separated into a leaf much higher up in the tree than the values {1, 1, 1}. However, as we will see, this step will make the values {1, 1, 1} change, and they will be prioritized soon after.

### 10.5.2 Building the learners

In order to illustrate the algorithm, we use the same example from Section 7.10 (Decision Trees for Regression), shown in Table 10.4. This dataset contains users with ages 10 to 80 years old, and for each user, the level of engagement with an app, from 1 to 7 days a week. Our goal is to build an XGBoost model that predicts the level of engagement from the age of the user.

**Table 10.4. The same table of customers as in Table 7.12 in Chapter 7. In the first row we record the age of the user in years where we record their age in years (the feature). In the second row we record their engagement with the app as the number of days in which they used it in the previous week (the label).**

User (feature)	Engagement (label)
10	7
20	5
30	6
40	0
50	1
60	0

70	4
80	3

### **BUILDING THE FIRST WEAK LEARNER AND CALCULATING THE RESIDUALS**

As we did for bagging and for boosting, we must start with a weak learner. The weak learner in this case is simply a classifier that gives to each of the inputs, a predicted label of 0.5. This sounds very simple, as it is predicting that every user will engage  $\frac{1}{2}$  a day with the app, but the idea is that we'll be building much stronger learners soon. We now check how good this classifier is, by subtracting the label from the predicted label, and obtaining a residual. The residuals are illustrated in Table 10.5.

**Table 10.5.** The table of users and engagements with two new columns. The first column is the predicted label, which is 0.5 for all the users. The second column is the residual, which is the difference between the actual label and the predicted label.

Age (feature)	Engagement (label)	Predicted label	Residual
10	7	0.5	6.5
20	5	0.5	4.5
30	6	0.5	5.5
40	0	0.5	-0.5
50	1	0.5	0.5
60	0	0.5	-0.5
70	4	0.5	3.5
80	3	0.5	2.5

### **FITTING A TREE TO THE RESIDUALS BY MAXIMIZING THE SIMILARITY SCORE**

Now that we have our first learner, we are on to build the second learner. This tree will be a regression decision tree that fits the residual column. The way we build it is by splitting the data into two nodes, in a way that maximizes the gain in similarity score. This gain is calculated as follows: we add the similarity scores of both of the nodes, and subtract the similarity score of the root node.

First, let's calculate the similarity score of the root node.

$$\text{Similarity} = \frac{(6.5 + 4.5 + 5.5 - 0.5 + 0.5 - 0.5 + 3.5 + 2.5)^2}{8} = 60.5.$$

Now, on to split the node. As we did with decision trees, we split the users by age, which means we'll make cuts at 15, 25, 35, 45, 55, 65, and 75.

Let's pick one of these and calculate the similarity scores of both of the nodes. We'll pick 65. In this case, the left node contains the users that are 55 or younger (namely 10, 20, 30, 40, 50, and 60), and the node in the right contains the users that are older than 65 (namely 70, and 80). Their corresponding residuals are as follows:

- Left node: {6.5, 4.5, 5.5, -0.5, 0.5, -0.5}
- Right node: {3.5, 2.5}

Now we calculate the similarity scores of the set of values in each of the nodes.

- Left node:  $\frac{(6.5 + 4.5 + 5.5 - 0.5 + 0.5 - 0.5)^2}{6} = 42.17$
- Right node:  $\frac{(3.5 + 2.5)^2}{2} = 18$

The total similarity score for this split is the sum of these two, or 60.167. Now let's do the same and calculate the tidal similarity score for each of the splits. I encourage you to do the calculations yourself or write a small program that does it. The answers are in Table 10.6.

- Split at 15: Similarity Score = 76.57
- Split at 25: Similarity Score = 80.67
- **Split at 35: Similarity Score = 96.8**
- Split at 45: Similarity Score = 73
- Split at 55: Similarity Score = 64.53
- Split at 65: Similarity Score = 60.67
- Split at 75: Similarity Score = 60.57

Now, we subtract the similarity of the root node (60.5) to get the similarity gain:

- Split at 15: Similarity Gain = 16.07
- Split at 25: Similarity Gain = 20.16
- **Split at 35: Similarity Gain = 36.3**
- Split at 45: Similarity Gain = 12.5
- Split at 55: Similarity Gain = 4.03
- Split at 65: Similarity Gain = 0.17
- Split at 75: Similarity Gain = 0.07

As we can see, the best similarity gain is obtained splitting the data at 35 years old. If you look at the data, this makes sense, since those users 35 or less seem to have the highest engagement

level, while the older users are less engaged. This is our first node in the second learner. We continue building the tree like that. I encourage you to figure out the way to split the existing nodes yourself. The answer is in Figure 10.23, where the left node is split at age 15 with a maximum similarity gain of 1.5, and the right node is split at 65 with a maximum similarity gain of 12.03.

**NOTE:** As you can see, maximizing the total similarity score of the nodes is the exact same thing as maximizing the similarity gain, since one is obtained by the other one by subtracting a constant value. However, for future steps in the algorithm, it is better to deal with similarity gain than with total similarity score.

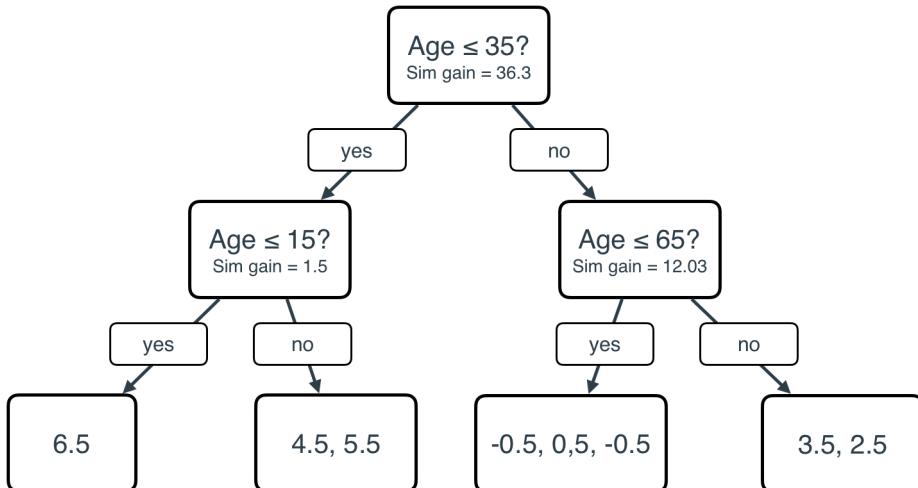


Figure 10.23. We have continued growing the tree. At each node, we can see the split and the maximum similarity gain that that split produced. In each we can see the labels of its elements.

#### **HYPERPARAMETERS - DECISIONS, DECISIONS, DECISIONS...**

Before we continue with the algorithm, it's time to introduce some hyperparameters. There are many hyperparameters in XGBoost that will certainly change the way the model is trained, and in this subsection I will show you some of the most important ones.

**Maximum tree depth:** One of the main selling points of XGBoost is that it overfits much less than decision trees. You can imagine that if you let trees grow to large depths, this leads to overfitting. Therefore, a very common hyperparameter is the maximum depth that we'll allow a tree to grow. For the current example, we'll let them grow to depth 2, but in the practice, they are normally allowed to have larger depths, such as 4 or more.

**Number of trees:** The number of trees is also an important hyperparameter. We'll use 3 in this example, but naturally, more trees are allowed in the practice.

**Minimum similarity gain:** If a splitting a node adds a large amount to the similarity score, then that is a good split. However, if it adds very little, we should consider if it's better to not make that split. In the running example, splitting on the left node gives us a similarity gain of only 1.5. Therefore, as we'll see very soon (when we talk about pruning), we'll decide to not perform this split, and consider the node a leaf. In the practice, one sets a threshold and only performs the splits where the similarity gain is higher than that threshold. In this example, we'll set this value to 3.

**Lambda:** In order to perform regularization and avoid overfitting, many times the similarity score is modified by adding a constant  $\lambda$  to the denominator. This will make the model less prone to creating leaves with very few elements. We won't see this in the running example (which means we are setting it to zero), but if you are using XGBoost to model your data, I encourage you to play with this parameter and notice the difference it makes.

**Minimum number of elements per leaf:** Just like with normal decision trees, we can set a threshold for the minimum number of elements we are willing to put in a leaf. For example, if a node has 20 elements, and the best split puts them into two leaves of 19 and 1 element, we can decide to avoid this split, since a leaf with 1 element hints at overfitting. Since this example is small, we will set this value at 1, which means we are always allowed to split a node, even if one of the leaves has only one element on it. As you can imagine, in the practice, this hyperparameter is set to a higher value.

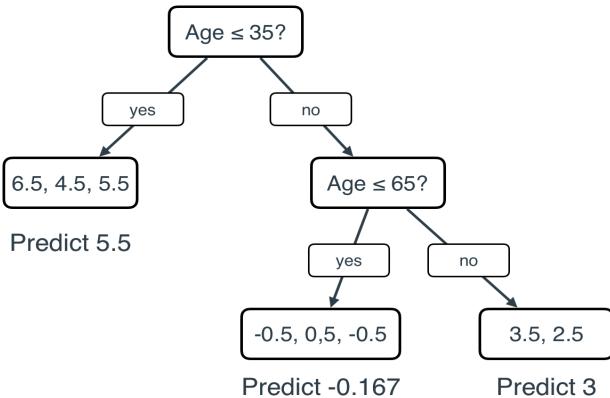
**Learning rate:** As we've learned in machine learning, the best way to go in order to avoid overfitting is to take small steps towards our goal. The learning rate is a small number, which we will multiply by the prediction made by each of our learners. In this example, we will set this value at 0.7, since we don't want to build too many trees, but in the practice, this is normally set to something much lower such as 0.1 or 0.3. We'll see this in more detail in the sections below when we update our predictions and residuals.

### TREE PRUNING

As we hinted in the previous section, we will not be performing splits that don't help us gain enough similarity score. In the tree in Figure 10.22, we can see that the split on the left node only helps us gain 1.5. Since we have set that hyperparameter to 5, and 1.5 is less than 5, then we won't perform this split. That means we prune that branch from the tree, obtaining the tree in Figure 10.23.

### MAKING THE PREDICTIONS

Now, we are ready to use this tree for predictions. The predictions are simple, we look at all the elements that ended up in a node, and for that node, we'll predict their average. Therefore, our tree with predictions looks like the one in Figure 10.24.



**Figure 10.24.** Our resulting tree after we pruned the left branch, since it didn't give us a high gain in similarity score. In each of the leaves, we have the elements that belong to it. Below the leaves, we show the prediction made at that leaf. Notice that this prediction is the average of the values in the leaf.

#### ***UPDATING THE PREDICTIONS, THE RESIDUALS AND BUILDING THE NEXT TREE***

We are almost done with our second tree. Now we update the predictions. Remember that the first prediction the model made was 0.5 (a terrible prediction). This second tree predicts the difference between the label and that prediction of 0.5, so our temptation is to add the prediction this tree makes to the original, and obtain a great prediction. But we must resist this temptation! Recall that in machine learning, we like to give very small steps in order to prevent overfitting as much as we can. Therefore, what we'll do is multiply the predictions made by this tree by the learning rate (which we set as 0.8), add that to the original 0.5, and that's our new prediction. Then we calculate the new residuals by subtracting the prediction from the previous residual. The results of this are in Table 10.5.

To summarize, these are the columns in Table 10.5.

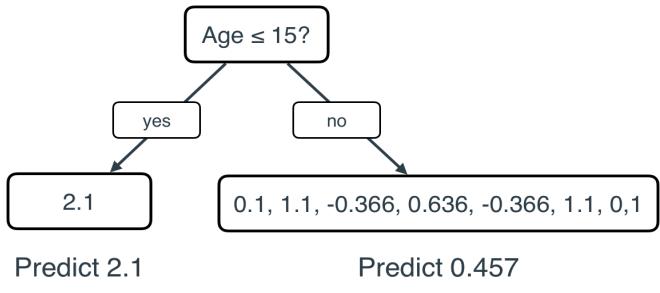
- Age: The age of the user.
- Engagement: The level of engagement of the user, from 1 to 7.
- First tree prediction: The prediction of the first tree.
- First residual: The difference between the label and the first prediction.
- Second tree prediction: The prediction of the second tree (meant to predict the first residual)
- Prediction times learning rate: The prediction of the second tree times the learning rate, which in this case is 0.7.
- Total prediction: The sum of predictions 1 and 2. That is the total prediction the model is making so far.
- Second residual: The difference between the label and the total prediction so far. This is the residual we want the next tree to predict.

**Table 10.5.** The table with age and engagement of the users, the results of the first and second learners, and their corresponding residuals.

Age (feature)	Engagement (label)	Prediction 1 (First tree)	First residual	Prediction 2 (Second tree)	Prediction 2 times learning rate	Total prediction (pred 1 + pred 2)	Second residual
10	7	0.5	6.5	5.5	4.4	4.9	2.1
20	5	0.5	4.5	5.5	4.4	4.9	0.1
30	6	0.5	5.5	5.5	4.4	4.9	1.1
40	0	0.5	-0.5	-0.167	-0.134	0.366	-0.366
50	1	0.5	0.5	-0.167	-0.134	0.366	0.636
60	0	0.5	-0.5	-0.167	-0.134	0.366	-0.366
70	4	0.5	3.5	3	2.4	2.9	1.1
80	3	0.5	2.5	3	2.4	2.9	0.1

Now, our goal is to build the third tree. This tree is built to predict the second residual in table 10.5. I encourage you to work it out given the specifications, and see if you got the same as I did. Recall that, given the steps and hyperparameters, you must do this to build the tree:

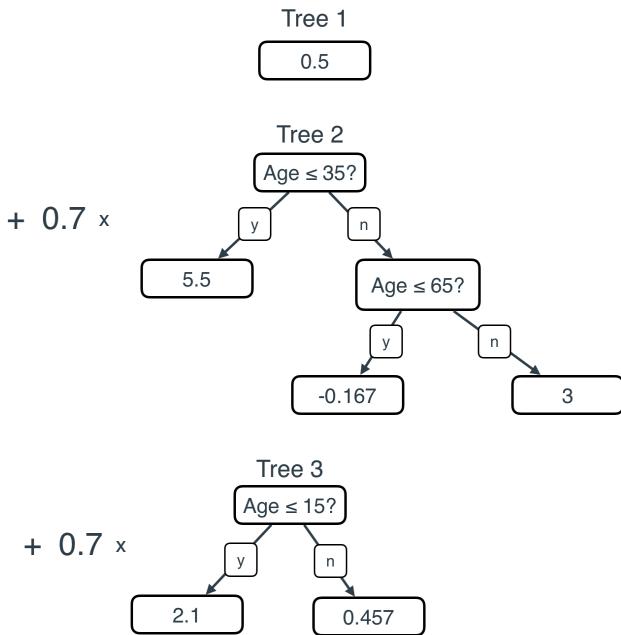
1. Build a tree trying to fit the “Second residual” column, where at each step you split the nodes in order to maximize the similarity gain.
2. Since our minimal elements per leaf is 1, we don’t need to worry. But if we had a higher number, we must make sure that no leaf is too small.
3. Stop building the tree when you reach depth 2.
4. Prune every branch of the three where the similarity gain is not higher than 3.



**Figure 10.25.** Our resulting tree after we pruned the left branch, since it didn't give us a high gain in similarity score. In each of the leaves, we have the elements that belong to it. Below the leaves, we show the prediction made at that leaf. Notice that this prediction is the average of the values in the leaf.

The tree I got is in Figure 10.25. Notice that the tree is small, because the left leaf only has one element (so it doesn't get split), and for the right leaf, no splitting gave us a similarity gain higher than 3.

Now, as a final step, we multiply the predictions given by this second tree by the learning rate (0.7), and that is our final prediction. A graphical way to see the resulting model is illustrated in Figure 10.26, where the first tree is simply one prediction, the second and third tree are larger, but their prediction is multiplied by the learning rate 0.7.



**Figure 10.26.** The three learners (Tree 1, Tree 2, and Tree 3). The first one simply predicts 0.5 for every data

point. The second and the third are more detailed, but their predictions get multiplied by the learning rate, in order to avoid overfitting.

Now, in order to see how the tree did, we can add the last prediction to the table. The final prediction is then calculated as the prediction given by the first tree, plus the learning rate times the prediction given by the second tree, plus the learning rate times the prediction given by the third tree. The formula is

$$\text{Final prediction} = \text{Prediction 1} + 0.7 * \text{Prediction 2} + 0.7 * \text{Prediction 3}.$$

The resulting values are in Table 10.6.

**Table 10.6. The predictions of our three learners, together with the final prediction.**

Age (feature)	Engagement (label)	Prediction 1 (First tree)	Prediction 2 (Second tree)	Prediction 3 (Third tree)	Final prediction
10	7	0.5	5.5	2.1	5.82
20	5	0.5	5.5	0.457	4.67
30	6	0.5	5.5	0.457	4.67
40	0	0.5	-0.167	0.457	0.7
50	1	0.5	-0.167	0.457	0.7
60	0	0.5	-0.167	0.457	0.7
70	4	0.5	3	0.457	2.9
80	3	0.5	3	0.457	2.9

Notice that the final prediction is much closer to the label than the first prediction and the second prediction. We can still do better, but we'll stop right here. I encourage you to continue adding trees to see how close you get to the predictions, or even to code this algorithm and try it with this small dataset.

#### **GRADIENT BOOSTING AND XGBOOST FOR CLASSIFICATION**

If I had an infinite number of pages, I would love to go through every single algorithm in detail, but unfortunately that is not the case. There are many algorithms that will not make it into this book, and two examples are Gradient Boost and XGBoost for classification. Both of them are

similar to the one I just illustrated on regression, but with some small differences. I highly encourage you to look them up, and understand them yourself.

However, I can show you how to code them in Python. They're both in the repo at [www.github.com/luisquiserrano/manning](https://www.github.com/luisquiserrano/manning). Gradient boost is in sklearn, but XGBoost is not, although if you install the xgboost package, the syntax is very similar to sklearn. Here I will show you how to code gradient boost and use it to model the spam email data we've been working on in this chapter.

As before, we import the package, define and fit the model as follows.

```
from sklearn.ensemble import GradientBoostingClassifier
gradient_boosting_model = GradientBoostingClassifier(random_state=0, n_estimators=5)
gradient_boosting_model.fit(new_X, new_y)
```

Then we can find the accuracy.

```
gradient_boosting_model.score(new_X, new_y)
0.9285714285714286
```

That's not a bad accuracy. As before, we can also plot the boundary (Figure 10.27) and notice that it classifies the data well, but it tries to avoid overfitting.

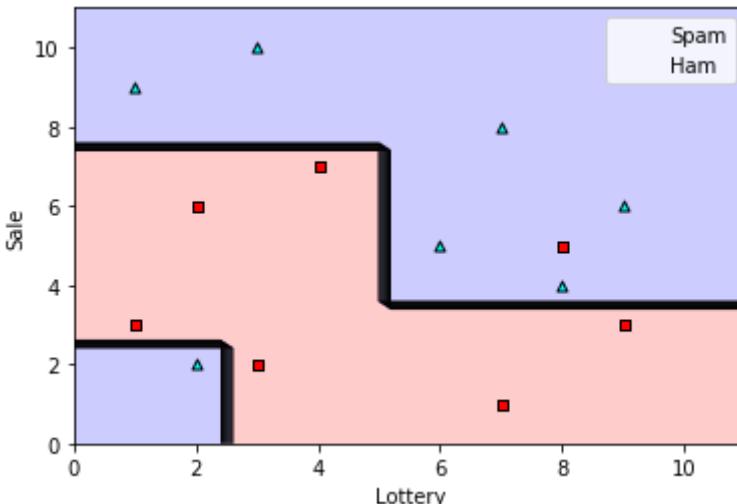


Figure 10.27. Plot of the gradient boosted trees classifier on our spam dataset.

## 10.6 Applications of ensemble methods

Ensemble methods are some of the most useful machine learning techniques used nowadays as they exhibit great levels of performance with relatively low cost. One of the places where

ensemble methods are used the 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 very strong combination of weak learners in an ensemble to win.

## 10.7 Summary

- Ensemble methods are ways we use to combine weak learners into a strong one. There are two major types of ensemble methods: Bagging and boosting.
- Ensemble methods can be used for regression and for classification.
- Bagging, or bootstrap aggregating, consists of building successive learners on random subsets of our data, and then building a strong classifier 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 then building a strong classifier based on a weighted majority vote of the learners.
- AdaBoost and XGBoost are two advanced boosting algorithms that produce great results with real datasets.
- Applications of ensemble methods range very widely, from recommendation algorithms to applications in medicine and biology.