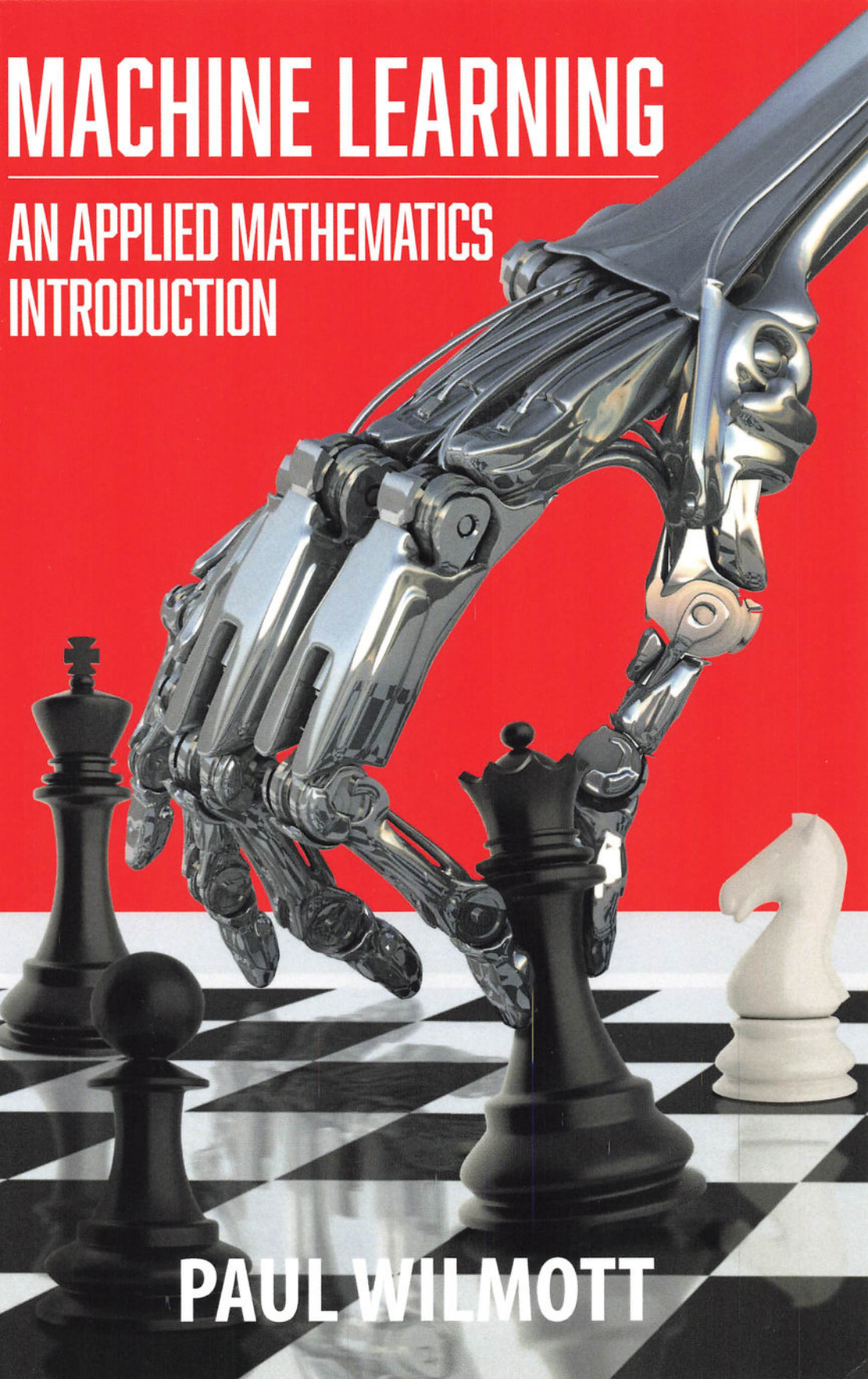
MACHINE LEARNING

AN APPLIED MATHEMATICS INTRODUCTION



PAUL WILMOTT

MACHINE LEARNING

An Applied Mathematics Introduction

First published 2019 by Panda Ohana Publishing

pandaohana.com

admin@pandaohana.com

First printing 2019

ISBN 978-1-9160816-0-4

Library of Congress Control Number: 2019938983

MACHINE LEARNING

An Applied Mathematics Introduction

First Edition

Paul Wilmott
www.wilmott.com

Panda Ohana Publishing

Other books by Paul Wilmott

Option Pricing: Mathematical Models and Computation. (With J.N.Dewynne and S.D.Howison.) Oxford Financial Press (1993)

Mathematics of Financial Derivatives: A Student Introduction. (With J.N.Dewynne and S.D.Howison.) Cambridge University Press (1995)

Mathematical Models in Finance. (Ed. with S.D.Howison and F.P.Kelly.) Chapman and Hall (1995)

Derivatives: The Theory and Practice of Financial Engineering. John Wiley and Sons (1998)

Paul Wilmott On Quantitative Finance. John Wiley and Sons (2000)

Paul Wilmott Introduces Quantitative Finance. John Wiley and Sons (2001)

New Directions in Mathematical Finance. (Ed. with H.Rasmussen.) John Wiley and Sons (2001)

Best of Wilmott 1. (Ed.) John Wiley and Sons (2004)

Exotic Option Pricing and Advanced Lévy Models. (Ed. with W.Schoutens and A. Kyprianou) John Wiley and Sons (2005)

Best of Wilmott 2. (Ed.) John Wiley and Sons (2005)

Frequently Asked Questions in Quantitative Finance. John Wiley and Sons (2006)

Paul Wilmott On Quantitative Finance, second edition. John Wiley and Sons (2006)

Paul Wilmott Introduces Quantitative Finance, second edition. John Wiley and Sons (2007)

Frequently Asked Questions in Quantitative Finance, second edition. John Wiley and Sons (2009)

The Money Formula: Dodgy Finance, Pseudo Science, and How Mathematicians Took Over the Markets. (With David Orrell). John Wiley and Sons (2017)

To JRO, from your fearless student

Contents

Prologue

This book aims to teach you the most important mathematical foundations of as many machine-learning techniques as possible. I also leave out what I judge to be the unnecessary boring bits. It is also meant to be orthogonal to most other books on and around this subject.

If you've already bought this book then you may want to skip the next few paragraphs. In them I explain who the book is for. If you have bought the book and it turns out you are the wrong audience then... Oops.

There are several different types of reader of machine-learning textbooks. And different quantities of those.

Judging by what's available on Amazon I reckon that there are an awful lot of people out there who just want to see the relevant lines of code. Often in Python or R. There are plenty of books that will show you a surprisingly compact coded-up representation of the solution of a problem. I'm not going to worry about these people. You are well catered for already.

Then there are quite a lot of books that describe the *uses* of machine-learning techniques. But this time not much code, and very elementary mathematics. These are great for getting an overview of the subject. But they lack in detail. You can spot them by the dodgy typeface they use for their mathematics. And that they are self published, although there is absolutely nothing wrong with that at all.

And then there are the specialist books. Each devoted to a narrow field, but extremely deep. I can't fault their typefaces. But they are a little bit scary. And usually very, very expensive.

No, I'm not trying to compete with any of the above.

I am aiming for a small audience, not well catered for. An audience, probably shrinking as I type. An audience of applied mathematicians. People who like *doing* mathematics, solving problems, perhaps old hands at classical techniques and keen to know more about this new data-driven revolution that they are on the edges of. You know who you are. Yes, you at the back, don't try sneaking out, sit down here at the front and let's learn this subject together...

A Word From The Machine

But first, I think we need to hear what The Machine has to say about this book.[1] (It is very insightful. I particularly liked the endorsement at the end.)

These volumes are meant to be in multiple sections. You can read them in this book at each of the four different places, from beginning to end. The main point you want to make here is that both the book and the text on as many of the techniques that you will use will be well-understood. But what will be done to make them all accessible, and that they won't be too hard to understand? In fact, each of the techniques will be quite easy to understand, and the material will be very well organized so that you can do what you want in the most short amount of time (i.e., in less than 40 pages). A few basic examples of different types of machine-learning can be found in the book, but I'll focus on the main points of the book. If you're interested in reading more about this subject, then see my book.

Acknowledgements

I would like to thank my co-author on *The Money Formula* (available in all good book shops), David Orrell, for his fresh pair of eyes in typo spotting. And Liam Larkin, the designer of my magazine ("The world's most expensive magazine," according to *Esquire*), for the cover.

I am especially indebted to Thijs van den Berg who got me into this subject. He also made many suggestions for additions and corrections to this book, 82.1% of which I implemented. He and I give training courses on machine learning, and so I'd also like to thank the many delegates on those courses for all their comments and questions that helped me to explain things as well as possible.

Finally, I must mention my wife, my children, Her Majesty's Revenue and Customs, the IRS, "Game of Thrones," Sky News and Brexit, without whom this book would have been finished in half the time.

[1] Text generated by `https://colab.research.google.com/github/ilopezfr/gpt-2/blob/master/gpt-2-playground_.ipynb`

About the Author

Professionally speaking...

Paul Wilmott studied mathematics at St Catherine's College, Oxford, where he also received his D.Phil. He is the author of *Paul Wilmott Introduces Quantitative Finance* (Wiley 2007), *Paul Wilmott On Quantitative Finance* (Wiley 2006), *Frequently Asked Questions in Quantitative Finance* (Wiley 2009), *The Money Formula: Dodgy Finance, Pseudo Science, and How Mathematicians Took Over the Markets* (with David Orrell) (Wiley 2017) and other financial textbooks. He has written over 100 research articles on finance and mathematics. Paul Wilmott was a founding partner of the volatility arbitrage hedge fund Caissa Capital which managed $170 million. His responsibilities included forecasting, derivatives pricing, and risk management.

Paul is the proprietor of `www.wilmott.com`, the popular quantitative finance community website, and the quant magazine *Wilmott*. He is the creator of the Certificate in Quantitative Finance, `cqf.com`, and the President of the CQF Institute, `cqfinstitute.org`.

On the other hand...

Paul was a professional juggler with the Dab Hands troupe, and has been an undercover investigator for Channel 4. He also has three half blues from Oxford University for Ballroom Dancing. At the age of 41 he finally won a sandcastle-building competition. He makes his own cheese. Its flavour has been described as "challenging."

Paul was the first man in the UK to get an online divorce. He was an expert on a TV show, tasked with forecasting a royal baby name and the winner of the Eurovision Song Contest among other things. He got everything wrong.

He played bridge for his school's D team. There was no E team.

And he plays the ukulele. Obviously.

Chapter 1

Introduction

Hi, my name is Paul Wilmott. (This chapter is called Introduction, after all.) I am an applied mathematician. For many years I worked in academia, researching this and that, supervising students, writing papers for learned journals. My early research was in the field of applicable/industrial mathematics. That meant speaking to people from industry and turning their problems, often about some physical process, into mathematical models. These mathematical models could then be solved to give some insight into the problems they had posed. This had many advantages over, say, building experiments, not least that we were much cheaper. After a few years of this one builds up a skill set of techniques and principles that help with modelling just about anything. Sometimes you'd have physical laws, such as conservation of mass, on which to base your models, although often not. If you have such solid physical laws then this should lead to a *quantitatively* good model. Otherwise you might end up with a model that is only *qualitatively* good. The latter could be thought of as a toy model. (Ok, I guess sometimes the models were even bad, but we tended not to publish those. Usually.)

I'm telling you all this because it very much influences the way that I see this relatively young field of machine learning. I bring to it some worries and some scepticism. I will be sprinkling my concerns throughout this book, although I do hope not to spoil the party too much. I'll come back to this a bit later in this chapter. And I am going to be contrasting classical mathematical modelling and machine learning. For two fields that are trying to achieve the same goal it's surprising how different they are.

I also bring my philosophy as an applied mathematician who likes to solve real-world problems. This means writing down some mathematical model as quickly as possible without having to worry too much about proving everything at every step of the way. Please note that just because something is not proved does not mean that it is wrong. It just means getting to the meat as quickly as possible without having to eat the vegetables. (Having

said all that I am sure that some readers will take it upon themselves to find as many errors in this book as they can. If you do find any then email me at `paul@wilmott.com` in the first instance before reviewing me on Amazon. I'm very approachable.)

This book does not include any code whatsoever. This is deliberate for two reasons. Mainly it's because I am a useless programmer. But also the upside is that I do have to teach the methods with enough detail that you can implement them yourself. I cannot take any shortcuts such as giving a no-math description of a technique followed by saying "Now write these two lines of code, press enter, and look at the pretty pictures." There are many programming packages you can use to implement the methods that I describe and you don't need to know what they are doing to use them. That's fine. It's like my feeble understanding of a carburettor doesn't stop me enjoying my Jensen Interceptor Convertible. But equally if I did understand the carburettor it would probably save me quite a bit of time and expense.

Finally, there are plenty of examples in this book, almost always using real data. But this is not a research-level tome, and so the examples are for illustration only. If it were a book describing the cutting edge of research then there would be a heck of a lot more cleaning and checking of the data and a lot more testing and validation of the models. But there'd also be a lot less explanation of the basics, and perhaps less mentioning of where things go wrong.

But now let's actually talk about the topic at hand.

1.1 The Topic At Hand

A definition from Wikipedia:

Machine learning is a field of computer science that uses statistical techniques to give computer systems the ability to 'learn' (e.g., progressively improve performance on a specific task) with data, without being explicitly programmed.

Where to start... with computers? Or with statistics? Or data?

This is not the place for a detailed history of machine learning. I could never do as well as google. (I also have this personal prejudice that any mathematician who knows about the history of their subject isn't a proper mathematician.) If you want some pithy paragraphs for a dinner-party conversation (before changing the subject to something more spicey, like religion or politics) then...

- The field of probability and statistics goes back to 1654, and a question of gambling debated by the French mathematicians Blaise Pascal and

Pierre de Fermat. Their problem was to figure out how to allocate a bet in a game of dice (called Points) if the game were to be suspended part way through, one side of the game doing better than the other. Thus was invented/discovered the concept of mathematical expectations.

- Who invented the first calculator is disputed. It might have been Pascal again or it might have been Wilhelm Schickard around 1623. Either way it was little more than an adding machine. However the first *programmable* computer was invented, but never built, by Charles Babbage, often called the father of the computer, in 1837. His computer was intended to use cards with holes punched in them for the programming. Such a system of programming existed into surprisingly recent times. To readers of a certain age this will be a familiar, if painful, memory.

- However, before these came data collection. And machine learning uses data. Lots and lots of it. Historically data was often in the form of surveys. The famous Domesday Book of the late 11th Century was a survey, ordered by William the Conqueror, of England and Wales. One of its purposes was to aid in the collection of taxes. *Plus ça change, plus c'est la même chose.*

Take data and statistical techniques, throw them at a computer and what you have is machine learning.

1.2 Learning Is Key

Except that there's more to it than that. And there's a clue in the title of the subject, in the word "learning." In traditional modelling you would sit down with a piece of paper, a pencil and a single malt and... tell the algorithm everything you know about, say, chess. You'd write down some code like "IF White Queen something something Black Bishop THEN something something." (I haven't played chess for forty years so apologies for a lack of precision here.) And you'd have many, many lines of IF, AND and OR, etc. code telling the programme what to do in complex situations, with nuances in positions. Or ... you'd write down some differential equation that captures how you believe that inflation will respond to a change in interest rates. As inflation rises so a central bank responds by increasing interest rates. Increasing interest rates causes...

You might eyeball the data a bit, maybe do a small amount of basic statistics, but mainly you'd just be using your brain.

Whatever the problem you would build the model yourself.

In machine learning your role in modelling is much more limited. You will decide on a framework, whether it be a neural network or a support vector machine, for example, and then the data does the rest. (It won't be long now before the machine even does that for you.) The algorithm *learns*.

1.3 A Little Bit Of History

As I said, I'm not going to give you much history in this book. That would be pointless, history is always being added to and being reassessed, and this field is changing very rapidly. But I will tell you the two bits of machine-learning history that I personally find interesting.

First let me introduce Donald Michie. Donald Michie had worked on cyphers and code cracking with his colleague Alan Turing at Bletchley Park during the Second World War. After the war he gained a doctorate and in the early 1960s, as a professor in Edinburgh, turned his attention to the problem of training — a word you will be hearing quite a lot of — a computer to play the game of Noughts and Crosses, a.k.a. Tic Tac Toe. Well, not so much a computer as an array of matchboxes, *sans* matches. Three hundred and four matchboxes, laid out to represent the stages of a game of Noughts and Crosses. I won't explain the rules of O&Xs, just like further down I won't be explaining the rules of Go, but each stage is represented by a grid on which one player has written an X alternating with the other player writing an O with each player's goal being to get three of their symbols in a row. (Damn, I did explain the rules. But really do not expect this for Go. Mainly because I don't fully understand the game myself.)

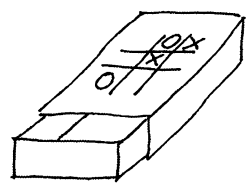


Figure 1.1: A matchbox representing a state in Noughts and Crosses.

In Figure 1.1 we see a sketch of what one of Professor Michie's matchboxes might have looked like. In this it is X's turn to play. There is clearly a winning move here. (Actually two of them.) I know that, and you know that, but if the computer were playing would it also know? Traditionally one might programme a computer with all the possible states of the game and the best next move in each case. That's doable for O&Xs but most definitely is not with any non-trivial game. What Professor Michie did instead was to

fill each of his 304 matchboxes with coloured beads, each colour representing one of the empty cells in the grid. (He took advantage of symmetries or there would have been even more than 304 matchboxes.) When it's our move we take the matchbox representing the current state, shake it, and remove a random bead. It's colour tells us where to place our X. This is followed by our opponent's play. The game continues, and we move from state to state and matchbox to matchbox. And if we win then each matchbox used in that game gets rewarded with extra beads of its chosen colour, and if we lose then a bead of that colour gets taken away. Eventually the matchboxes fill up with beads representing the most successful action at each state. The machine, and Michie called it MENACE, for Machine Educable Noughts And Crosses Engine, has learned to play Noughts and Crosses. There's some jargon in the above: Training; State; Action; Reward; Learn. We'll be hearing more of these later. And we shall return to Donald's matchboxes in the chapter on reinforcement learning.

The next thing that I found interesting was the story of how Google DeepMind programmed a computer to learn to play Go. And it promptly went on to beat Lee Sedol, a 9-dan professional Go player. (No, I didn't know you could become a 9th dan by sitting on your butt either. There's hope for us all.) AlphaGo used a variety of techniques, among them neural networks. Again it learned to play without explicit programming of optimal play. Curiously not all of that play was against human opponents... it also played against itself. To show how difficult Go is, and consequently how difficult was the task for Google DeepMind, if you used matchboxes to represent states *à la* Donald Michie then you'd need a lot more than 304 of them, you'd probably have to fill the known universe with matchboxes. I would strongly recommend watching the creatively titled movie "The AlphaGo Movie" to see a better description than the above and also to watch the faces of everyone involved. Lee Sedol started moderately confident, became somewhat gutted, and finally resigned (literally and psychologically). Most of the programmers were whooping for joy as the games progressed. Although there was one pensive-looking one who looked like he was thinking "Oh... my... God... what... have... we... done?"

At times the computer made some strange moves, moves that no professional Go player would have considered. In the movie there was discussion about whether this was a programming error, lack of sufficient training, or a stroke of computer genius. It was always the last of these. It has been said that the game of Go has advanced because of what humans have learned from the machine. (Did a chill just go down your spine? It did mine.)

In this book I am going to guide you through some of the most popular machine-learning techniques, to the level at which you can start trying them out for yourself, and there'll be an overview of some of these for the rest of this chapter.

The book is not meant to be rigorous, it is meant to get you interested and sitting at your PC with a spreadsheet (or whatever) open. It's the book I wish I'd read when starting out in this field.

I'll also give some warnings. The warnings are inspired by how easy it is to get carried away by machine learning, because it's so much gosh-darned fun! Overconfidence is not good when you are dealing with a topic that judging by current evidence seems to be taking over the planet.

1.4 Key Methodologies Covered In This Book

There are three main categories of machine learning. Two of these categories concern how much help we give the machine. And the third is about teaching a machine to do things.

Principal categories

- **Supervised Learning:** In supervised learning you are given both inputs and outputs. The input would be a vector, i.e. typically more than one variable. For example, we start with pictures of dogs that have been digitized and turned into vectors representing the colours in the pixels. These pictures have each been classified as Hound, Toy, Terrier, etc. We train our algorithm on these pictures and then we would want it to correctly classify any new picture we give it. We can also use supervised learning for regression, predicting numerical values for outputs for any new data points we input (e.g. input temperature and time of day, and predict electricity usage). Supervised learning includes many different algorithms, some mentioned below.

- **Unsupervised Learning:** Unsupervised learning is when the data is not labelled. This means that we only have inputs, not outputs. In a sense the algorithm is on its own. The algorithm finds relationships or patterns for you. We show the computer digitized pictures of dogs, and the computer groups or clusters them together according to whatever features *it* deems most important. Perhaps it will come up with Hound, Toy, Terrier,... or perhaps black, brown, white,... or something completely different that humans don't even notice. Unsupervised learning might at first seem strange, but you'll understand it if I said to you "Here are the contents of my desk drawer. Please sort them out." Unsupervised learning includes many different algorithms, some mentioned below.

- **Reinforcement Learning:** In reinforcement learning an algorithm learns to *do* something by being rewarded for successful behaviour and/or being punished for unsuccessful behaviour. (And we're back with dogs again!) The above-mentioned MENACE was a simple example of this, the rewards and punishments being the addition or subtraction of beads to or from the matchboxes. Computers learning to play board games or video games from the 1980s seem to get a lot of publicity. That would usually involve reinforcement learning.

Principal techniques

I'm going to describe many of the most important techniques within the above principal categories. And the order I've chosen to explain them is governed mainly by their mathematical complexity. Most of them can be implemented in just a few lines of Python code, but my job is not to give you those few lines but to explain what that code is doing. Below are the supervised and unsupervised techniques we'll be seeing. Reinforcement learning is treated separately, right at the end of the book.

I personally don't necessarily see some of these techniques as machine learning, they share too many similarities to classical statistical methods. But I've included them here anyway because other people think they are machine learning and they do share the fundamental characteristic of using lots and lots of data.

- K **Nearest Neighbours:** K nearest neighbours is a supervised-learning technique in which we measure distances between any new data point (in any number of dimensions) and the nearest K of our already-classified data and thus conclude to which class our new data point belongs. A silly example: We are at a party, and out of the five people standing closest to us three are wearing glasses, so we are probably glass wearers too. The method can also be used for regression.
- K **Means Clustering:** K means clustering is an unsupervised-learning technique. We have lots of data in vector form (representing many dimensions of information about each point), and we have K, a number, points or centroids in the space. Each data point is associated with its nearest centroid, and that defines to which category each data point belongs. Then we find the optimal position for these centroids that gives us the best division into the K classes. And finally we play around with the number K to see what effect it has on the division of the data into categories.
- **Naïve Bayes Classifier:** Naïve Bayes classifier is a supervised-learning technique that uses Bayes Theorem to calculate the probability of new

data points being in different classes. The naïve bit refers to the assumptions made, which are rarely true in practice but that doesn't usually seem to matter.

- **Regression Methods:** Regression methods are supervised-learning techniques that try to explain a numerical dependent variable in terms of independent variables. You'll probably know of linear regression at least. But you can go much further with more complicated types of regression.

- **Support Vector Machines:** A support vector machine is another supervised-learning technique, one that divides data into classes according to which side of a hyperplane in feature space each data point lies.

- **Self-organizing Maps:** A self-organizing map is an unsupervised-learning technique that turns high-dimensional data into nice, typically two-dimensional, pictures for visualizing relationships between data points. Imagine a chess board and into the squares you put data points with similar characteristics. You don't specify those characteristics, they are found as part of the algorithm.

- **Decision Trees:** This is a supervised-learning technique. A decision tree is just a flowchart. "How many legs does it have? Two, four, more than four?" Four. Next, "Does it have a tail? Yes or no." Yes. And so on. Like a game of 20 Questions. But can the machine learn and improve on how humans might classify? Is there a best order in which to ask the questions so that the classifying is done quickly and accurately? It can also be used for regression.

- **Neural Networks:** A neural network (sometimes with artificial in front) is a type of machine learning that is meant to mimic what happens in the brain. An input vector is transformed, typically by multiplying by a matrix. So far, so linear. Then the result is put into some non-linear activation function. That's one layer. But that result then usually goes into another layer, where it undergoes similar transformations, and so on. In this form it's really just a non-linear function of the original data. But it can get more complicated by having the data not just going in one direction but also looping back on itself. The parameters in the neural network are found by training. It can be used for either supervised or unsupervised learning.

1.5 Classical Mathematical Modelling

I want to explain, briefly, how machine learning is different from classical mathematical modelling. And I must stress that I'm not talking about statistical modelling or econometrics which share some similarities with machine learning. This is to some extent a personal view of my experience in applied mathematical modelling.

There's a well-recognised, and well-trodden, path that (we) mathematical modellers go along when faced with a new problem. It goes something like this...

First decide on your variables

Variables come in two main types, independent and dependent. Independent variables represent the domain in which your problem is specified. A common group of independent variables would be spatial and time coordinates, x, y, z, and t.

Contrast this with the dependent variables, these are the things you are solving for. You might have a problem in heat flow in which case your dependent variable would probably be temperature. As another example consider the oscillation of a 'cello string (guitar string for you in the cheap seats). The dependent variable would be distance by which the string is perturbed from rest. (So here you'd have a dependent variable that is a distance from rest, but also an independent variable that's a distance, distance along the string, as well as time.)

You really want the minimum of variables to specify the system and its behaviour. Einstein said something along the lines of "Make things as simple as possible, and no simpler." That's probably Rule Number 1 of modelling. Rule Number 2 would be "If you are going to make things more complicated then do so one new feature at a time." Already that's a big difference to machine learning in which one tends to throw in everything at the start, including the kitchen sink.

The goal at this stage in classical mathematical modelling is to get to the first minimally interesting non-trivial problem.

Figure out what are the main drivers in your system

What drives your system? Are there forces pushing and pulling something? Are molecules bouncing off each other at random? Do buyers and sellers move the price of milk up and down?

If you are lucky you'll have some decent principles to base your model

on, such as conservation of something. The 'cello string moves according to Newton's second law, the force caused by the bend in the string makes the string move.

Or maybe the physics is too complicated but the statistics is robust, such as in a coin toss.

(There's also the category of simply feeling right, such as the inverse square law of gravity.)

But maybe you have to fall back on some common sense or observation. To model the dynamical system of lions and gazelles (the former like to eat the latter) then you would say that the more lions there are the more they breed, ditto for gazelles. And the more gazelles there are the more food for lions, which is great for the lions but not so much for the gazelles. From this it is possible to write down a nice differential-equation model. The 'cello-string model will give accurate behaviour for the string, but while the lion-gazelle model is usefully explanatory of certain effects it will not necessarily be quantitatively accurate. The latter is a toy model.

Your model will probably need you to give it values for some parameters. Again if you are lucky you might be able to get these from experiment; the acceleration due to gravity, or the viscosity of a fluid. And perhaps those parameters will stay constant. But maybe the parameters will be difficult to measure, unstable, or perhaps not even exist, the volatility of a stock price for example.

These are issues that the mathematical modeller needs to address and will make the difference between a model with reliable outputs and one that is merely useful for explanation of phenomena.

Decide on your type of mathematics

There are many branches of mathematics, some are more easily employed than others when it comes to practical problems.

Discrete mathematics, or continuous mathematics? Will you be using calculus, perhaps ordinary or partial differential equations? Or perhaps the model is going to use probabilistic concepts. If you are trying to figure out how to win at Blackjack then clearly you'll be working with discrete values for the cards and probability will play a major role. As will optimization. Change to poker and game theory becomes important.

Mathematicians do suffer from the classic "If all you have is a hammer every problem becomes a nail." This can affect the evolution of problem solutions, as the problem becomes known as, say, a problem in subject X, when subject Y might also be useful.

And the answer is...

Setting out a problem and a model is nice. But you'll almost always want to find its solution. If lucky, again, you'll get a formula using a pencil and paper. More often than not, especially if it's a real-world, often messy, problem then you'll have to do some number crunching. There are many numerical methods, many for each branch of mathematical modelling. Numerical solution is different from modelling, don't confuse the two. Numerical analysis is a subject in its own right.

1.6 Machine Learning Is Different

Machine learning could not be more different if it tried. Almost nothing in the above carries over to machine learning.

One big difference is the role of data. In classical mathematical modelling you might not have any data. Say you have a problem in fluid mechanics then you build your model on conservation of mass and momentum (I say "your model," but of course it's due to Euler, Navier and Stokes). You'd come up with a model while sitting in your office with nothing more than a pencil and some paper. And the whisky. There would be but a few parameters (fluid density and viscosity) and you are done. You then have to solve for your particular geometry. If this were a machine-learning problem then you show the machine a great deal of data, perhaps in the form of movies of flow past various shapes, and let it do its thing. All being well the machine would learn and somewhere deep inside it would have indirectly hit on something like the Navier–Stokes equation. You certainly wouldn't have specified any drivers, conservation laws, etc.

Sherlock Holmes would no doubt embrace machine learning, "It is a capital mistake to theorise before one has data."

In classical modelling, parameters often have some meaning or nice interpretation. They certainly do if they represent something physical. But how can you interpret parameters in the middle hidden layer of a neural network? They are just numbers that happen to give you the best fit of your model to data.

And the role of type of mathematics would become the choice of machine-learning scheme. Is the problem best approached via supervised or unsupervised learning? Would it be a self-organizing map or K means clustering that gave the most reliable results?

Very often the building blocks of a machine-learning algorithm are very simple. A neural network might involve matrix multiplications and a few simple non-linear functions. Simple models can lead to complex behaviour. But is it the right simple model and a quantitatively useful behaviour?

1.7 Simplicity Leading To Complexity

Let's remind ourselves about a few simple mathematical concepts that lead to complex structured behaviour.

Cellular automata

A cellular automaton is typically made up of a grid of cells, each cell being in one of several states. The cells change state according to some rules. Although dating back to the 1940s with Stanislaw Ulam and John von Neumann at the Los Alamos National Laboratory, the most famous cellular automaton is probably what is known as The Game of Life designed by John Conway in the 1970s. In this each cell is either alive or dead, and is meant to represent a population of living creatures. See Figure 1.2 for an example.

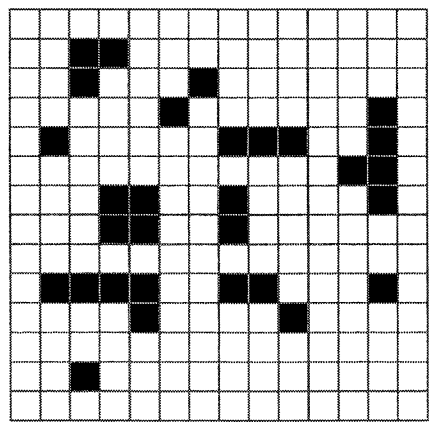


Figure 1.2: A snapshot in The Game of Life.

The rules are:

- Any live cell with fewer than two live neighbours dies, as if caused by underpopulation
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overpopulation
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction

A nice video of The Game of Life on a torous can be found here:
`https://www.youtube.com/watch?v=lxIeaotWIks`.
And a three-dimensional version here:
`https://www.youtube.com/watch?v=iiEQg-SHY1g`.

I worked on a cellular automaton model in the early 1990s, a model representing the separation one finds in granular material when they are of different sizes. This model was created because of the lack of progress we had made in trying to develop a more classical continuum model.

Fractals

Technically to understand fractals you need to know about Hausdorff Dimension versus Topological Dimension. But we can get part way there by considering the length of the coastline of Great Britain. It's fairly obvious that the shorter the ruler that you use for measuring then the longer the measured coastline. The shorter the ruler the more nooks and crannies you will get into. This is completely different for a classical two-dimensional shape, such as a circle, for which the length of the boundary converges as the ruler shrinks. Lewis Fry Richardson measured various coastlines and found a power law approximation to the measured length. From this follows the idea of the fractal dimension of a coastline. In the case of the west coast of Britain he found it to be 1.25, so somewhere between a line and an area.

Simple algorithms can be used to generate complex fractal shapes. Here is an algorithm for generating the Barnsley Fern, Figure 1.3.

$$f_1(x,y) = \begin{bmatrix} 0.00 & 0.00 \\ 0.00 & 0.16 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$f_2(x,y) = \begin{bmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 1.60 \end{bmatrix}$$

$$f_3(x,y) = \begin{bmatrix} 0.20 & -0.26 \\ 0.23 & 0.22 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 1.60 \end{bmatrix}$$

$$f_4(x,y) = \begin{bmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 0.44 \end{bmatrix}.$$

To generate a fern simply choose a starting x and y, say $(0,0)$, and place a dot at that point. Now pick one of the above four transformations at random and input the x and y. You pick the first transformation with probability 1%, the second with probability 85%, and the last two each with 7%. This gives you a new position for a dot. Now take the new x and y and one of the four transformations... repeat until you have filled in the shape sufficiently.

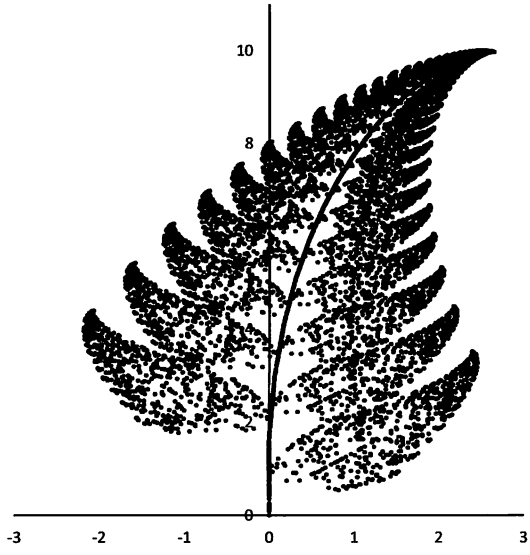


Figure 1.3: Computer-generated fern.

Chaos

And then there's everybody's favourite, chaos. A simple nonlinearity can lead to extremely complex and unpredictable behaviour despite perfect determinism. Take for example a basic population model (known as the Logistic Map)

$$x_{n+1} = \lambda x_n(1 - x_n),$$

where x_n is the number in the population as a function of the generation n. This can be thought of as a model for population dynamics because it has two key features: For small populations the size of the next generation is proportional to the current population, with a growth coefficient of λ; If the population gets too big then the growth rate shrinks because of competition for resources.

Depending on the size of λ you can get x_n converging to zero, converging to some other value, oscillating between two or more values, or chaos. A primary feature of chaos is sensitive dependence on initial conditions, the Butterfly Effect. The departure of two paths with initially close initial conditions is governed by the Lyapunov Exponent, and is the reason why chaotic systems are unpredictable too far into the future.

See Figure 1.4 for an example plot of x_n with $\lambda = 3.7653$.

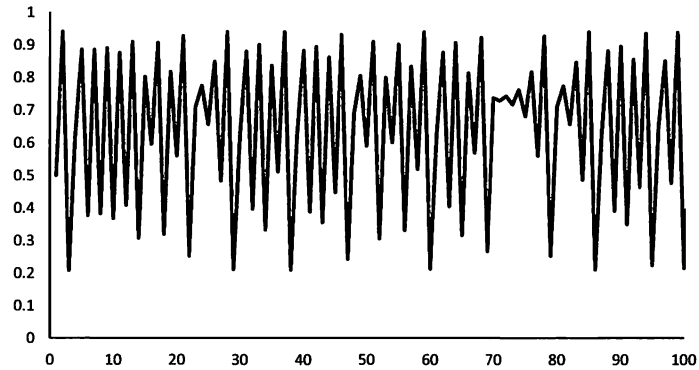


Figure 1.4: Chaos.

And your point is, Paul?

Those last few pages were rather tangential to our topic. But let me reassure you that you have not been charged for them.

I mention all of the above as both an encouragement and as a warning to anyone embarking on machine learning. On a positive note we see complex behaviour arising out of the simplest of models. And some of the machine learning you will be seeing can be very simple indeed. The building blocks of basic neural networks are just matrix multiplication and a fairly bland activation function.

On the other hand once we have represented something quite complicated, such as millions of data points, with something simple, how do we ever know that it's the *right* simple model?

I recall a great deal of excitement in my mathematics department in the late 1980s because a group had discovered they could use dynamical systems (chaos) to model the foreign exchange markets. As I recall the excitement soon faded once they learned more about how the currency markets actually worked, and that their forecasts were essentially useless. (I could tell by looking at their shoes that they hadn't struck it rich.)

Further Reading

If you want to learn about a wide variety of mathematical-modelling methods then you can't do much better that reading all of Professor Jim Murray's *Mathematical Biology*, published by Springer and now in two volumes. To date it is the only mathematics book I have ever read in bed.

For more about cellular automata see Stephen Wolfram's *A New Kind of Science*, published in 2002 by Wolfram Media.

For fractals anything by Benoit Mandelbrot, for example on the financial markets *The (Mis)Behaviour of Markets: A Fractal View of Risk, Ruin and Reward* published by Profile Books.

For chaos there is the classic *Chaos: Making a New Science* by James Gleick, published by Penguin.

Chapter 2

General Matters

There are some issues that are common to many branches of machine learning. In this chapter I put these all together for some brief discussion. More often than not in later chapters I will simply refer you to this chapter for further consideration.

2.1 Jargon And Notation

Every technical subject has its own jargon, and that includes machine learning. Important jargon to get us started is as follows. (Every chapter will introduce more.)

One small complaint here: Some rather simple things I've grown up with in applied mathematics seem to be given fancy names when encountered in machine learning. I'll say no more.

- **Data (points, sets):** To train our machine-learning algorithm we will need lots of data to input. Each data point could also be called an example or a sample. If we have a supervised-learning problem each input (vector) will have an associated output (vector). You can think of these as the independent and associated dependent variables.

- **Feature (vector):** A feature is a characteristic of the data. If you are from statistics then think of it as an explanatory variable. Although in machine learning we will often have many more explanatory variables than in classical stats models.

  It might be numerical (height of tree 6.3m) or it might be descriptive (eye colour blue). Often if it is descriptive then you will still have to give it a numerical label in order to do mathematical manipulations.

For example, you might want to group cats into different breeds, and the features might be length of fur (numerical) or whether or not it has a tail (descriptive but easily represented by value zero or one). (No tail? If you are wondering, the Manx cat is tailless.) Each data point will often be represented by a feature vector, each entry in the vector representing a feature.

- **Classification:** We will often be dividing our data into different classes or categories. In supervised learning this is done *a priori*. (Is the washing machine reliable or not?) In unsupervised learning it happens as part of the learning process. Sometimes we represent classes via vectors. Classifying sandwiches we might represent ham as $(1, 0, \ldots, 0)$, cheese as $(0, 1, \ldots, 0)$, and so on. (Although I tend to use column vectors in this book.) When only one of the entries in the feature vector is 1 and the rest 0 it is called one-hot encoding.

- **Regression:** You want to output a numerical forecast? Then you need to do a regression. This contrasts with the above classification problems where the result of an algorithm is a class.

I have tried to make my notation consistent throughout this book, and also to align with common conventions as much as possible.

- I have tended to use the superscript $\cdot^{(n)}$ to denote the n^{th} out of a total of N data points. That means I would have, say, N flowers to categorize, and a general flower would have the label n. The class, output or value of the n^{th} individual will be $y^{(n)}$.

- Vectors are bold face, and are column vectors. The vector of features of the n^{th} individual will be $\mathbf{x}^{(n)}$.

- I use subscript m to denote the m^{th} feature out of a total of M features. A feature might be salary, or number of words in an email.

- When I have categories, classes or clusters I will use K to be the total number of categories, etc. and k for one of them.

I keep jargon to a minimum but hope to include all that is important.

2.2 Scaling

The first part of many algorithms that you'll be seeing is a transformation and a scaling of the data. We are often going to be measuring distances

between data points so we must often make sure that the scales for each feature are roughly the same.

Suppose we were characterising people according to their salary and number of pets. If we didn't scale then the salary numbers, being in the thousands, would outweigh the number of pets. Then the number of pets a person had would become useless, which would be silly if one was trying to predict expenditure on dog food.

All we have to do is adjust entries for each feature by adding a number and multiplying by a number so as to make sure that the scales match.

This is easy to do. And there are a couple of obvious ways (a second is in the parentheses below). With the original unscaled data take one of the features, say the top entry in the feature vector, measure the mean and standard deviation (or minimum and maximum) across all of the data. Then translate and rescale the first entries in all of the feature vectors to have a mean of zero and a standard deviation of one (or a minimum of zero and a maximum of one). Then do the same for the second entry, and so on.

Just to be clear we use the same translation and scaling for the same feature across all data points. But it will be a different translation and scaling for other features.

This scaling and translation is a common first step in many machine-learning algorithms. I don't always mention the rescaling at the start of each chapter so please take it as read.

And don't forget that when you have a new sample for prediction you must scale all its features first, using the in-sample scaling.

One small warning though, if you have outliers then they can distort the rest of the data. For example one large value for one feature in one sample can, after rescaling, cause the values for that feature for the rest of the samples to be tiny and thus appear to be irrelevant when you measure distances between feature vectors. How important this effect is will depend on what type of rescaling you use. Common sense will help determine if and how you should rescale.

2.3 Measuring Distances

As I just said, we'll often be working with vectors and we will want to measure the distances between them. The shorter the distance between two feature vectors the closer in character are the two samples they represent. There are many ways to measure distance.

Euclidean distance The classic measurement, using Pythagoras, just square the differences between vector entries, sum these and square root.

This would be the default measure, as the crow flies. It's the L^2 norm.

Manhattan distance The Manhattan distance is the sum of the absolute values of the differences between entries in the vectors. The name derives from the distance one must travel along the roads in a city laid out in a grid pattern. This measure of distance can be preferable when dealing with data in high dimensions. This is the L^1 norm.

Chebyshev distance Take the absolute value of the differences between all the entries in the two vectors and the maximum of these numbers is the Chebyshev distance. This can be the preferred distance measure when you have many dimensions and most of them just aren't important in classification. This is the L^∞ norm.

Cosine similarity In some circumstances two vectors might be similar if they are pointing in the same direction even if they are of different lengths. A measure of distance for this situation is the cosine of the angle between the two vectors. Just take the dot product of the two vectors and divide by the two lengths. This measure is often used in Natural Language Processing, e.g. with Word2vec, mentioned briefly later.

2.4 Curse Of Dimensionality

In any problem involving data in high dimensions in which one is measuring distances one has to be careful not to be struck by the curse of dimensionality. Sounds frightening. Suppose you are working in M dimensions, that is your data has M features. And suppose you have N data points. Having a large number of data points is good, the more the merrier. But what about number of features?

You might think that the more features you have the better because it means that you can be more precise in classification of data. Unfortunately it doesn't quite work like that. Think how those N data points might be distributed in M dimensions. Suppose that the numerical data for each feature is either zero or one, to keep things simple. There will therefore be 2^M possible combinations. If N is less than this value then you run the risk of having every data point being on a different corner of the M-dimensional hypercube. The consequence of this is that the distances on which you base your analysis become somewhat meaningless. If everyone is unique then classification becomes meaningless.

There are only seven billion people on the planet but certain companies are trying to collect so much data on each of them/us that you can imagine the curse of dimensionality becoming an issue. The solution? Don't collect so much data, Mark.

In practice things are not usually as bad as all that, this is very much a worst-case scenario. It is certainly possible that there is some relationship between features so that your 13-dimensional, say, problem might only be effectively in six dimensions. Sometimes you might want to consider doing some preparatory analysis on your features to see if the dimension can be reduced. Which leads us onto ...

2.5 Principal Components Analysis

Principal Components Analysis (PCA) is a technique for finding common movements in data. For example you might have data for different crimes in different places (and we will have this data later). It seems reasonable that a place in which there are a large number of "Violence against the person - with injury" cases might also have a large number of "Violence against the person - without injury" cases. You might think about throwing away one of these features (a strange word for a crime!) in order to reduce your problem's dimensions. You could do that or you could do something a bit more sophisticated by using PCA.

PCA involves first finding the correlation matrix for all of your original features and then finding the eigenvectors and eigenvalues of this matrix. The eigenvector associated with the largest eigenvalue is the first principal component, and so on. From the sizes of the eigenvalues you get a measure of how much each eigenvalue contributes to the overall behaviour of your data. The larger the eigenvalue then the more it contributes. I'm going to assume that my readers know about eigenvalues and eigenvectors.

Now all you have to do is write the feature vector for each data point as the sum of the principal component vectors. That's easy, and unique, because the principal component vectors will almost certainly be orthogonal if you've used enough real data. In this way your problem is recast in terms of the principal components, and then, rather than throwing out all the "Violence against the person - without injury" cases you would instead throw out the principal component with the smallest eigenvalue. Or rather you only keep those components that give a total contribution of 95%, or 99%, or whatever level you deem suitable. And how do we know those percentages? Easy just divide each component's eigenvalue by the sum of all the eigenvalues and that gives you the fraction explained by that component.

2.6 Maximum Likelihood Estimation

Maximum Likelihood Estimation (MLE) is a common method for estimating parameters in a statistical/probabilistic model. In words, you simply find the parameter (or parameters) that maximizes the likelihood of observing what actually happened. Let's see this in a few classical examples.

Example: Taxi numbers

You arrive at the train station in a city you've never been to before. You go to the taxi rank so as to get to your final destination. There is one taxi, you take it. While discussing European politics with the taxi driver you notice that the cab's number is 1234. How many taxis are in that city?

To answer this we need some assumptions. Taxi numbers are positive integers, starting at 1, no gaps and no repeats. We'll need to assume that we are equally likely to get into any cab. And then we introduce the parameter N as the number of taxis. What is the MLE for N?

Well, what is the probability of getting into taxi number 1234 when there are N taxis? It is

$$\frac{1}{N} \quad \text{for} \quad N \geq 1234 \quad \text{and zero otherwise.}$$

What value of N maximizes this expression? Obviously it is $N = 1234$. That is the MLE for the parameter N. It looks a bit disturbing because it seems a coincidence that you happened to get into the cab with the highest number. But then the probability of getting into any cab is equally likely. It is also disturbing that if there are N taxis then the average cab number is $(N+1)/2$, and we somehow feel that should play a role.

Example: Coin tossing

Suppose you toss a coin n times and get h heads. What is the probability, p, of tossing a head next time?

The probability of getting h heads from n tosses is, assuming that the tosses are independent,

$$\frac{n!}{h!(n-h)!}p^h(1-p)^{n-h} = \binom{n}{h}p^h(1-p)^{n-h}.$$

Applying MLE is the same as maximizing this expression with respect to p. This likelihood function (without the coefficient in the front that is independent of p) is shown in Figure 2.1 for $n = 100$ and $h = 55$. There is a very obvious maximum.

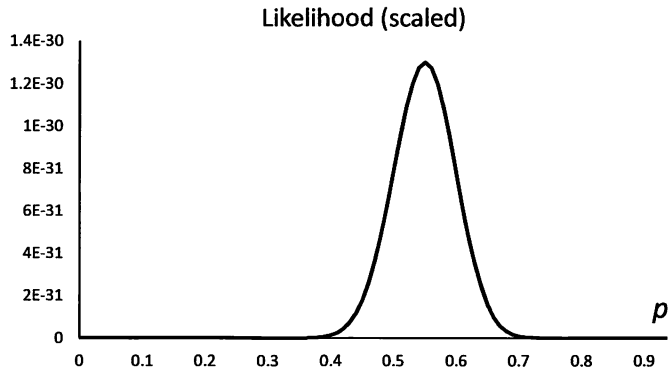


Figure 2.1: Likelihood versus probability p.

Often with MLE when multiplying probabilities, as here, you will take the logarithm of the likelihood and maximize that. This doesn't change the maximizing value but it does stop you from having to multiply many small numbers, which is going to be problematic with finite precision. (Look at the scale of the numbers on the vertical axis in the figure.) Since the first part of this expression is independent of p we maximize

$$h \ln p + (n - h) \ln(1 - p) \tag{2.1}$$

with respect to p. See Figure 2.2.

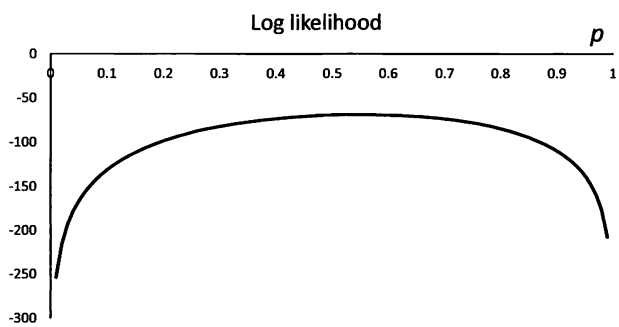


Figure 2.2: Log likelihood versus probability p.

This just means differentiating with respect to p and setting the derivative equal to zero. This results in

$$p = \frac{h}{n},$$

which seems eminently reasonable.

Example: A continuous distribution and more than one parameter

For the final example let's say we have a hat full of random numbers drawn from a normal distribution but with unknown mean and standard deviation, that's two parameters. The probability of drawing a number x from this hat is

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right),$$

where μ is the mean and σ the standard deviation which are both to be estimated. The log likelihood is then the logarithm of this, i.e.

$$\ln(p(x)) = -\frac{1}{2}\ln(2\pi) - \ln\sigma - \frac{1}{2\sigma^2}(x-\mu)^2.$$

If draws are independent then after N draws, x_n, the likelihood is just

$$p(x_1)p(x_2)\ldots p(x_N) = \prod_{n=1}^{N} p(x_n).$$

And so the log-likelihood function is

$$\ln\left(\prod_{n=1}^{N} p(x_n)\right) = \sum_{n=1}^{N} \ln(p(x_n)).$$

This gives us the nice log likelihood of

$$-N\ln\sigma - \frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2 \tag{2.2}$$

for the normal distribution. (I've dropped a constant at the front because it doesn't affect the position of the maximum.)

An example is plotted in Figure 2.3. For this plot I generated ten random xs from a normal distribution with mean of 0.1 and standard deviation of 0.2. These ten numbers go into the summation in expression (2.2), and I've plotted that expression against μ and σ. The maximum should be around $\mu = 0.1$ and $\sigma = 0.2$. (It will be at *exactly* whatever is the actual mean and standard deviation of the ten random numbers I generated, as I'll show next.)

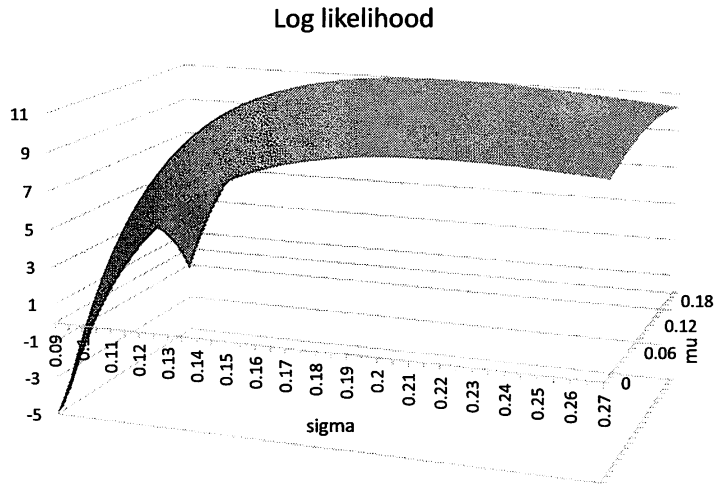


Figure 2.3: Log likelihood for the normal distribution.

To find the maximum likelihood estimate for μ you just differentiate with respect to μ and set this to zero. This gives

$$\mu = \frac{1}{N}\sum_{n=1}^{N} x_n.$$

I.e. μ is the simple average. And differentiating with respect to σ gives

$$\sigma = \frac{1}{N}\sum_{n=1}^{N}(x_n - \mu)^2. \tag{2.3}$$

And this again looks as expected.

You could also use this technique to decide among different distributions. Suppose you have three hats full of random numbers. For each hat you know the distribution and parameters. Someone picks one of the hats, you don't know which one it is, and they start drawing out numbers from it. You could use MLE to tell you which hat is the one most likely to have been chosen.

All of the above has been done analytically, i.e. finding the parameters that maximize the likelihood. But that won't be happening when we use MLE in machine learning for more complicated problems. Then we'll need numerical methods.

2.7 Confusion Matrix

A confusion matrix is a simple way of understanding how well an algorithm is doing at classifying data. It is really just the idea of false positives and false negatives.

Let's take a simple example.

An algorithm looks at pictures of fruit in order to classify them as apples, pears, oranges, etc. Most it gets right, some it gets wrong. Can we quantify this?

There are 100 items of fruit. Let's focus on how well our algorithm identifies apples.

		ACTUAL CLASS	
		Apple	Non apple
PREDICTED CLASS	Apple	11 TP	9 FP
	Non apple	5 FN	75 TN

Figure 2.4: (Mis)Identification of apples. TN = True Positives, FP = False Positives, etc.

Sixteen of the 100 are apples. The rest are a mix of pears, oranges, bananas etc. Our algorithm labels each item as whatever fruit it thinks it is. It thinks there are 20 apples. Unfortunately the algorithm had both identified as apples some that weren't and some that were apples it missed. Only 11 of the apples did it correctly identify. So it had some false positives and some false negatives. This is shown in Figure 2.4.

In order to interpret these numbers they are often converted into several rates:

- Accuracy rate: $\frac{TP+TN}{Total}$ where Total = TP + TN + FP + FN. This measures the fraction of times the classifier is correct
- Error rate: $1 - \frac{TP+TN}{Total}$.

- True positive rate or Recall: $\frac{TP}{TP+FN}$
- False positive rate: $\frac{FP}{TN+FP}$
- True negative rate or Specificity: $1 - \frac{FP}{TN+FP}$
- Precision: $\frac{TP}{TP+FP}$. If the algorithm predicts apple how often is it correct?
- Prevalence: $\frac{TP+FN}{Total}$. What is the fraction of apples?

Which of these measures is best depends on the context. For example if you are testing for cancer then you will want a small false negative rate and aren't too concerned about high false positives. On the other hand you don't want to go around predicting everyone has cancer.

It is useful to compare numbers with the null error rate. That means how often would you be wrong if you always predicted the largest class. In the figure we have non apple being the largest actual category, so if we called every fruit a non apple then we would be wrong $\frac{11+5}{11+5+9+75}$ of the time. The null error rate is particularly useful for comparison if one class is much larger than the other.

Considering that there are only four numbers here (three if you scale to have 100, say, as the total) there are an awful lot of ways they are sliced and diced to give different interpretative measures. On the Wikipedia page for Confusion Matrix I counted 13 different measures. Here's a good one, the Matthews correlation coefficient, defined by

$$\frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}.$$

This is a type of correlation that measures how good the classifier is. The number it yields is between plus and minus one. Plus one means perfect prediction, zero means no better than random. This measure is good when you have unbalanced numbers, with one category being much larger or smaller than others.

Receiver operating characteristic

Yet another way of looking at how well an algorithm is classifying is the receiver operating characteristic or ROC (curve). This is plotted, an example is shown in Figure 2.5, as follows. Suppose there is a threshold/parameter

in your algorithm that determines whether you have an apple or a non apple. Plot the true positive rate against the false positive rate as the threshold/parameter varies. Points above the 45° diagonal are good, and the further into the top left of the plot the better.

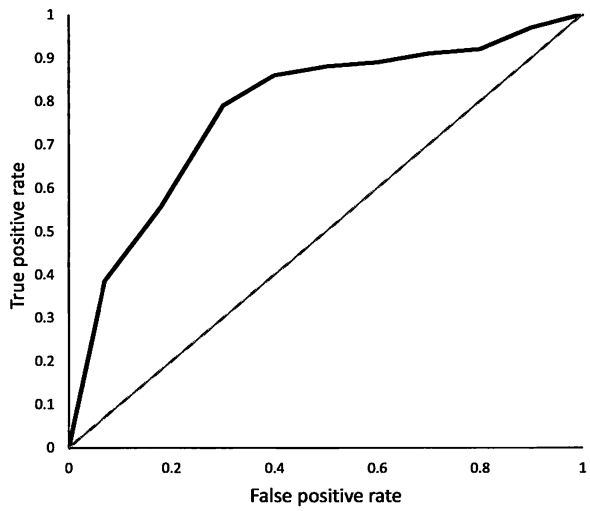


Figure 2.5: Receiver operating characteristic.

The area under the ROC curve (the AUC) is then a measure of how good different algorithms are. The closer to one (the maximum possible) the better. If you enter Kaggle competitions they often use the AUC to rank competitors. (If you do enter Kaggle competitions after reading this book, and I hope you will, don't forget my 10% cut of winnings. You did read the small print didn't you?)

2.8 Cost Functions

In machine learning a cost function or loss function is used to represent how far away a mathematical model is from the real data. One adjusts the mathematical model, usually by varying parameters within the model, so as to minimize the cost function. This is then interpreted as giving the best model, of its type, that fits the data.

Cost functions can take many forms. They are usually chosen to have properties that make some mathematical sense for the problem under investigation and also to have some tractability.

Let's look at an example. Suppose we have data for age and salary of a sample of lawyers then we might want to look for a linear relationship between the two. Old lawyers earn more perhaps. We might represent the n^{th} lawyer's age by $x^{(n)}$ and their salary by $y^{(n)}$. And we'll have N of them. See Figure 2.6, although the numbers here are made up. Note that I haven't bothered to scale the features, the xs. That's because I'm not measuring any distances between feature vectors here.

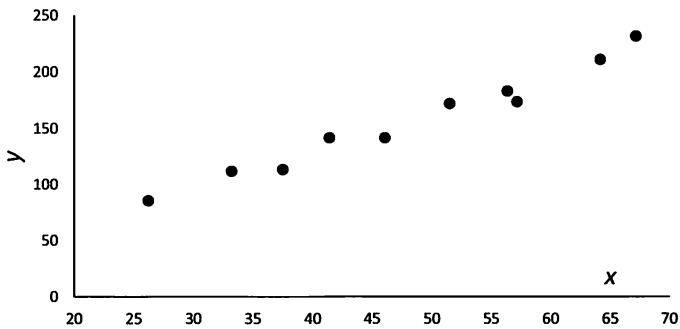


Figure 2.6: Fit a straight line through these points.

We want to find a relationship of the form

$$y = \theta_0 + \theta_1 x, \tag{2.4}$$

where the θs are the parameters that we want to find to give us the best fit to the data. (Shortly I'll use $\boldsymbol{\theta}$ to represent the vector with entries being these θs.) Call this linear function $h_\theta(x)$, to emphasise the dependence on both the variable x and the two parameters, θ_0 and θ_1.

We want to measure how far away the data, the $y^{(n)}$s, are from the function $h_\theta(x)$. A common way to do this is via the quadratic cost function

$$J(\boldsymbol{\theta}) = \frac{1}{2N} \sum_{n=1}^{N} \left(h_\theta(x^{(n)}) - y^{(n)} \right)^2. \tag{2.5}$$

This is just the sum of the squares of the vertical distances between the points and the straight line. The constant in front, the $\frac{1}{2N}$, makes no difference to the minimization but is there by convention, and can also be important when N is large, to scale quantities.

We want the parameters that minimize (2.5). This is called ordinary least squares (OLS).

This is a popular choice for the cost function because obviously it will be zero for a perfect, straight line, fit, but it also has a single minimum. This

minimum is easily found analytically, simply differentiate (2.5) with respect to both θs and set the results to zero:

$$\sum_{n=1}^{N}\left(\theta_0+\theta_1 x^{(n)}-y^{(n)}\right)=\sum_{n=1}^{N}x^{(n)}\left(\theta_0+\theta_1 x^{(n)}-y^{(n)}\right)=0.$$

This can be trivially solved for the two θs:

$$\theta_0=\frac{\left(\sum y\right)\left(\sum x^2\right)-\left(\sum x\right)\left(\sum xy\right)}{N\sum x^2-\left(\sum x\right)^2}$$

and

$$\theta_1=\frac{N\left(\sum xy\right)-\left(\sum x\right)\left(\sum y\right)}{N\sum x^2-\left(\sum x\right)^2}.$$

I have dropped all the (n) superscripts in these to make them easier to read, it's obvious what is meant.

The cost function as a function of θ_0 and θ_1 is shown in Figure 2.7 using the same data as in Figure 2.6. Actually it's the logarithm of the cost function. This makes it easier to see the minimum. Without taking logs the wings in this plot would dominate the picture. And the fitted line is shown in Figure 2.8.

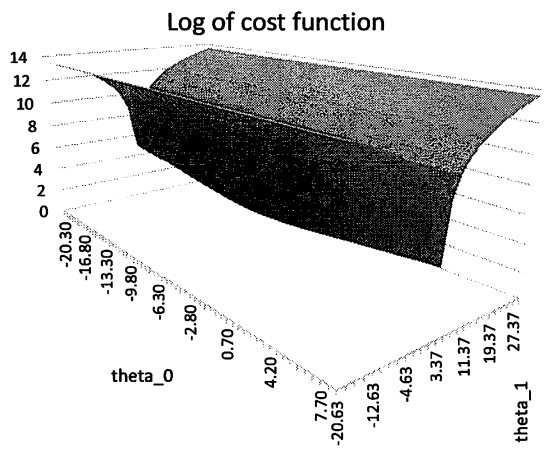


Figure 2.7: The logarithm of the cost function as a function of the two parameters.

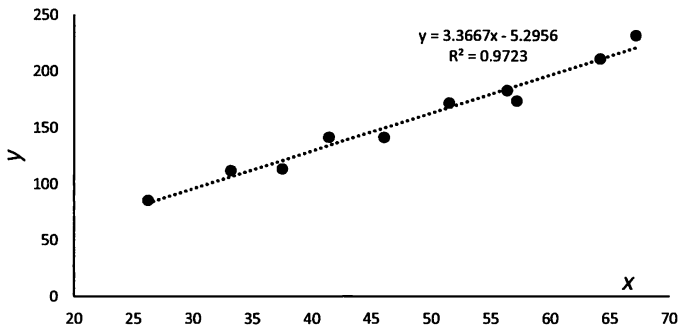


Figure 2.8: The linear fit.

Many machine learning methods fit models by looking for model parameters that minimize some cost function.

Later we'll see cost functions in higher dimensions, i.e. more explanatory features (as well as the lawyer's age we could also include the annual fee for the school he went to, etc.).

In some problems minimizing a cost function and maximizing a likelihood can lead to the same or similar mathematical optimization problems. Often this is when some aspect of the model is Gaussian. And, of course, applying MLE really requires there to be a probabilistic element to your problem, whereas the cost-function approach does not.

Other cost functions are possible, depending on the type of problem you have. For example if we have a classification problem — "This is an apple" instead of "0.365" — then we won't be fitting with a linear function. If it's a binary classification problem then apples would be labelled 1, and non apples would be 0, say. And every unseen fruit that we want to classify is going to be a number between zero and one. The closer to one it is the more likely it is to be an apple.

In such a classification problem we would want to fit a function that ranged from zero to one, to match the values for the class labels. And linear, as we'll see later, would be a silly choice. Typically we fit an S-shaped sigmoidal function. Probably the most common example would be

$$h_\theta(x) = \frac{1}{1 + e^{-\theta_0 - \theta_1 x}}.$$

The quadratic cost function is just not suitable in this situation. Instead we often see the following.

$$J(\boldsymbol{\theta}) = -\frac{1}{N} \sum_{n=1}^{N} \left(y^{(n)} \ln \left(h_\theta(x^{(n)}) \right) + (1 - y^{(n)}) \ln \left(1 - h_\theta(x^{(n)}) \right) \right).$$

This looks a bit strange at first, it's not immediately clear where this has come from. But if you compare it with (2.1) then you'll start to see there might be a link between minimizing this cost function and maximizing the likelihood of something (there's a sign difference between the two). The analogy is simply that the ys represent the class of the original data (the head/tail or apple/non apple) and the h_θ is like a probability (for y being 1 or 0). More anon.

This cost function is also called the cross entropy between the two probability distributions, one distribution being the ys, the empirical probabilities, and the other being h_θ, the fitted function. We'll see more about the cross entropy in a couple of sections' time.

Regularization

Sometimes one adds a regularization term to the cost function. Here's an example applied to OLS in a single variable:

$$J(\boldsymbol{\theta}) = \frac{1}{2N}\left(\sum_{n=1}^{N}\left(h_\theta(x^{(n)}) - y^{(n)}\right)^2 + \lambda\theta_1^2\right).$$

The addition of the second, regularization, term on the right encourages the optimization to find simpler models. Note that it doesn't include the θ_0 parameter. The larger the θ_1 the greater the change to the forecast for a small change in the independent variable. So we might want to reduce it.

It's not easy to appreciate the importance of the regularization term with a simple one-dimensional linear regression. Instead think of fitting a polynomial to some data. If you fit with a high-order polynomial you will get a lot of rapid wiggling of the fitted line. Is that real, or are you over fitting? The regularization term reduces the size of all the coefficients, except for the first one representing a constant level, thus reducing such possibly irrelevant wiggles.

Back to the linear regression in one dimension, the minimum is now at

$$\theta_0 = \frac{\left(\sum y\right)\left(\lambda + \sum x^2\right) - \left(\sum x\right)\left(\sum xy\right)}{N\left(\lambda + \sum x^2\right) - \left(\sum x\right)^2}$$

and

$$\theta_1 = \frac{N\left(\sum xy\right) - \left(\sum x\right)\left(\sum y\right)}{N\left(\lambda + \sum x^2\right) - \left(\sum x\right)^2}.$$

A little warning, just because I've just given a few explicit solutions for where the minimum is don't expect this to happen much in this book. More often than not we'll be minimizing cost functions etc. numerically.

The idea of regularization is easily generalized to higher dimensions.

2.9 Gradient Descent

So you want to find the parameters that minimize a loss function. And almost always you are going to have to do this numerically. We'll even see an example later where there is technically an explicit solution for the minimizing parameters but since it involves matrix inversion you might as well go straight into a numerical solution anyway.

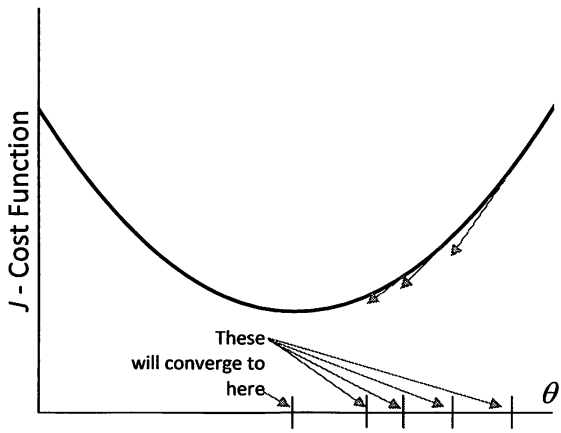


Figure 2.9: Illustrating gradient descent. Change θ in the direction of the slope, a distance proportional to the slope and a parameter β.

If we have a nice convex function then there is a numerical method that will converge to the solution, it is called gradient descent. For gradient descent to work straight out of the box we really ought to have a cost function with a single minimum, the global minimum, otherwise we are likely to converge to a local minimum. But things can be done even if there are local minima, they'll get a mention shortly.

The method is illustrated in Figure 2.9.

The scheme works as follow. Start with an initial guess for each parameter θ_k. Then move θ_k in the direction of the slope:

$$\text{New } \theta_k = \text{Old } \theta_k - \beta \, \partial J / \partial \theta_k.$$

Update all θ_k simultaneously, and repeat until convergence.

Here β is a learning factor that governs how far you move. If β is too small it will take a long time to converge. If too large it will overshoot and might not converge at all. It is clear, as I mentioned, that this method might converge to only a local minimum if there is more than one such.

The loss function J is a function of all of the data points, i.e. their actual values and the fitted values. In the above description of gradient descent we have used all of the data points simultaneously. This is called batch gradient descent. But rather than use all of the data in the parameter updating we can use a technique called stochastic gradient descent. This is like batch gradient descent except that you only update using *one* of the data points each time. And that data point is chosen randomly. Hence the stochastic.

Let me write the cost function as

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{N} J_n(\boldsymbol{\theta}),$$

with the obvious interpretation. So in the case of OLS, Equation (2.5), we would have

$$J_n(\boldsymbol{\theta}) = \frac{1}{2N}\left(h_\theta(x^{(n)}) - y^{(n)}\right)^2.$$

$J_n(\boldsymbol{\theta})$ is the contribution of data point n to the cost function.

Stochastic gradient descent means pick an n at random and then update according to

$$\text{New } \theta_k = \text{Old } \theta_k - \beta\, \partial J_n/\partial\theta_k.$$

Repeat, picking another data point at random, etc.

There are several reasons why we might want to use stochastic gradient descent instead of batch. For example:

- Since the data points used for updating are chosen at random the convergence will not be as smooth as batch gradient descent. But surprisingly this can be a good thing. If your loss function has local minima then the bouncing around can bounce you past a local minimum and help you converge to the global minimum.

- It can be computationally more efficient. If you have a very large dataset it is likely that many of the data points are similar so you don't need to use all of them. If you did it would take longer to update the parameters without adding much in terms of convergence.

And then there's mini batch gradient descent in which you use subsets of the full dataset, bigger than 1 and smaller than n, again chosen randomly.

Finally if you are using a stochastic version of gradient descent you could try smoothing out the bouncing around, if it's not being helpful, by taking an average of the new update and past updates, leading to an exponentially weighted updating. This is like having momentum in your updating and can speed up convergence. You'll need another parameter to control the amount of momentum.

2.10 Training, Testing And Validation

Most machine-learning algorithms need to be trained. That is, you give them data and they look for patterns, or best fits, etc. They know they are doing well when perhaps a loss function has been minimized, or the rewards have been maximized. You'll see all of this later. But you have to be careful that the training is not overfitting. You don't want an algorithm that will give perfect results using the data you've got. No, you want an algorithm that will do well when you give it data it hasn't seen before.

When doing the training stage of any algorithm you could use all of the data. But that doesn't tell you how robust the model is. To do this properly there are several things you should try. First of all, divide up the original data into training and test sets. Do this randomly, maybe use 75% for training and then the remaining 25% for testing. How well does the trained algorithm do on the test data?

Another thing you can do if there is any time dependence in your data — maybe it was collected over many years — instead of taking 25% out at random you split the data into two groups, before and after a certain date. Now train these two data sets independently. Do you get similar results? If you do then that's great, the model is looking reliable. If you don't then there are three, at least, explanations. One is that the model is fine but there are regions in one of the two halves that are just not reached in the other half. Second, there might have been a regime shift. That is, the world has changed, and you need a time-dependent model. Or maybe you are simply barking up the wrong tree with whatever algorithm you are using.

Epochs

In some machine-learning methods one uses the same training data many times, as the algorithm gradually converges, for example, in stochastic gradient descent. Each time the whole training set of data is used in the training that is called an epoch or iteration. Typically you won't get decent results until convergence after many epochs.

One sees a decreasing error as the number of epochs increases, as shown in Figure 2.10. But that does not mean that your algorithm is getting better, it could easily mean that you are overfitting.

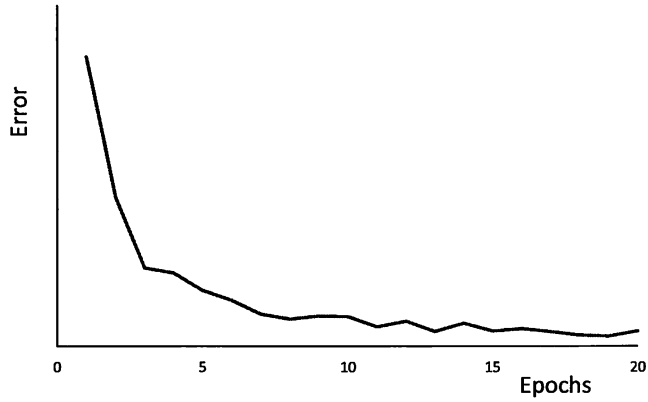


Figure 2.10: Error versus epochs during training.

This can happen if the algorithm has seen the training data too many times, i.e. there have been too many epochs. To test for this you introduce the test set of data, the data that you have held back. All being well you will get results like in Figure 2.11. The test set is not as good as the training set, obviously, but both are heading in the right direction. You can stop training when the errors have levelled out consistently. There is a *caveat* to this, if the test error is much bigger than the training error then that could also be a sign of overfitting.

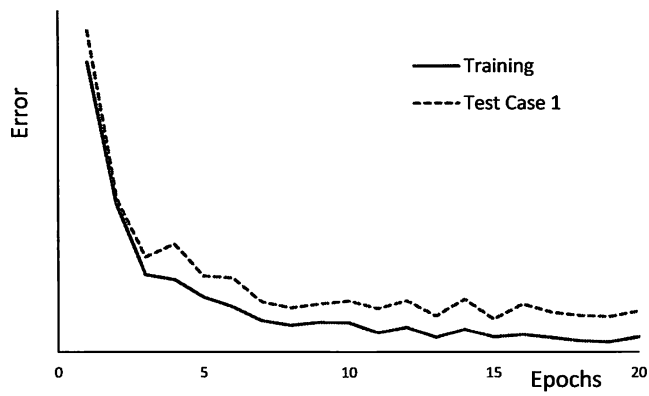


Figure 2.11: Training and testing are looking good. Stop training when the error levels out.

On the other hand if you get results like in Figure 2.12 where the test-set error begins to rise again then you have overfitted.

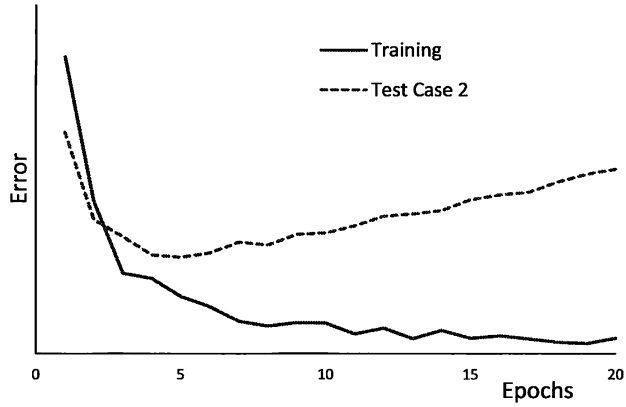


Figure 2.12: Looks like we have overfitting.

To help avoid overfitting sometimes we divide up our original data into *three* sets. The third data set is called the validation set. We train the data on the training set as before. Then after a few epochs we use the validation set to see how the algorithm copes with out-of-sample data. We keep doing this with more and more epochs. If we see the error for the validation set rising then we stop at that number of epochs. But we can't be sure that the algorithm hasn't learned to cope with new samples because our algorithm knows about the validation set since we used it to decide when to stop. So now we bring in the test data and see how the algorithm copes with that.

Note that we don't always plot error versus epoch. There'll be times when we plot error versus a model parameter. This can help us choose the best parameters for our model.

2.11 Bias And Variance

Suppose there is a relationship between an independent variable, x, and a dependent variable, y, given by

$$y = f(x) + \epsilon.$$

Here ϵ is an error term, ϵ has mean of zero (if it wasn't zero it could be absorbed into $f(x)$) and variance σ^2. The error term, whose variance could also be x dependent, captures either genuine randomness in the data or noise due to measurement error.

And suppose we find, using one of our machine-learning techniques, a deterministic model for this relationship:

$$y = \hat{f}(x).$$

Now f and $\hat{f}$ won't be the same. The function our algorithm finds, $\hat{f}$, is going to be limited by the type of algorithm we use. And it will have been fitted using a lot of real training data. That fitting will probably be trying to achieve something like minimizing

$$\sum_{n=1}^{N} \left(\hat{f}(x^{(n)}) - y^{(n)}\right)^2.$$

The more complex, perhaps in terms of parameters, the model then the smaller this error will be. Smaller, that is, for the training set.

Now along comes a new data point, x', not in the training set, and we want to predict the corresponding y'. We can easily see how much is the error in our prediction.

The error we will observe in our model at point x' is going to be

$$\hat{f}(x') - f(x') - \epsilon. \tag{2.6}$$

There is an important subtlety here. Expression (2.6) looks like it only has the error due to the ϵ. But it's also hiding another, possibly more important and definitely more interesting, error, and that is the error due to what is in our training data. A robust model would give us the same prediction whatever data we used for training our model.

So let's look at the average error, the mean of (2.6), given by

$$E\left[\hat{f}(x')\right] - f(x')$$

where the expectation $E[\cdot]$ is taken over random samples of training data (having the same distribution as the training data).

This is the definition of the bias,

$$\text{Bias}(\hat{f}(x')) = E\left[\hat{f}(x')\right] - f(x').$$

We can also look at the mean square error. And since $\hat{f}(x')$ and ϵ are independent we easily find that

$$E\left[(\hat{f}(x') - f(x') - \epsilon)^2\right] = \left(\text{Bias}(\hat{f}(x'))\right)^2 + \text{Var}(\hat{f}(x')) + \sigma^2, \tag{2.7}$$

where

$$\text{Var}(\hat{f}(x')) = E\left[\hat{f}(x')^2\right] - E\left[\hat{f}(x')\right]^2.$$

Expression (2.7) is the mean square error for our new prediction.

This shows us that there are two important quantities, the bias and the variance, that will affect our results and that we can control to some extent by tweaking our model. The squared error (2.7) is composed of three terms,

the bias of the method, the variance of the method and a term that we are stuck with (the σ^2).

A good model would have low bias and low variance as illustrated in the top left of Figure 2.13.

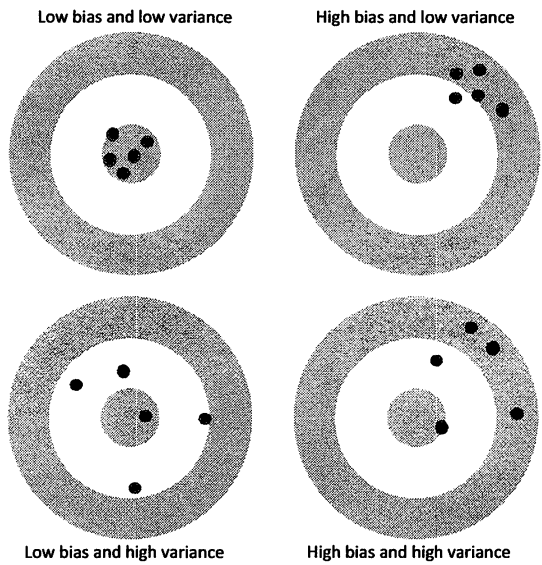


Figure 2.13: Examples of low and high bias and variance.

Bias is how far away the trained model is from the correct result on average. Where "on average" means over many goes at training the model, using different data. And variance is a measure of the magnitude of that error.

Unfortunately, we often find that there is a trade-off between bias and variance. As one is reduced, the other is increased. This is the matter of over- and underfitting.

Overfitting is when we train our algorithm too well on training data, perhaps having too many parameters for fitting. An analogy is the two-year old playing with his first jigsaw puzzle. After many attempts, and quite a few tears, he finally succeeds. Thereafter he finds it much easier to solve the puzzle. But has he really learned how to do jigsaw puzzles or has he just memorized the one picture?

Underfitting is when the algorithm is too simple or not sufficiently trained. It might work on average in some sense but fails to capture nuances of the data.

Let's see how this works with a simple example. We'll work with the data in Figure 2.14. This shows the relationship between y and x, including the random component ϵ. The function $f(x)$ is just a shifted sine wave, and there's a uniformly distributed random variable added to it. Each dot represents a sample, and the full training set is shown.

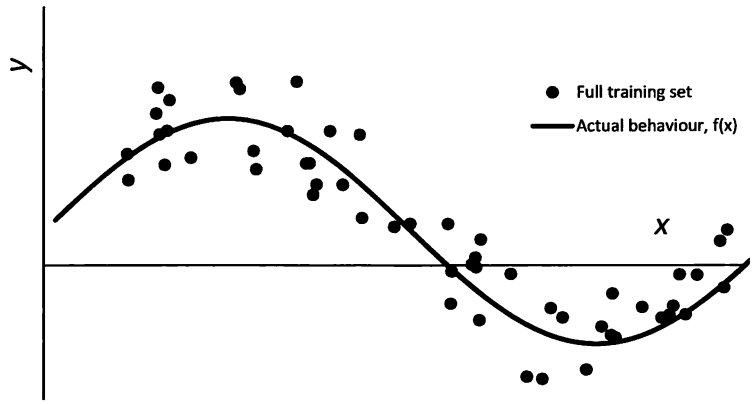


Figure 2.14: The true relationship between y and x (a shifted sine wave), also showing the random ϵ.

Let's start with a simple model for the relationship between x and y, i.e. a simple $\hat{f}(x)$.

It doesn't get any simpler than a constant. So in Figure 2.15 I have shown our first model, it is just a constant, the dashed line. I have chosen for this simple model the average y value for a *subset of all the data*. It doesn't actually matter which subset of the data because if I changed the subset the average wouldn't change all that much. This is clearly very underfitted, to say the least.

Along comes an unseen sample, represented by the hollow dot in the figure. But our forecast is the solid dot. There are three sources of error, the error due to the ϵ, the variance and the bias.

The ϵ we are stuck with, we can't forecast that, although it is in the training data and will therefore be implicitly within the model. The variance (of the model) is tiny. As I said, if I used a different subset of the training data then the model, here the average, would hardly change at all. So most of the model error is the bias.

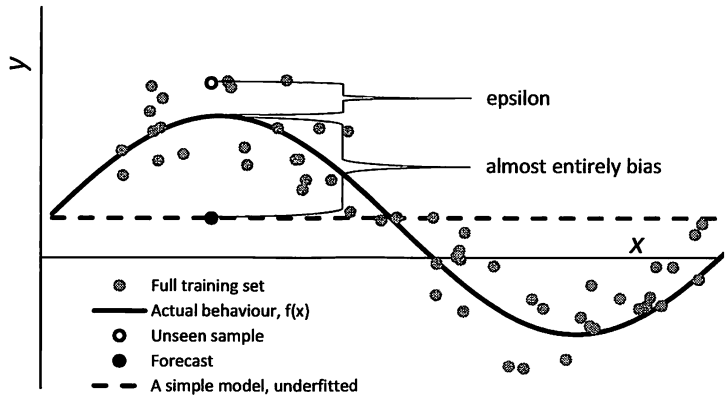


Figure 2.15: Underfitted model with high bias.

For an underfitted model the mean squared error will be dominated by the bias and the ϵ.

In the next two figures we see what happens when we use a more complicated model and two different subsets of the full set of training data. In both cases I have used Excel's smooth-fitting option with a subset of the data, so that's two different models and thus two different forecasts. The forecasts move about mainly because of the variance in the model.

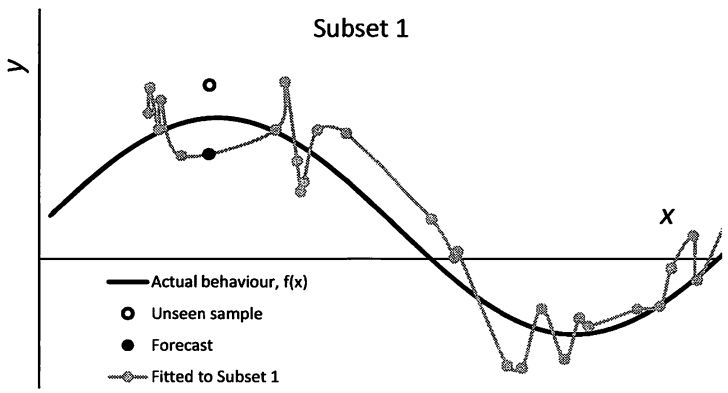


Figure 2.16: A complicated model (over)fitted to one subset of the training data.

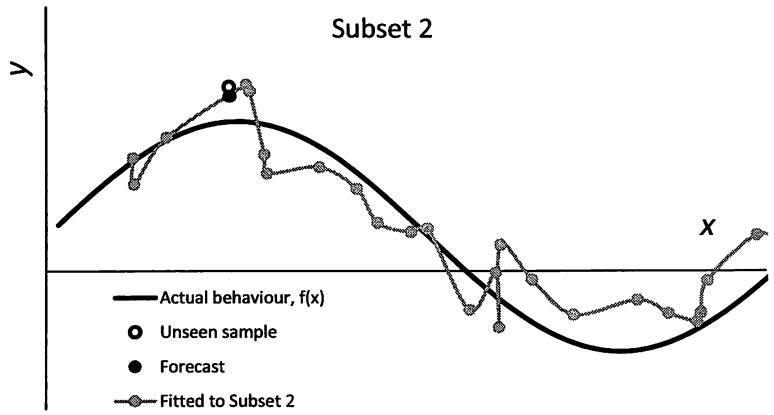


Figure 2.17: The same model (over)fitted to a different subset.

Finally, two more plots, fitted to the same subsets but using a simpler model, just a cubic polynomial. You see much less bias and variance now.

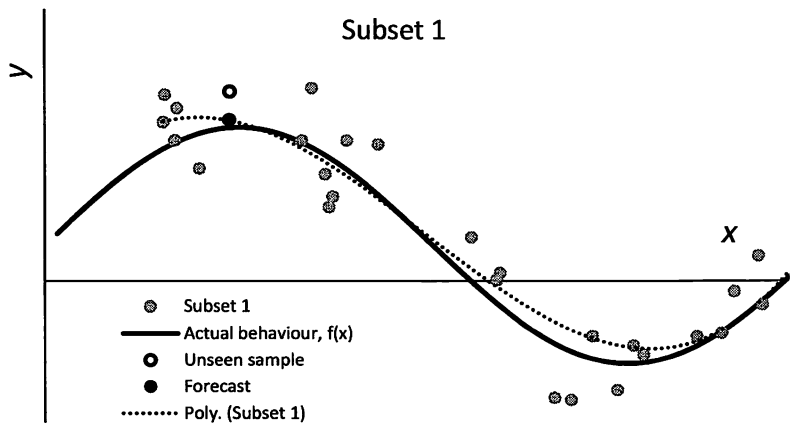


Figure 2.18: A good fit using a model that is not too simple or too complex.

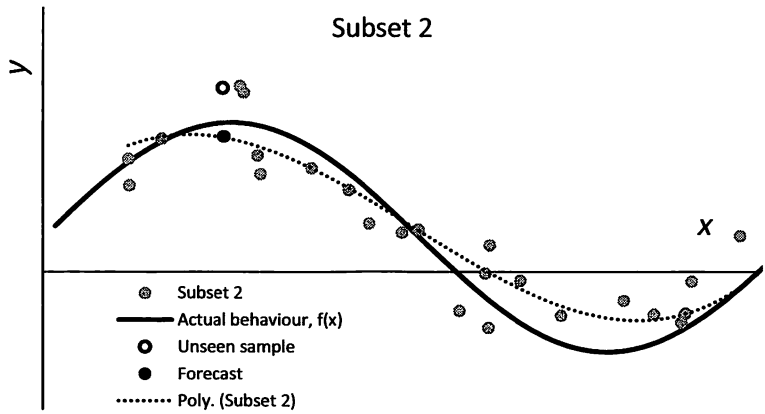


Figure 2.19: Same model, different dataset.

The trade off between bias and variance is shown in Figure 2.20.

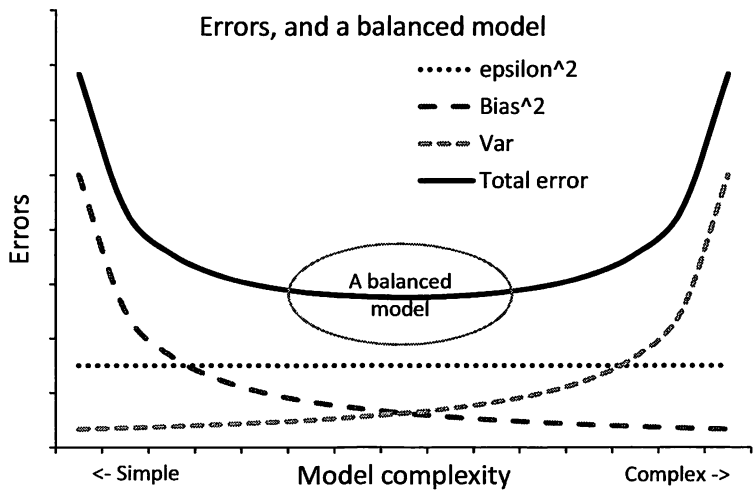


Figure 2.20: The various error terms.

2.12 Lagrange Multipliers

In using most machine-learning techniques there comes a point where we need to find some parameters. More often than not this must be done via an optimization problem. And sometimes our optimization problem is constrained, "Find the parameters that maximize... subject to the constraint ..."

Such constrained optimization problems can often be addressed via the method of Lagrange multipliers.

Suppose we want to find the maximum of the function $f(x_1, x_2)$ then we would differentiate with respect to the two variables, and find the values of the variables that make these two expressions zero. (This might give us a maximum, minimum or saddle point, so we'd have a tiny bit of extra work to do to see which one is the maximum.) But what if we want to maximize $f(x_1, x_2)$ subject to the constraint $g(x_1, x_2) = 0$, representing a curve in x_1, x_2 space? We can visualize this problem by thinking of x_1 and x_2 being latitude and longitude and f being the height of a hill. The curve $g(x_1, x_2) = 0$ is a path along the hillside as seen from above. So you walk along this path until you reach the highest point.

The method of Lagrange multipliers says that we can find the solution by looking at the Lagrangian

$$L = f(x_1, x_2) - \lambda g(x_1, x_2) \tag{2.8}$$

and finding its stationary points with respect to x_1, x_2 *and* λ.

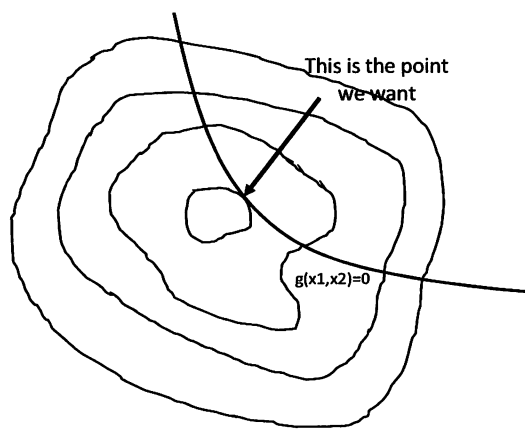


Figure 2.21: The contour map of the function $f(x_1, x_2)$ and the line $g(x_1, x_2) = 0$.

To explain why this works look at the contour map for our hill in Figure 2.21. Here we can see where the greatest value of f along the path is also where, in the view from above, the path and a contour touch. Geometrically this means that the vector that is orthogonal to the contour is in the same direction as the vector that is orthogonal to the constraint or mathematically

$$\left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}\right) \propto \left(\frac{\partial g}{\partial x_1}, \frac{\partial g}{\partial x_2}\right).$$

But differentiating (2.8) with respect to x_1 and x_2 gives us this, with λ being the constant of proportionality. And differentiating (2.8) with respect to λ is just the constraint again.

Although I've explained the method in two dimensions it is applicable in any number, and also with any number of constraints, just add to the Lagrangian terms linear in the constraints, each having its own λ.

The method can be generalized to inequality constraints. So we would now want to maximize $f(x_1, x_2)$ subject to $g(x_1, x_2) < 0$. We can still work with (2.3) but now

$$\lambda \geq 0$$

and

$$\lambda g(x_1, x_2) = 0.$$

This last expression is called complementary slackness and says that either $g(x_1, x_2) < 0$ and the constraint is satisfied so we don't need the λ, or $g(x_1, x_2) = 0$ and we are on the boundary of the permitted region and we are back with the earlier version of Lagrange multipliers. This generalization goes by the name of Karush–Kuhn–Tucker (KKT).

Be careful that Lagrange multipliers and KKT only give *necessary* conditions for a solution. So you will need to check that any solution they give actually works.

2.13 Multiple Classes

Many of the techniques we shall be learning classify data into two groups, an email is or isn't spam. For example the Support Vector Machine in its simplest version divides sample data according to which side of a hyperplane they lie. But what if we have more than one class, we have spam, not spam, and emails from the mother-in-law?

There are three techniques commonly used: One-versus-one classification; One-versus-all classification; Softmax function.

But first let me tell you something that is not, and should not be, used.

Suppose you want to determine who a painting is by. You might show your algorithm many paintings by various artists. You might want to use

numerical labels for each artist. Paintings by van Gogh might be labelled as 1, paintings by Monet as 2, by Cassatt as 3, and so on. And then you show it a painting by an unknown artist. It gives you its numerical output. It is 2.7. Who is that painting by? Well, it seems to think that it's quite likely by Cassatt but possibly by Monet. But that would be nonsense. There is no ordering of artists such that you can say that Monet is halfway between van Gogh and Cassatt. You would only have a single, scalar, output if the classes could be ordered in such a way that the associated numbers are meaningful. E.g. Which rides is a person allowed to go on at Disneyland? There might be three types of ride, ones for toddlers, ones for children and ones for adults. That's three classes but, according to the signs I've seen on rides, they correspond exactly to a person's height. So that would be fine.

Generally however you don't want to use a scalar to represent more than two classes. But you still might want to use numbers for classification. What can you do? Here are some common methods.

One versus one

If we have K classes, A, B, C, ..., then we will set up and train $K(K-1)/2$ binary classifiers using subsets of the training data: A vs B; B vs C; C vs A; .... Our new, unclassified, sample is then tested against each of these, gets a vote each time (for example resulting in (A,B,A)), and the chosen class is the one with the most votes (A).

One versus all

The one-vs-all technique requires classifiers to produce a degree of confidence for any decision, and not simply a class label. This limits its usefulness. But it works like this. If we have K classes then we train K binary classifiers: A vs not-A; B vs not-B; C vs not-C; .... The winning class is the one that has the highest confidence score.

Softmax function

Your classification algorithm could be trained on feature vectors in which one entry is one and all other entries are zero, the one-hot encoding mentioned earlier. But then the output vector for a new sample would be a K-dimensional vector but typically the entries could be anything. You can turn these outputs into something like a probability by using the softmax function. The softmax function takes an array of values $(z_1, z_2, \ldots, z_K)$,

whether positive or negative, and turns them into a new array with values between zero and one, and such that they sum to one:

$$\frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}$$

This gives you a probabilistic interpretation of which class a new sample is from. The entry with the highest value gives you its class, but you also get an idea of the confidence in that result.

2.14 Information Theory And Entropy

In its original form information theory concerned the transmission of messages. If you have a crackly telephone line then messages might be corrupted between the transmitter and the receiver. A simple Morse code message of dots and dashes could very easily be messed up as some dots turn to dashes and vice versa. If there is some probability of corruption of the data then how much information from the original message gets through and conversely how much longer must a message be to ensure, to some given accuracy, that the correct message is received?

Some of information theory is going to be important to us. These are the concepts of self information or surprisal (me too, I wasn't expecting such a word), and information entropy, or simply entropy.

If the sun rises in the East there is no information content, the sun always rises in the East. If you toss an unbiased coin then there is information in whether it lands heads or tails up. If the coin is biased towards heads then there is more information if it lands tails.

Surprisal associated with an event is the negative of the logarithm of the probability of the event

$$-\log_2(p).$$

People use different bases for the logarithm, but it doesn't make much difference, it only makes a scaling difference. But if you use base 2 then the units are the familiar bits. If the event is certain, so that $p = 1$, the information associated with it is zero. The lower the probability of an event the higher the surprise, becoming infinity when the event is impossible.

But why logarithms? The logarithm function occurs naturally in information theory. Consider for example the tossing of four coins. There are 16 possible states for the coins, HHHH, HHHT, ..., TTTT. But only four bits of information are needed to describe the state. HTHH could be represented by 0100.

$$4 = \log_2(16) = -\log_2(1/16).$$

Going back to the biased coin, suppose that the probability of tossing heads is 3/4 and 1/4 of tossing tails. If I toss heads then that was almost expected, there's not that much information. Technically it's $-\log_2(0.75) = 0.415$ bits. But if I toss tails then it is $-\log_2(0.25) = 2$ bits.

This leads naturally to looking at the average information, this is our entropy:

$$-\sum p \log_2(p),$$

where the sum is taken over all possible outcomes. (Note that when there are only two possible outcomes the formula for entropy must be the same when p is replaced by $1-p$. And this is true here.)

For an unbiased coin the entropy is easily found to be 1. For a coin with zero probability of either heads or tails then the entropy is zero. For the 75:25 coin it is 0.811. Entropy is a measure of uncertainty, but uncertainty linked to information content rather than the uncertainty associated with betting on the outcome of the coin toss.

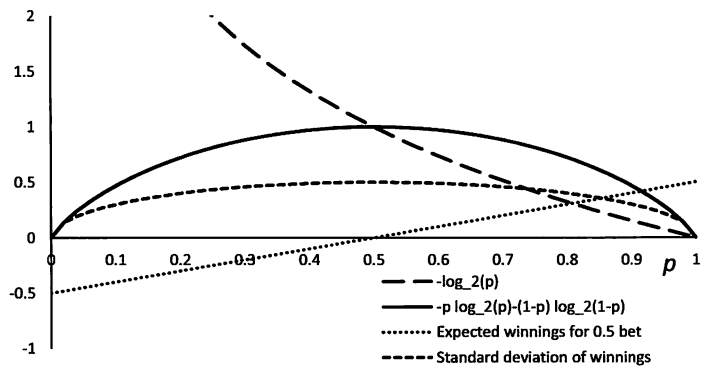


Figure 2.22: The information function, the entropy, expected winnings and standard deviation. See text for details.

In Figure 2.22 I plot four quantities for the coin tossing, all against p the probability of heads. There is the information function, the $-\log_2(p)$, the entropy $p\log_2(p) - (1-p)\log_2(1-p)$ and for comparison a couple of lines usually associated with coin tossing. One is the expected winnings for betting \$0.5 on heads, $p - 0.5$, and the standard deviation of the winnings $\sqrt{p(1-p)^2 + (1-p)p^2}$. You can see that the standard deviation, the risk, in the coin toss is qualitatively similar to the entropy.

Entropy is going to be important when we come to decision trees, it will tell us how to decide what order in which to ask questions so as to minimize entropy, or, as explained later, maximize information gain.

Cross entropy again

Suppose you have a *model* for the probability of discrete events, call this p_k^{M} where the index k just means one of K possibilities. The sum of these probabilities must obviously be one. And suppose that you have some empirical data for the probabilities of those events, p_k^{E}. With the sum again being one. The cross entropy is defined as

$$-\sum_k p_k^{\mathsf{E}} \ln\left(p_k^{\mathsf{M}}\right).$$

It is a measure of how far apart the two distributions are.

Let me give you an example.

Suppose that you have a machine-learning algorithm that is meant to tell you whether a fruit is a passion fruit, orange or guava. As a test you input the features for an orange. And the algorithm is going to output three numbers, perhaps thanks to the softmax function, which can be interpreted as the probabilities of the fruit in question (the orange) being one of P, O or G. Will it correctly identify it as an orange?

The model probabilities come out of the algorithm as

$$p_{\mathsf{P}}^{\mathsf{M}} = 0.13, \;\; p_{\mathsf{O}}^{\mathsf{M}} = 0.69 \;\text{ and }\; p_{\mathsf{G}}^{\mathsf{M}} = 0.18.$$

Ok, it's done quite well. It thinks the fruit is most likely to be an orange. But it wasn't 100% sure. Empirically we know that

$$p_{\mathsf{P}}^{\mathsf{E}} = 0, \;\; p_{\mathsf{O}}^{\mathsf{E}} = 1 \;\text{ and }\; p_{\mathsf{E}}^{\mathsf{M}} = 0,$$

because it definitely *is* an orange. The cross entropy is thus

$$-\left(0 \times \ln(0.13) + 1 \times \ln(0.69) + 0 \times \ln(0.18)\right) = 0.371.$$

The cross entropy is minimized when the model probabilities are the same as the empirical probabilities. To see this we can use Lagrange multipliers. Write

$$L = -\sum_k p_k^{\mathsf{E}} \ln\left(p_k^{\mathsf{M}}\right) - \lambda\left(\sum_k p_k^{\mathsf{M}} - 1\right). \tag{2.9}$$

The second term on the right is needed because the sum of the model probabilities is constrained to be one. Now differentiate (2.9) with respect to each model probability, and set the results to zero:

$$\frac{\partial L}{\partial p_k^{\mathsf{M}}} = -\frac{p_k^{\mathsf{E}}}{p_k^{\mathsf{M}}} - \lambda = 0.$$

But since the sums of the two probabilities must be one we find that $\lambda = -1$ and

$$p_k^{\mathsf{M}} = p_k^{\mathsf{E}}.$$

Because the cross entropy is minimized when the model probabilities are the same as the empirical probabilities we can see that cross entropy is a candidate for a useful cost function when you have a classification problem.

If you take another look at the sections on MLE and on cost functions, and compare with the above on entropy you'll find a great deal of overlap and similarities in the mathematics. The same ideas keep coming back in different guises and with different justifications and uses.

2.15 Natural Language Processing

Some of the methods we will look at are often used for Natural Language Processing (NLP). NLP is about how an algorithm can understand, interpret, respond to or classify text or speech.

NLP is used for

- **Text/sentiment classification:** Spam emails versus non spam is the example often given. And is a movie review positive or negative?
- **Answering questions:** Determining what a question is about and then searching for an answer
- **Speech recognition:** "Tell us in a few words why you are contacting BT today"

We will later give a couple of examples.

Text is obviously different from numeric data and this brings with it some special problems. Text also contains nuances that a native speaker might understand intuitively but which are difficult for an algorithm to appreciate. No kidding.

Here are a few of the pre-processing techniques we might need to employ and issues we need to deal with when we have text.

- **Tokenize:** Break up text into words plus punctuation (commas, full stops, exclamation marks, question marks, ellipses, ...)
- **Stop words:** Some words such as a, the, and, etc. might not add any information and can be safely removed from the raw text
- **Stemming:** One often reduces words to their root form, i.e. cutting off their endings. Meaning might not be lost and analytical accuracy might be improved. For example, all of love, loves, loving, loved, lover, ... would become lov

- **Lemmatize:** Lemmatizing is a more sophisticated version of stemming. It takes into account irregularity and context. So that eat, eating, ate, ... are treated as all being one word

- **Categorize:** Identify words as nouns, verbs, adjectives, etc.

- **'N'-grams:** Group together multiple words into phrases such as "red flag," "jump the shark," etc.

- **Multiple meanings:** Sometimes one needs to look at surrounding words to get the correct meaning. "The *will* of the people" or "I *will* have pizza for lunch" or "In his *will* he left all his money to his cat." Some words might be spelled the same, and have the same root, but be subtly different, for example, read and read

Word2vec

Word2vec are methods that assign vectors to words. This requires a large number of dimensions to capture a decent vocabulary. Words that have something in common end up, after training, near each other. Rather cleverly, combinations of words can be represented by simple vector operations, such as addition and subtraction. For example, fish + chips + mushy peas might result in a vector close to the dinner vector.

2.16 Bayes Theorem

Bayes Theorem relates probabilities of events given important information about the events. In symbols we have

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

where $P(A|B)$ is the probability of A being true/having happened given that B is true/happened. But $P(A)$ is the probability of A being true/having happened without any information/independently of B. Thus $P(A|B)$ is a *conditional* probability.

It's easy to see how this works in the following sketch, Figure 2.23.

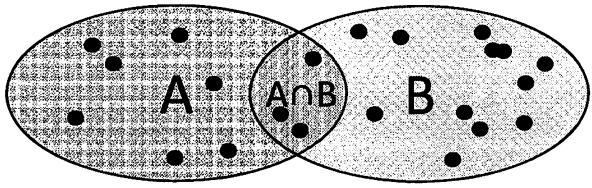


Figure 2.23: Demonstration of Bayes Theorem.

There are two ellipses here, with an overlapping area, and dots inside them. Imagine that the left ellipse is red, the right blue, and the overlapping area is thus purple. There are a total of 22 dots, ten in Ellipse A, 15 in Ellipse B, and thus three in both ellipses, in the purple area.

Assuming a dot is chosen at random, the probability that a dot is in Ellipse A is

$$P(A) = \frac{10}{22}.$$

The probability of being in B is

$$P(B) = \frac{15}{22}.$$

The probability that a dot is in *both* A and B, called the intersection of A and B, i.e. in the purple region, is written as

$$P(A \cap B) = \frac{3}{22}.$$

The conditional probability of A given B is

$$P(A|B) = \frac{3}{15}.$$

That is, out of the 15 in B there are three that are also in A. This is equal to

$$\frac{3/22}{15/22} = \frac{P(A \cap B)}{P(B)}.$$

That's a simple example of

$$P(A|B) = \frac{P(A \cap B)}{P(B)}.$$

But symmetrically

$$P(B|A) = \frac{P(B \cap A)}{P(A)}.$$

It's obvious that $P(B \cap A) = P(A \cap B)$ and so

$$\frac{P(A|B)}{P(A)} = \frac{P(B|A)}{P(B)}.$$

This is just another, symmetrical, way of writing Bayes Theorem.

2.17 What Follows

Having set the scene I am now going to walk you through some of the most important machine-learning techniques, method by method, chapter by chapter. I have chosen to present them in order of increasing difficulty. The first methods can be done in a simple spreadsheet, as long as your datasets aren't too large. As the methods get more difficult to implement then you might have to move on to proper programming for anything other than trivial examples. Although I am not going to cover programming in this book there are very many books out there which do this job wonderfully. In particular you will find many books on machine learning using the programming language *du jour*, Python.

All of the methods have worked illustrative examples. And they use real data, not made up. Having said that, none of the results, such as they are, come with any warranty or guarantee. Despite using real data the examples are illustrative. Sometimes I've violated good-practice guidelines, for example not worrying about the curse of dimensionality, in pursuit of a good story. In some cases the end results seem encouraging, perhaps even useful. Please also take those with a pinch of salt. Equally some of the results are perhaps unexpected. I have left those as a lesson in how machine learning can throw up the unexpected. Whether the unexpected results are meaningful though would require further analysis.

My primary goal is to get you up and running with some machine learning as soon as possible.

As the Ramones would say, "Hey, ho, let's go!"

Further Reading

Most of these topics are covered in depth in many decent books on numerical analysis and statistics/probability, but not all in one book as far as I know.

I would recommend that you take a look at Claude Shannon's classic paper introducing information theory from 1948 "A Mathematical Theory of

Communication," *The Bell System Technical Journal*, Vol. 27, pp. 379–423, 623–656, July, October, available online.

Chapter 3

K Nearest Neighbours

3.1 Executive Summary

K nearest neighbours is a supervised-learning technique. We take classified data represented by a vector of features. Think of coloured dots in space, the dimension of that space being the number of features and the colours being the categories. We want to classify a new dot, i.e. figure out its colour, given where it is in space.

We measure distances between the new data point and the nearest K of our already-classified data and thus conclude to which class our new data point belongs by majority voting.

Instead of having classes we can associate a number with each data point and so use the method for regression.

3.2 What Is It Used For?

K nearest neighbours is used for

- Classifying samples according to their numerical features
- Performing a regression of numerical values based on the features of the sample
- Example: Predict whether a spouse will like a Christmas present based on price, size, — of the present, not of the spouse — Amazon rating, etc.
- Example: Forecast the sales of a mathematics text book based on features such as price, length, number of university courses in the area, etc.

- Example: Credit scoring, classify as good or bad risk, or on a scale, based on features such as income, value of assets, etc.

- Example: Predict movie rating (number of stars) based on features such as amount of action sequences, quantity of romance, budget, etc.

- Example: Classify risk of epilepsy by analysing EEG signals

3.3 How It Works

K nearest neighbours (KNN) is perhaps the easiest machine-learning technique to grasp conceptually. Although really there is no learning at all.

It can be used for classification, determining into which group an individual belongs. Or it can be used for regression, predicting for a new individual a numerical value based on the values for similar individuals.

We start off with N points of already-classified data, making this a supervised-learning technique. Each data point has M features, so each point is represented by a vector with M entries. Each entry represents a feature. The first entry in the vector might be the number of words in an email, the second entry might be the number of exclamation marks, the third might be number of spelling mistakes, and so. And then each vector is classified as spam or not spam. The number of classes does not have to be two.

When we have a new data point to classify we simply look at the K original data points nearest to our new point and those tell us how to classify our new point.

In Figure 3.1 we see $N = 20$ data points with $M = 2$ features (represented by the two axes) classified into two groups, circles and triangles. Note that there could be any number of features/dimensions, I have two here just so I can easily draw the pictures (you'll be reading that a lot). And there could be any number of groups.

We add a new point, for the moment represented by a square in the figure. Should that new point be classified as a circle or a triangle?

Suppose we decide that $K = 3$, that we will look at the three nearest neighbours, then from Figure 3.2 we see that two of the nearest neighbours are circles and only one is a triangle. Thus we classify our square as being a circle (if you see what I mean). This is simple majority voting.

Of course, this example is only in two dimensions, so we can draw the pictures. The two dimensions just means that each data point has two features. In practice there might be many more features and thus we'd be working with hyperspheres to determine the nearest neighbours.

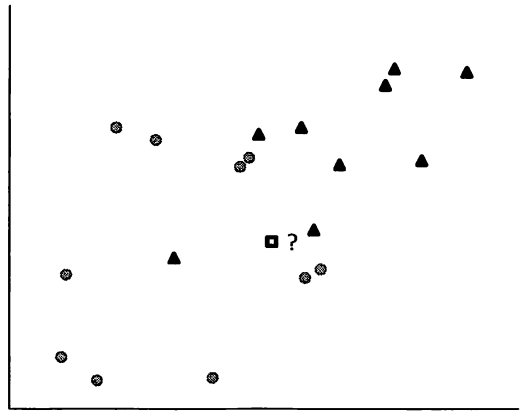


Figure 3.1: Does the square belong with the circles or the triangles?

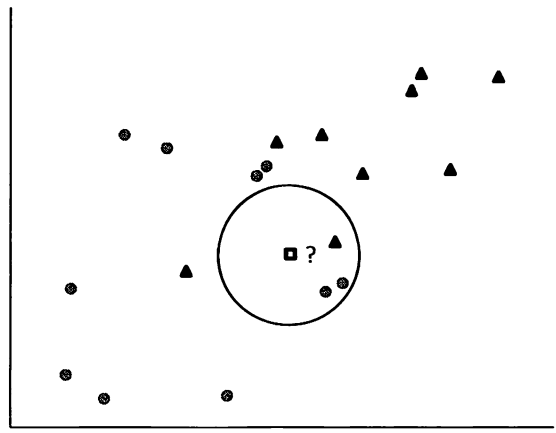


Figure 3.2: Finding the K nearest neighbours.

Choosing K

A big question is how to choose K. If it is small then you will get low bias but high variance (see Chapter 2). A large K gives the opposite, high bias but low variance.

3.4 The Algorithm

The idea is almost too simple to put into mathematics!

The K Nearest Neighbours Algorithm

Step 0: Scaling

As mentioned in Chapter 2, the first thing we do is to scale all features so that they are of comparable sizes. Perhaps scale so that the mean number of words in an email is zero with a standard deviation of one. Do this for all features.

Step 1: A new data point

We now take our new, unclassified data point, and measure the distance from it to all of the other data points. We look at the K points with the smallest distance. From those K points the classification with the most number of appearances determines the classification of our new point.

There is no real learning, as such, done in KNN. An algorithm in which a model is generalized only as new data points are added is called lazy. The opposite, in which an algorithm is trained before being used on new samples, is called eager. Lazy methods are useful when there is new data appearing all the time. In such a situation eager models would quickly become outdated.

3.5 Problems With KNN

Computer issues In this book I rarely talk about hardcore computer issues like memory and calculation speed. But I will make an observation here about the K nearest neighbours method. Although the non-existent learning part of the algorithm is lightning fast(!) the classification of new data points can be quite slow. You have to measure the distance between the new point and all of the already-classified points.

Skewed data If there are many more data points of one class than others then the data is skewed. In the above figures that might mean many more circles than triangles. The circles would tend to dominate by sheer numbers.

It is therefore common to weight the number of K nearest data points by either the inverse of the distance from our unclassified point or by a decaying exponential of the distance.

3.6 Example: Heights and weights

You can easily find data for men's and women's heights and weights online. In Figure 3.3 I show sample data, just 500 points for each gender. Note that the data has been shifted and scaled so as to have a mean of zero and a standard deviation of one for each of height and weight. Obviously I've used the same transformation for both genders. Because of the high degree of correlation I could have done a simple PCA which would have had the effect of spreading out the data. I haven't done that here for two reasons. First, because I want to focus on the KNN method. And second, because in higher-dimensional problems it might not be so obvious that there are any patterns. I've also sketched a couple of ellipses to help you see roughly where the different data points are.

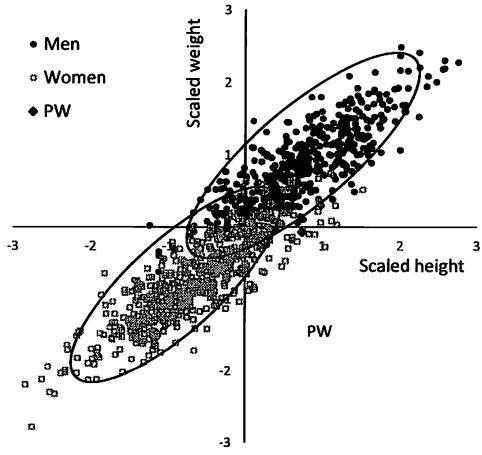


Figure 3.3: Heights and weights for 500 men, 500 women, and one PW, all scaled.

We can use KNN on a new sample to decide whether they are male or female. The new sample is the diamond in the plot (it's me!). You can see that I'm probably female, at least for small K. Maybe it's time to stop the diet.

When $K = 1$ the male/female regions are as shown in Figure 3.4. You can see several islands, pockets of males within what looks like an otherwise female zone and vice versa.

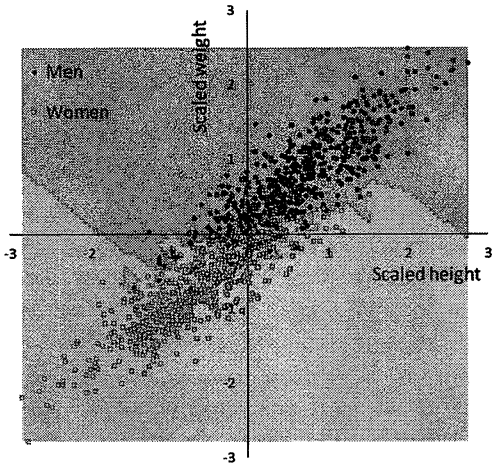


Figure 3.4: Male and female regions when $K = 1$.

When $K = 21$ the boundary has been smoothed out and there is just a single solitary island left. See Figure 3.5.

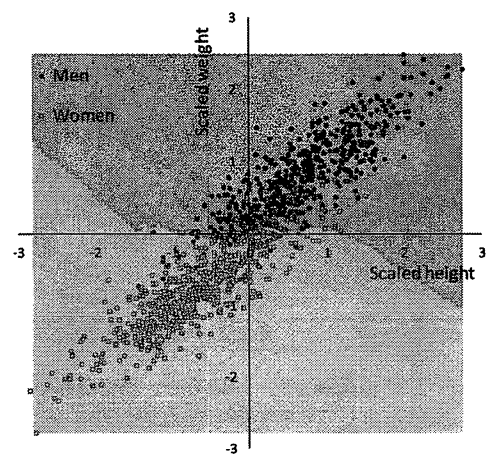


Figure 3.5: Male and female regions when $K = 21$.

As an extreme case take $K = 101$. See Figure 3.6.

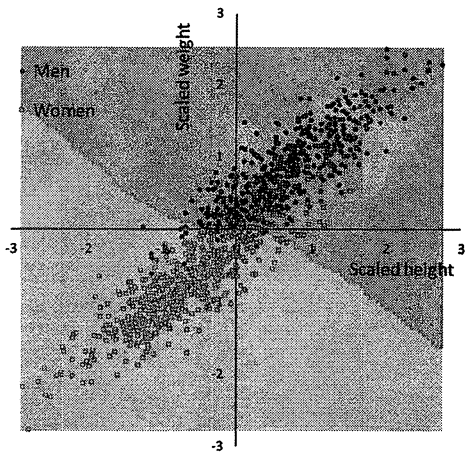


Figure 3.6: Male and female regions when $K = 101$.

Figures 3.4 and 3.6 nicely show the differences between bias and variance. The former figure shows high variance but low bias. It's a complicated model because it only looks at the nearest neighbour. Yes, I know that sounds like it's a simpler model but it's really not. The model changes a lot as we move around because the nearest neighbours change a lot. The latter figure shows high bias and low variance. It's a simpler model because the nearest neighbours don't change so much as we look at new samples. It has a high bias and low variance. In the extreme case in which we use the same number of neighbours as data points then every prediction would be the same.

We can plot misspecification error as a function of the number of neighbours, K. I have done this in Figure 3.7 using out-of-sample data. It looks like $K = 5$ is optimal.

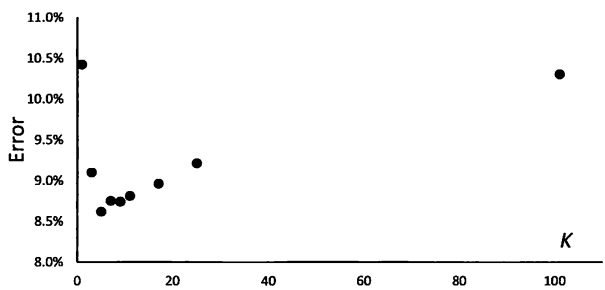


Figure 3.7: Error versus K.

We can also get a probabilistic interpretation of the classification. In Figure 3.8 is shown the probability of being male. This uses $K = 5$ and the obvious calculation.

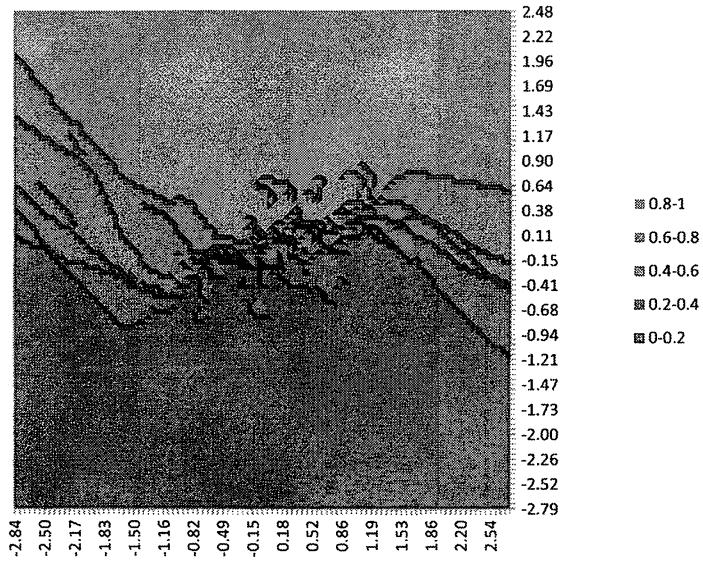


Figure 3.8: Probability of being male.

3.7 Regression

I've explained how KNN can be used for classification but it can also easily be used for regression.

The idea is simple. Instead of samples being labelled with classes, such as type of fruit, the labels are numbers. The features might be age, IQ, and height with the labels being salary. Your goal is to find the salary for a new sample. You tell me your age, IQ and height and I'll tell you how much you earn.

This regression can be done in a variety of ways.

- The simplest method is to just average the numbers (salaries) of the nearest K neighbours
- You could get more sophisticated by weighting the average, perhaps weighted by the inverse of the distances to the new data point. This

inverse weighting might be a bit too extreme, if you have a new data point that is *exactly* the same as one in your training set then it will give *exactly* the same label, no room for perhaps a confidence interpretation.

- Or exponentially weighted by distance
- Or using a Gaussian function of the distance, a Gaussian kernel
- Or by fitting a hyperplane to the K data points. This is a local linear regression. Figure 3.9 shows an example, with one independent variable for ease of plotting. Here $K = 5$. The bold dots are the nearest five points to our new sample. The faint dots represent the ones we are ignoring in the linear regression

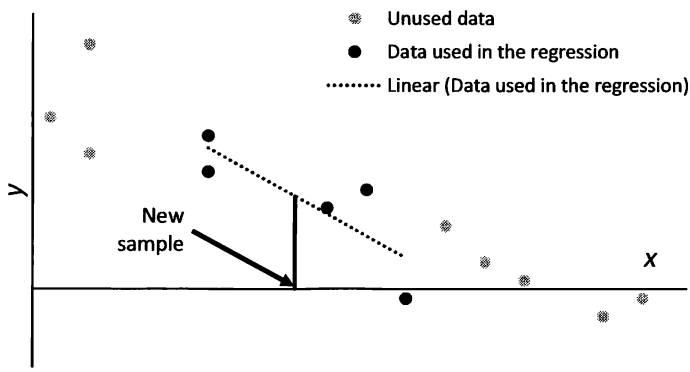


Figure 3.9: Linear regression on $K = 5$ nearest neighbours.

The weighting might require another parameter, as well as the K, such as the rate of decay in the exponential examples.

These methods generally go by the name of kernel smoothing.

Oh, and don't forget that the weights must add up to one!

Further Reading

There are not many books devoted to KNN in general, perhaps because it is so straightforward. Here are a couple you might want to look at.

Algorithms for Data Science Hardcover by Brian Steele, John Chandler and Swarna Reddy, published by Springer in 2017, covers the method, algorithms and also has exercises and solutions.

If you want to read something with depth then you should look at *Lectures on the Nearest Neighbor Method* (Springer Series in the Data Sciences) by Gérard Biau and Luc Devroye, published in 2015. This book covers the methods and provides a rigorous basis for them.

There are however many books and research papers that cover the application of KNN to *specific* problems.

Chapter 4

K Means Clustering

4.1 Executive Summary

K means clustering is an unsupervised-learning technique. We have many individual data points each represented by vectors. Each entry in the vector represents a feature. But these data points are not labelled or classified *a priori*.

Our goal is to group these data points in a sensible way. Each group will be associated with its centre of mass. But how many groups are there and where are their centres of mass?

4.2 What Is It Used For?

K means clustering is used for

- Grouping together unlabelled data points according to similarities in their features
- Example: Classification of customers according to their purchase history, each feature might be expenditure on different types of goods
- Example: Optimal placement of car parks in a city
- Example: Optimizing the neck size and arm length of shirts
- Example: Grouping together similar images, without them being previously classified

K means clustering (KMC) is a very simple method for assigning individual data points to a collection of groups or clusters. Which cluster each

data point is assigned to is governed simply by its distance from the centres of the clusters.

Since it's the machine that decides on the groups this is an example of unsupervised learning. KMC is a very popular clustering technique. It is highly intuitive and visual, and extremely easy to programme.

The technique can be used in a variety of situations:

- Looking for structure within datasets. You have unclassified data but you suspect that the data fall naturally into several categories.

  You might have data for car models, such as prices, fuel efficiency, wheel size, speaker wattage, etc. You find that there are two natural clusters, and they turn out to be cars that appeal to people with no taste and cars that appeal to everyone else. The data might look like in Figure 4.1. Here there are two obvious distinct groups.

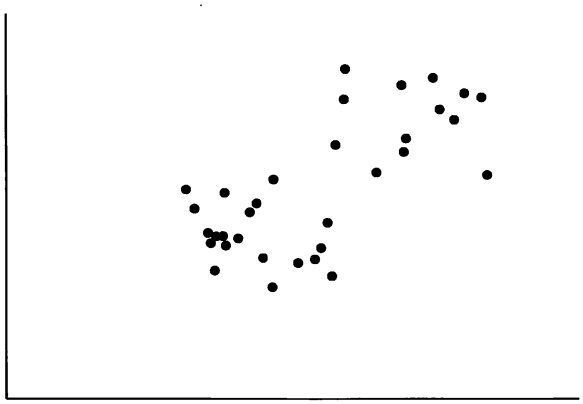


Figure 4.1: Two fairly distinct groups.

- Dividing up data artificially even if there is no obvious grouping.

  The data in Figure 4.2 might represent family income on the horizontal axis with the vertical axis being the number of people in the household. A manufacturer of cutlery wants to know how many knives and forks to put into his presentation boxes and how fancy they should be. There might not be obvious groups but that doesn't necessarily matter.

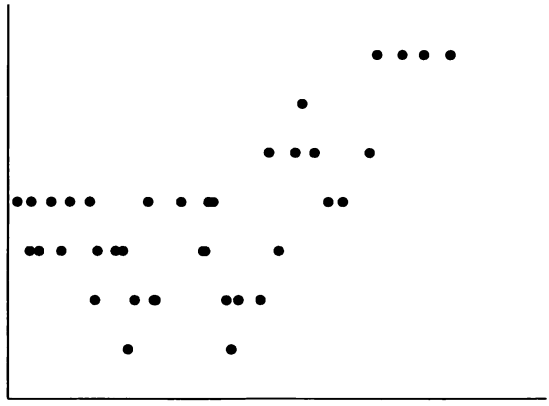


Figure 4.2: How would you group this data?

4.3 What Does K Means Clustering Do?

We have a dataset of N individuals, each having an associated M-dimensional vector representing M features, so $\mathbf{x}^{(n)}$ for $n = 1$ to N. Each entry in the vectors represents a different numerical quantity. For example each vector could represent an individual household, with the first element in each vector being an income, the second the number of cars, ..., the M^{th} being the number of guns.

We are going to group these data points into K clusters.

We will pick a number for K, say three. So we will have three clusters. Each of these three clusters will have a centre, a point in the M-dimensional feature space. And each of the N individual data points is associated with the centre that is closest to it. (Like houses and postboxes. The postbox is like the centre for the cluster. And each house is associated with one postbox.) The goal of the method is to find the best positions for the centres of these clusters. (And so you could use KMC to tell you where is best to put the postboxes.)

In Figure 4.1 there are clearly two distinct groups. If I asked you to put dots in the picture representing the centres of the two groups where would you put them? In this example it would be quite easy to do a reasonable job by hand. But in more than two or three dimensions? Not so easy. We need the machine to do it for us.

Mathematically, the idea is to minimize the intra-cluster (or within-cluster) variance. And the intra-cluster variance is just a measure of how far each individual point is to its nearest centre. (How far is the house from the nearest postbox.)

Typically you might then choose a different K and see what effect that has on distances.

The algorithm is really simple. It involves first guessing the centres and then iterating until convergence.

The K Means Algorithm

Step 0: Scaling

As explained in Chapter 2 we first scale our data, since we are going to be measuring distances.
Now we start on the iteration part of the algorithm.

Step 1: Pick some centres

We need to seed the algorithm with centres for the K clusters. Either pick K of the N vectors to start with, or just generate K random vectors. In the latter case they should have the same size properties as the scaled datasets, either in terms of mean and standard deviation or minimum and maximum. Call these centres $\mathbf{c}^{(k)}$ for $k = 1$ to K. See the two diamonds in Figure 4.3.

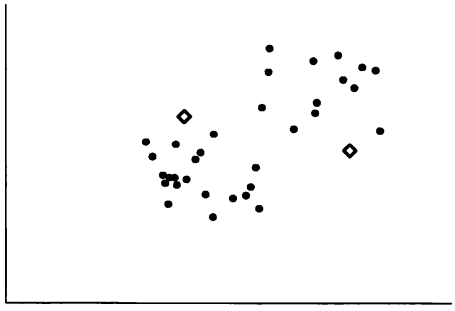


Figure 4.3: First pick some centres.

Step 2: Find distances of each data point to the centres

Now for each data point (vector $\mathbf{x}^{(n)}$) measure its distance from the centres of each of the K clusters. We've discussed measuring distances in Chapter 2. The measure we use might be problem dependent. If we have the house/postbox problem then we'd probably use the Manhattan distance (unless you expect people to walk through each other's back gardens). But often we'd use the obvious Euclidean distance:

$$\text{Distance}^{(n,k)} = \sqrt{\sum_{m=1}^{M} \left(x_j^{(n)} - c_j^{(k)}\right)^2} \quad \text{for } k = 1 \text{ to } K.$$

Each data point, that is each n, is then associated with the nearest cluster/centre:

$$\underset{k}{\operatorname{argmin}}\, \text{Distance}^{(n,k)}.$$

This is easily visualized as follows. Suppose K is two, there are thus two clusters and two corresponding centres. Let's call them the red cluster and the blue cluster. We take the first data point and measure its distance from each of the two centres. That's two distances. The smaller of these turns out to be the distance to the blue centre. So we paint our first data point blue. Repeat for all of the other data points, so each data point gets coloured. See Figure 4.4 in which I have drawn a line dividing the two groups of dots.

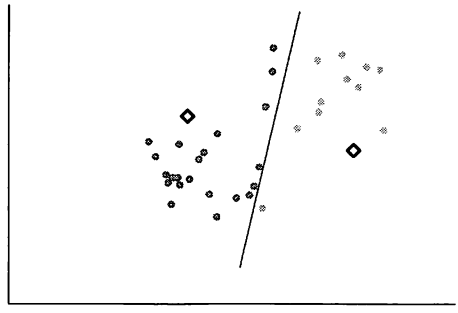


Figure 4.4: Assign each data point to its nearest center.

Step 3: Find the K centroids

Now take all of the data points associated with the first centre and calculate the centroid, its centre of mass. In the colourized version, just find the centroid of all the red dots. Do the same with all the blue dots. You will find K centroids. These will be the cluster centres for the next iteration.
Go back to Step 2 and repeat until convergence. See Figures 4.5 and 4.6.

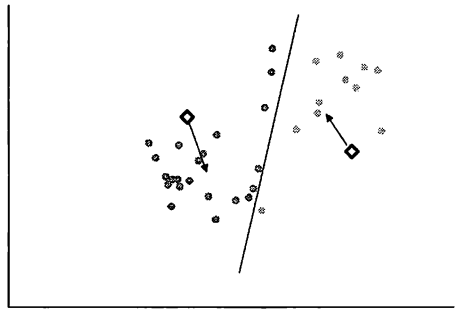


Figure 4.5: Update the centroid.

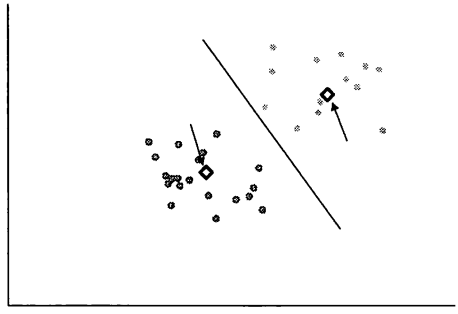


Figure 4.6: Repeat until converged.

4.4 Scree Plots

Adding up all the squared distances to the nearest cluster gives us a measure of total distance or an error. This error is a decreasing function of the number of clusters, K. In the extreme case $K = N$ you have one cluster for each data point and this error will be zero.

If you plot this error against K you will get a scree plot. It will be one of the two types of plots shown in Figure 4.7. If you get the plot with an elbow where the error falls dramatically and then levels off then you probably have data that falls into nice groupings (the triangles in Figure 4.7). In this example there is a big decrease in error going from $K = 2$ to $K = 3$, but there isn't so much of a drop for $K = 3$ to $K = 4$. The number of clusters is obvious; it is three in this plot.

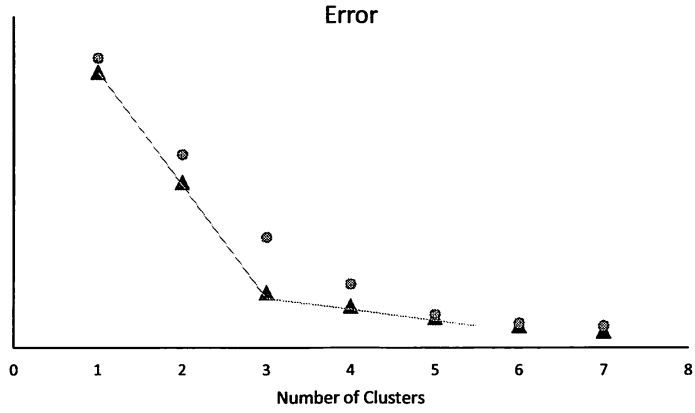


Figure 4.7: Two ways that error could decrease with number of clusters. The triangles are an example with an elbow.

If the error only falls gradually (the circles in Figure 4.7) then there is no obvious best K. Here we don't see large a large drop in error followed by a flattening out. That doesn't mean that there isn't a natural grouping of the data, for example you could have data like in Figure 4.8, but it will be harder to find that grouping.

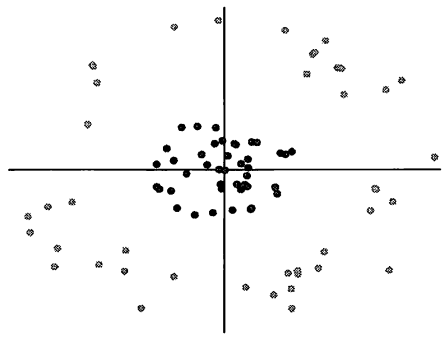


Figure 4.8: There is an obvious pattern here but it will take some work — a change of coordinate system, for example — to find it.

Note that while convergence is typically quite rapid, the algorithm might converge to a local minimum distance. So one would usually repeat several times with other initial centres.

4.5 Example: Crime in England, 13 dimensions

For our first real example I am going to take data for crime in England. The data I use is for a dozen different categories of crime in each of 342 local authorities, together with population and population density data. The population data is only for scaling the numbers of crimes, so we shall therefore be working in 13 dimensions. Don't expect many 13-dimensional pictures.

The raw data looks like in Figure 4.9. The full list of crimes is:

Burglary in a building other than a dwelling
Burglary in a dwelling
Criminal damage
Drug offences
Fraud and forgery
Offences against vehicles
Other offences
Other theft offences
Robbery
Sexual offences
Violence against the person - with injury
Violence against the person - without injury

Local Authority	Burglary in a building other than a dwelling	Burglary in a dwelling	Criminal damage	Drug offences	Fraud and forgery	Offences against vehicles	...	Population	Population per Square Mile
Adur	280	120	708	158	68	382	...	58500	3610
Allerdale	323	126	1356	392	79	394	...	96100	198
Alnwick	94	33	215	25	11	71	...	31400	75
Amber Valley	498	367	1296	241	195	716	...	116600	1140
Arun	590	299	1806	471	194	819	...	140800	1651
Ashfield	784	504	1977	352	157	823	...	107900	2543
Ashford	414	226	1144	196	162	608	...	99900	446
Aylesbury Vale	696	377	1490	502	315	833	...	157900	453
Babergh	398	179	991	137	152	448	...	79500	346
Barking & Dagenham	639	1622	2353	1071	1194	3038	...	155600	11862
Barnet	1342	3550	2665	1198	1504	4104	...	331500	9654
Barnsley	1332	860	3450	1220	322	1661	...	228100	1803
Barrow-in-Furness	190	134	1158	179	59	227	...	70400	2339
Basildon	756	1028	1906	680	281	1615	...	164400	3874
Basingstoke & Deane	1728	598	426	930	182	1159	...	147900	605

Figure 4.9: A sample of the crime data.

The crime numbers are first scaled with population in the local authorities. And then there is the second translation and scaling as discussed in Chapter 2.

A straightforward application of K-means clustering results in the following scree plot, shown in Figure 4.10.

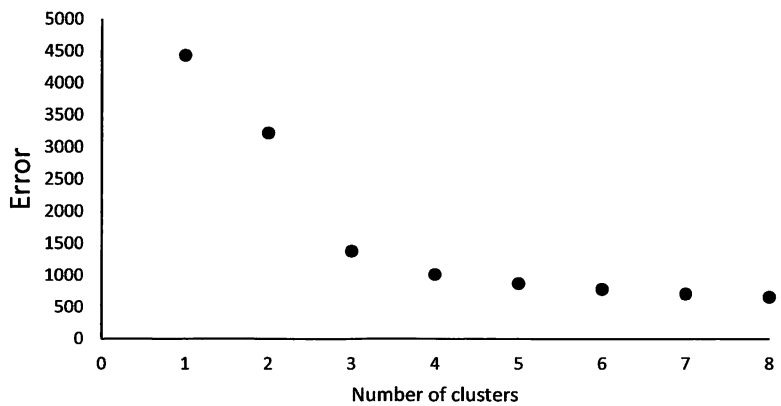


Figure 4.10: The scree plot for crime in England.

There is a moderately convincing elbow at around three clusters. And those three clusters are particularly interesting. The results are shown in the following table, Table, 4.1, in original variables but with the crime numbers scaled per head of population.

	Cluster 1	Cluster 2	Cluster 3
Number in cluster	1	68	273
Burglary in a building other than a dwelling	0.0433	0.0059	0.0046
Burglary in a dwelling	0.0077	0.0079	0.0030
Criminal damage	0.0398	0.0156	0.0114
Drug offences	0.1446	0.0070	0.0029
Fraud and forgery	0.1037	0.0042	0.0020
Offences against vehicles	0.0552	0.0125	0.0060
Other offences	0.0198	0.0018	0.0009
Other theft offences	0.6962	0.0313	0.0154
Robbery	0.0094	0.0033	0.0004
Sexual offences	0.0071	0.0015	0.0008
Violence against the person - with injury	0.0560	0.0098	0.0053
Violence against the person - without injury	0.0796	0.0128	0.0063
Population per Square Mile	4493	10952	1907

Table 4.1: Table of cluster results.

Cluster 1 has precisely one point. It is the City of London. In this data it appears as an outlier because the population figures for each local authority are people who *live* there. And not many people live in the City. We thus can't tell from this analysis how safe the City really is, since many of the crimes probably happen to non residents. For example, Burglary from a Dwelling is similar in both the City and in Cluster 2.

The other two clusters clearly represent dangerous local authorities (Cluster 2) and safer ones (Cluster 3). And what stands out to me, as someone who spends most of his time in the countryside, is that the safe places are the least densely populated. Phew.

This example also nicely illustrates an important point, specifically the effect of scaling. Although there is nothing wrong in having a cluster containing a small number of data points, here there could possibly be an issue.

Outliers can easily mess up the scaling. Having a large value for a feature in a small number of samples will usually cause that feature for the remaining samples to be rescaled to pretty small numbers. Although this will depend on the type of rescaling you use. And so when it comes to measuring distances this feature will not fully participate. In the present case what I should now do, if this were a research paper, is to remove the outlier, the City of London, and redo the calculations.

The above is quite a high-dimensional problem, 13 features meaning 13 dimensions, compared to relatively few, 342, training points. We might have expected to suffer from the curse of dimensionality mentioned in Chapter 2. From the results it looks like, luckily, we didn't. The reason for this might be the common one that the different features don't seem to be independent. Theft, robbery, burglary are very similar. Fraud and forgery, and sexual offences are not.

If you wanted to reduce the dimensions early on, before using K means, then you could use Principal Components Analysis. Or, as possibly here, don't use features that common sense says are very closely related.

We are now going to do a few financial/economic examples. There is an abundance of such data, for different financial or economic quantities and in vast amounts. Some of it you might have to pay for, for example the so-called tick data used in high frequency trading, but my examples all use the free stuff.

Warning I am going to be using time-series data below. Time-series data is not necessarily suitable for K means clustering analysis because the time component is lost, or rather it is not utilized. Nevertheless in the following examples we shall find some interesting results.

4.6 Example: Volatility

Of course, we can do problems in any number of dimensions. So let's start our financial examples with a one-dimensional example.

This problem, like several in this book, uses financial data. Financial data is particularly easy to get hold of. For example, take a look at `finance.yahoo.com` for many share and index prices. I'm going to be using the S&P500 index now. I first downloaded the index historical time series going back to 1950. From this I calculated in Excel a 30-day volatility. Now if you are from finance you'll know what I mean. But if you aren't then this is not the place to go into too many details. (I can give you the names of a few excellent technical books.) Suffice to say from the column of SP500 daily closing prices you calculate a column of daily returns, and then calculate the standard deviation of a window of these returns. Volatility is then just a scaled version of this. In Figure 4.11 is a plot of that volatility.

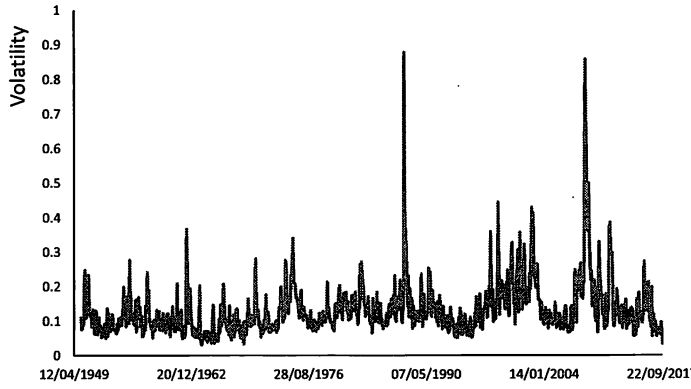


Figure 4.11: Thirty-day SPX volatility.

Remember that this KMC analysis completely throws away any time dependence in the behaviour of volatility. However I do have some motivation for using K means on this data and that is that there is a model used in finance in which volatility jumps from one level to another, from regime to regime. And the volatility in this plot seems to behave a bit like this. You see that often the volatility is pretty low, sometimes kind of middling, and occasionally very large.

That's not exactly what we have here. Here there is a continuum of volatility levels. But I'm not going to worry about that. I'm going to work with three clusters, $K = 3$. Remember this is just an illustrative example. Let's see how it goes. I will find those three clusters and then take this model a little bit further.

The algorithm very quickly finds the following clusters, see Table 4.2.

	Cluster 1	**Cluster 2**	**Cluster 3**
Number in cluster	586	246	24
SPX volatility	9.5%	18.8%	44.3%

Table 4.2: Volatility clusters in the S&P500 Index.

How volatility moves from cluster to cluster is shown in Figure 4.12.

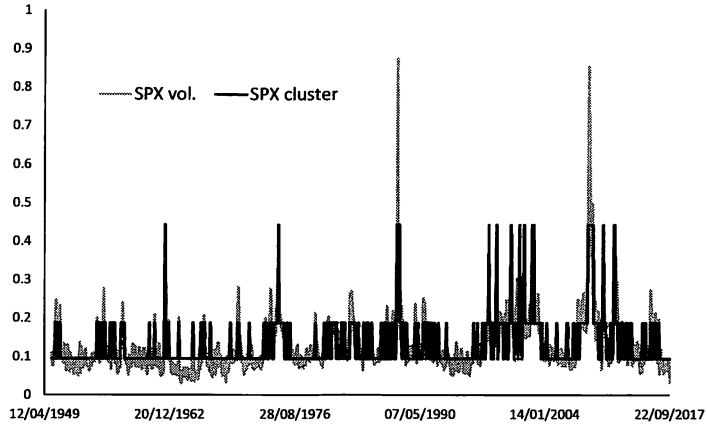


Figure 4.12: The original volatility and its clusters.

We can take this a step further and use it to give an idea of the likelihood of jumping from one volatility regime to another. And this is where I cunningly bring back a very simple, weak time dependence.

Rather easily one finds the following matrix of probabilities, Table 4.3.

	To:		
From:	**Cluster 1**	**Cluster 2**	**Cluster 3**
Cluster 1	84%	16%	0%
Cluster 2	38%	57%	5%
Cluster 3	0%	54%	46%

Table 4.3: Transition probabilities for a jump volatility model.

We interpret this as, for example, the probability of jumping from Cluster 1 to Cluster 2 is 16% every 30 days.

4.7 Example: Interest rates and inflation

Continuing with a financial theme, or rather economic, let's look at base rates and inflation. This is motivated by how central banks supposedly use base rates to control inflation. But again, this example is for illustrative purposes. (I will eventually stop saying this.) There will certainly be an important time component to these two variables, but I'm going to ignore it

in this KMC analysis.

I found data, shown in Figure 4.13, for United Kingdom interest rates and inflation going back to 1751.

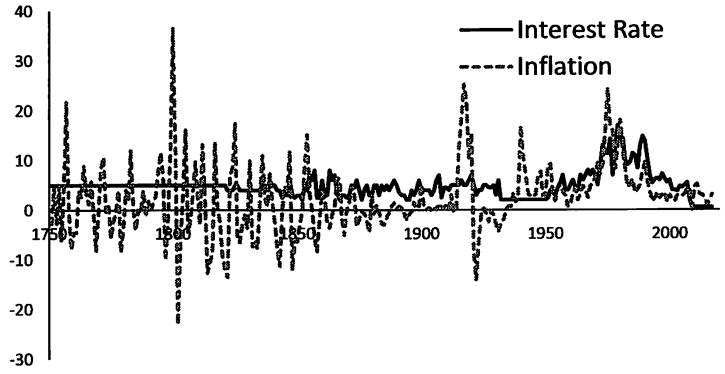


Figure 4.13: Inflation and interest rates going back to 1751.

It seems unlikely that something simple like K means clustering will do a good job with all of this data so I restricted myself to looking at data since 1945. In Figure 4.14 I have plotted inflation against the interest rate. I have also joined the dots to show how the data evolves, but, of course, this is not modelled by the simple K means algorithm.

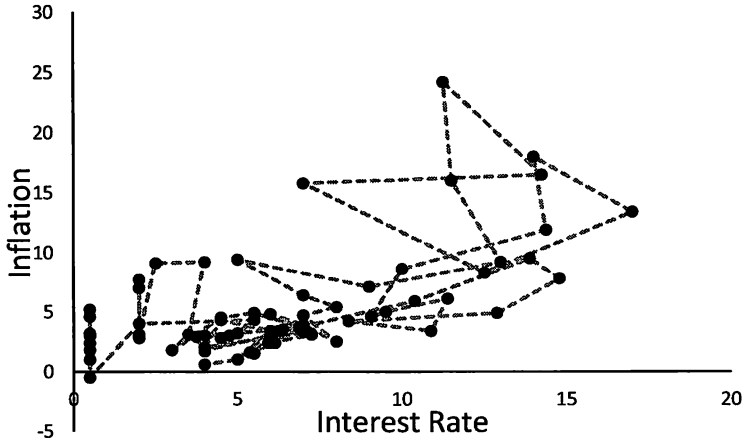


Figure 4.14: Inflation versus interest rates, joined chronologically.

Even though the interest rate and inflation numbers are ballpark the same order of magnitude I still translated and scaled them first.

With four clusters we find the results in Table 4.4.

	Cluster 1	Cluster 2	Cluster 3	Cluster 4
No. in cluster	25	30	11	7
Description	V. Low Base Rate, Normal Inflation	'Normal Economy' 'Normal Economy'	High Base Rate, Medium Inflation	High Base Rate, High Inflation
Interest rate	2.05%	6.15%	11.65%	12.77%
Inflation	3.21%	3.94%	6.89%	16.54%

Table 4.4: Clusters in interest rates and inflation. (In original scales.)

The centres of the clusters and the original data are shown in Figure 4.15. I have here plotted in the scaled quantities, and with the same length of axes, to make it easier to see (and draw) the lines dividing the nearest cluster.

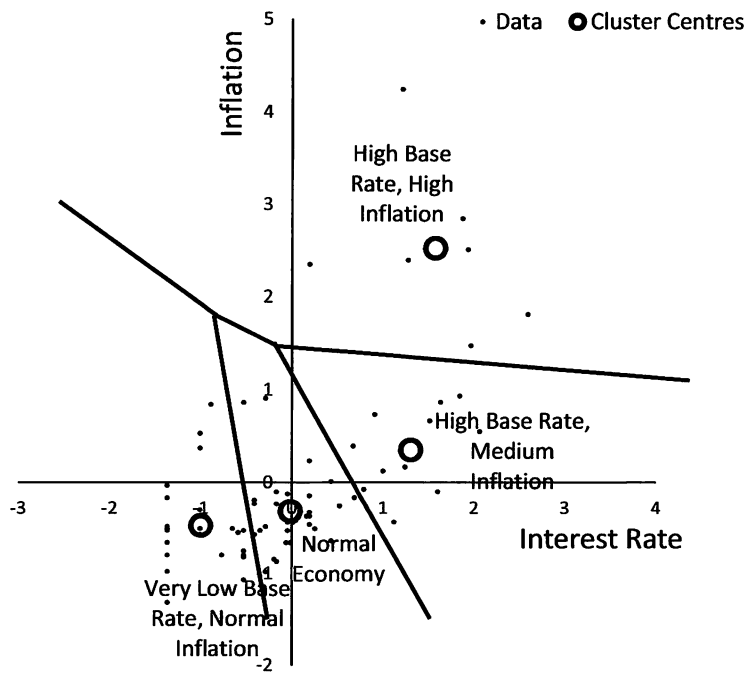


Figure 4.15: Inflation versus interest rate showing the four clusters. (Scaled quantities.) This is a Voronoi diagram (see text).

How the cluster evolves through time is shown in Figure 4.16.

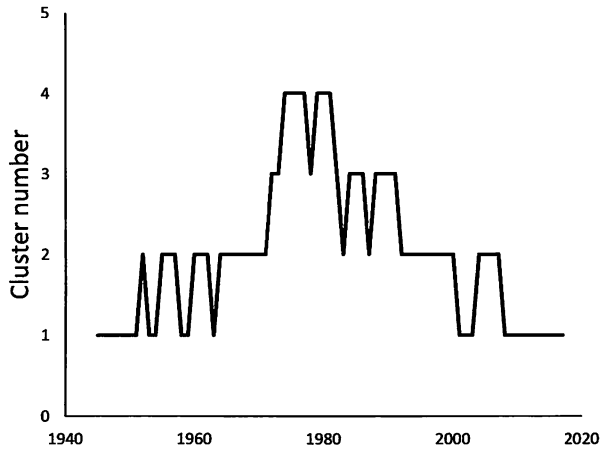


Figure 4.16: Evolution of the cluster number.

Again we can look at the probability of jumping from one cluster to another. The results are given in Table 4.5.

	To:			
From:	**Cluster 1**	**Cluster 2**	**Cluster 3**	**Cluster 4**
Cluster 1	88.9%	11.1%	0.0%	0.0%
Cluster 2	5.6%	86.1%	8.3%	0.0%
Cluster 3	0.0%	25.0%	58.3%	16.7%
Cluster 4	0.0%	0.0%	33.3%	66.7%

Table 4.5: Transition probabilities from cluster to cluster.

An interesting experiment that you can do is to shuffle the data randomly. In this example it means keeping the interest rate data, say, fixed and reorder the inflation data randomly. If there was any structure to the data initially then it will probably be lost during this shuffling.

4.8 Example: Interest rates, inflation and GDP growth

Take the data in the previous example and add to it data for GDP growth. We now have a three-dimensional problem. (And now I've used quarterly data for this.)

With six clusters I found that three of the clusters that I had earlier

did not change much but the normal economy broke up into three separate clusters. See Figure 4.17 for the results plotted in two dimensions.

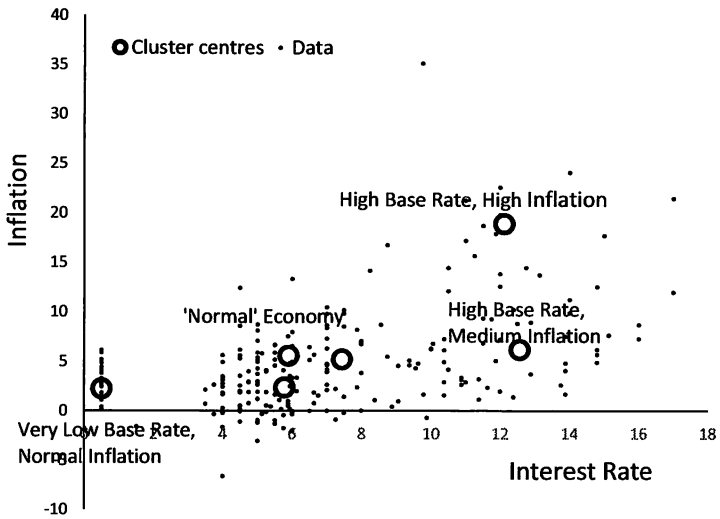


Figure 4.17: Inflation versus interest rate. (The GDP-growth axis would come out of the page. Original scaling.)

The clusters are in Table 4.6.

	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5	Cluster 6
Number in cluster	31	107	23	31	17	38
Interest rate	0.50%	5.77%	5.89%	7.44%	12.11%	12.56%
Inflation	2.24%	2.37%	5.59%	5.17%	18.89%	6.18%
GDP growth	0.48%	0.71%	-0.63%	2.25%	-0.51%	0.33%

Table 4.6: Clusters for interest rates, inflation and GDP growth.

4.9 A Few Comments

- **Voronoi diagrams:** Figure 4.15 is a Voronoi diagram. This is a partitioning of a space into regions depending on the distance, usually Euclidean but not necessarily, from a set of points. The use of Voronoi diagrams dates back to Descartes in the 17th century. An interesting use of them was by John Snow in 1854 to show how most people dying in a cholera outbreak were nearest to a particular water pump.

- **Choosing K:** If you are lucky then it will be clear from the error-versus-K plot what is the optimal number of clusters. Or perhaps the number will be obvious, something about the nature of the problem will be a clue. Not quite so convincing but it helps plausibility if you can give each cluster a name, like I did in the first interest rate example.

- There is a small tweak you must do to the distance measure if you are going to do one of examples I mentioned right at the start of this chapter. Specifically with reference to the neck sizes of shirts. Can you guess what it is? Clue: If you have a 17-inch neck you aren't going to be able to wear a 15-inch shirt.

Further Reading

For KMC implemented using the programming language R, see *Applied Unsupervised Learning with R: Uncover previously unknown patterns and hidden relationships with k-means clustering, heirarchical clustering and PCA* by Bradford Tuckfield, published by Packt in 2019.

There are many books in the "For Dummies" series that cover different aspects of machine learning. KMC is found in *Predictive Analytics For Dummies, 2nd Edition* by Anasse Bari, published by John Wiley & Sons in 2016.

If you want to read about modelling volatility as a jump process then go to `Ito33.fr` and download their technical publications.

I did promise I could suggest a few excellent books on quantitative finance if you want to learn about that subject. I can wholeheartedly recommend *Paul Wilmott Introduces Quantitative Finance*, *Paul Wilmott On Quantitative Finance* (in three amazing volumes) and *Frequently Asked Questions in Quantitative Finance*. All by Paul Wilmott and published by John Wiley & Sons.

Chapter 5

Naïve Bayes Classifier

5.1 Executive Summary

Naïve Bayes classifier is a supervised-learning technique. We have samples of data representative of different classes. Then using Bayes Theorem we calculate the probability of new data being in each class.

5.2 What Is It Used For?

Naïve Bayes classifier is used for

- Classifying data, often text, and analysing sentiment
- Example: Natural Language Processing (NLP), getting the computer to understand humans
- Example: Email spam detection
- Example: Classify whether a piece of news is good or bad
- Example: Predict in which direction tweets on twitter are going to influence an election or referendum
- Example: Determine if said tweets are from a Russian bot

Naïve Bayes Classifier (NBC) is sometimes explained using examples of book reviews. By analysing the statistics of a review and comparing with a catalogue of other reviews the algorithm learns which words appear in good reviews and which in bad reviews. It is another example of supervised learning since the training examples are already classified.

First we need a large collection of good reviews, and a large collection of bad reviews. We count up incidences of certain words appearing in each of these categories. And thus the probability of each word appearing in each class. Then if we are given a new review a straightforward use of Bayes Theorem will give us the probability that the new review is either good or bad.

Of course we are not restricted to just two classes, good or bad, nor does it have to be book reviews. As my example will show.

The reason that the method is called naïve is that the assumptions are usually quite unrealistic. The main assumption being that words are independent of each other, which is clearly not the case. However one always reads that this doesn't seem to matter all that much in practice.

5.3 Using Bayes Theorem

In our NBC example shortly we shall be applying Bayes Theorem to political speeches. An example of Bayes Theorem would be

$$P(\text{Politician is left wing}|\text{Uses the word "Comrade"}) = \frac{P(\text{Uses the word "Comrade"}|\text{Politician is left wing})P(\text{Politician is left wing})}{P(\text{Uses the word "Comrade"})}.$$

So if we hear a politician using the word "Comrade" — this won't be something my American readers will have experienced — then we could calculate the probability of him being left wing by studying the speeches of left-wing politicians, and knowing how often politicians generally use the word, and the probability of any random politician being left wing.

5.4 Application Of NBC

But we are going to apply NBC not just to a single word but to whole phrases and ultimately entire speeches. And also we don't calculate the exact probability of a politician being left wing. Instead we *compare* probabilities for being left wing and right wing. You will see when I go through the details in a moment.

Suppose we hear a politician saying "Property is theft, Comrade" and want to know whether he is left or right wing. We'd like to know

$$P(\text{Left}|\text{Says, "Property is theft, Comrade"}). \tag{5.1}$$

And similar for being right wing.

We use Bayes Theorem as follows. We first write (5.1) as

$$P(\text{Left}|\text{Says, "Property is theft, Comrade"}) =$$

$$\frac{P(\text{Says, "Property is theft, Comrade"}|\text{Left})P(\text{Left})}{P(\text{Says, "Property is theft, Comrade"})}.$$

And we'd write something similar for the probability of being right wing. We compare for Left and Right and the larger determines which way our politician swings. But note that we don't need to calculate the denominator in the above. The denominator does not depends on the classes, whether left or right. And so all we need to do is calculate the numerator for left and right and then compare, to see which is the larger.

We just have to estimate

$$P(\text{"Property is theft, Comrade"}|\text{Left})P(\text{Left})$$

and

$$P(\text{"Property is theft, Comrade"}|\text{Right})P(\text{Right}).$$

Now comes the naïve part. We shall assume that

$$P(\text{Says, "Property is theft, Comrade"}|\text{Left}) =$$

$$P(\text{Says, "Property"}|\text{Left}) \times P(\text{Says, "is"}|\text{Left}) \times$$
$$P(\text{Says, "theft"}|\text{Left}) \times P(\text{Says, "Comrade"}|\text{Left}).$$

That is, all the words are independent. (Which is clearly not true for most sentences.) From training data we would know how often politicians of different persuasions use the words "property," "theft," etc. (Not so much the "is." That's a stop word that we would drop.) Thus each of those probabilities, for both class of politician, is easily found from many speeches of many politicians.

Let's write all this in symbols.

5.5 In Symbols

The text, or whatever, we want to classify is going to be written as $\mathbf{x}$, a vector consisting of entries x_m for $1 \leq m \leq M$, so that there are M words in the text. We want to find

$$P(C_k|\mathbf{x}) \tag{5.2}$$

for each of the K classes (political persuasions) C_k. And whichever probability is largest gives us our decision (that's just maximum likelihood as mentioned in Chapter 2).

Bayes tells us that (5.2) can be written as

$$\frac{P(C_k)\ P(\mathbf{x}|C_k)}{P(\mathbf{x})}.$$

The denominator we can ignore because it is common to all classes (and we are only *comparing* the numbers for each class, the exact number doesn't matter).

If the features are all independent of each other then this simplifies — if they're not then this is much harder — and the numerator becomes

$$P(C_k) \prod_{m=1}^{M} P(x_m|C_k).$$

This is what we must compare for each class. The term $P(x_m|C_k)$ we will get from the data, just look at the speeches of other politicians and look at the probabilities of words, the x_ms, appearing and which political direction they lean.

Finally, because we are here multiplying potentially many small numbers we usually take the logarithm of this expression. This doesn't change which class gives the maximum likelihood just makes the numbers more manageable:

$$\ln(P(C_k)) + \sum_{m=1}^{M} \ln(P(x_m|C_k)).$$

In the terminology of Bayes Theorem the leading term, the $\ln(P(C_k))$, is known as the prior (probability). It is the probability estimated before taking into account any further information. Here we would typically get this probability from knowledge of the population as a whole, here the probability of a politician being in each class, or from our training set.

5.6 Example: Political speeches

I am going to work with the speeches or writings of some famous politicians or people with political connections. And then I will use this training set to classify another speech.

The usual caveats apply here. This is not research-level material so I have not gone to the lengths that I would if this were for an academic journal. On the other hand it is not data that I have made up just for this book to give nice results. This is very much warts and all.

I have taken eight speeches/writings. For example one is:

A spectre is haunting Europe-the spectre of Communism. All the Powers of old Europe have entered into a holy alliance to exorcise this spectre:

Pope and Czar, Metternich and Guizot, French Radicals and German police-spies...

This is immediately recognisable as the start of the *Communist Manifesto* by Marx and Engels.

I have also used the opening thousand or so words from each of

- Churchill: "Beaches" speech
- JFK: Inaugural address
- Benn: Maiden speech as MP
- Thatcher: "The Lady's not for turning" speech
- May: Syria strikes speech
- Corbyn: Post-Brexit-Referendum speech
- Trump: State of the Union speech

And I have classified them as either Left or Right.

Here's a selection of words used by the left and their probabilities and log probabilities:

Word	**Prob.**	**Log Prob.**
...		
ablaze	0.013%	-8.983
able	0.050%	-7.597
abolish	0.025%	-8.290
abolished	0.013%	-8.983
abolishing	0.019%	-8.578
abolition	0.088%	-7.037
about	0.094%	-6.968
above	0.019%	-8.578
abroad	0.019%	-8.578
absence	0.013%	-8.983
absolute	0.031%	-8.067
absolutely	0.013%	-8.983
...		

Table 5.1: Left-wing words.

And by the right:

Word	**Prob.**	**Log Prob.**
...		
add	0.029%	-8.130
addiction	0.015%	-8.824
additional	0.015%	-8.824
address	0.015%	-8.824
administration	0.081%	-7.119
admits	0.015%	-8.824
adopt	0.015%	-8.824
advance	0.015%	-8.824
advantage	0.022%	-8.418
advantages	0.015%	-8.824
adverse	0.015%	-8.824
advert	0.015%	-8.824
...		

Table 5.2: Right-wing words.

There are a couple of minor tweaks we have to do before trying to classify a new speech:

1. Remove all stop words from the texts. Stop words, mentioned in Chapter 2, are words like the, a, of, etc., words that don't add any information but could mess up the statistics. (There are some words that are sometimes stop words but sometimes aren't, like like. A speech with many likes in it is very, very easy to classify. As irritating.)

2. There will be words in our new to-be-classified speech that do not appear in any of our training data. In order for them not to give a log probability of minus infinity we give them a default log probability of zero. Although there is indubitably some commodious information in a speech that is rhetorically portentous.

And then I took the following speech:

happy join today down history greatest demonstration freedom history nation years great American symbolic shadow stand today signed...

You will have noticed that I have removed a lot of stop words. So it's not as immediately recognisable as it might be!

For a bit of fun I took the speech and imagined doing a real-time analysis of it, in the sort of way you might carefully listen to the report by a CEO

and try to read into his presentation whether the news is good or bad before quickly putting in your buy or sell order. And the results are shown in Figure 5.1. By 15 important words the speech has already set its tone. (Forty words of actual speech.)

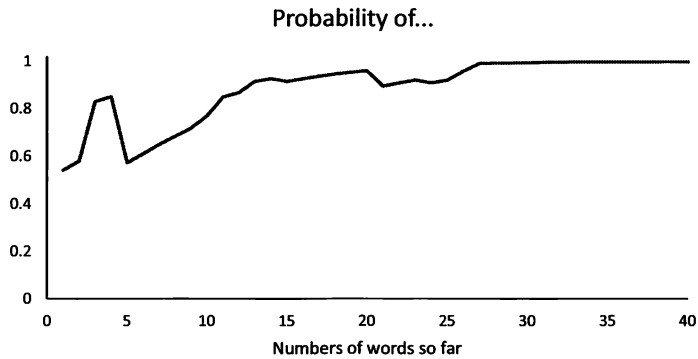


Figure 5.1: Running probability.

This is supposed to be the probability that the speaker is right wing. Can you guess who it is?

Well, the person is not exactly famous for being right wing. And he was not exactly a politician. But he is famous for giving good speech. It is, of course, the "I Have A Dream" speech by Martin Luther King Jr. What does this tell us? Maybe MLK was right wing. Maybe this analysis was nonsense, too little data, too little follow up. Or maybe the classifier has found something I wasn't looking for, it's lumped in MLK with Churchill and Thatcher, and not with Jeremy Corbyn, because of his powers of rhetoric.

There are many ways you can take this sort of analysis, but with much of ML (that's Machine Learning, not Martin Luther) you have to treat it with care and don't be too hasty. If this were anything but an illustrative example I would have used a lot more data, more politicians, more speeches and done a full testing and validation. However I shall leave it here as I am simply demonstrating the technique.

Further Reading

For a short and inexpensive overview of the subject of NLP see *Introduction to Natural Language Processing: Concepts and Fundamentals for Beginners* by Michael Walker, published in 2017.

Chapter 6

Regression Methods

6.1 Executive Summary

Regression methods are supervised-learning techniques. They try to explain a dependent variable in terms of independent variables. The independent variables are numerical and we fit straight lines, polynomials or other functions to predict the dependent variables. The method can also be used for classification, when the dependent variables are usually zeroes and ones.

6.2 What Is It Used For?

Regression methods are used for

- Finding numerical relationships between dependent and independent variables based on data
- Classifying data based on a set of numerical features
- Example: Find a relationship between the number of tomatoes on a plant, the ambient temperature and how much they are watered
- Example: Determine the probability of getting cancer given lifestyle choices (quantity of alcohol consumed, cigarettes smoked, how often you visit Amsterdam, etc.)

You will no doubt know about regression from fitting straight lines through a set of data points. For example, you have values and square footage for many individual houses, is there a simple linear relationship between these two quantities? Any relationship is unlikely to be perfectly linear so what is the *best* straight line to put through the points? Then you can move on

to higher dimensions. Is there a linear relationship between value and both square footage and number of garage bays?

As an introduction to this supervised-learning technique we shall start with looking for linear relationships in multiple dimensions, and then move on to logistic regression which is good for classification problems.

6.3 Linear Regression In Many Dimensions

We saw a brief summary of linear regression in one dimension in Chapter 2. Going over to multiple regression is mathematically simple. We now have M independent, explanatory, variables, representing the features, and so we write all xs as vectors. For each of N data points we will have the independent variable $\mathbf{x}^{(n)}$ (square footage of a house, number of garage bays, etc.) and the dependent variable $y^{(n)}$ (property value).

We will fit the linear function $h_\theta(\mathbf{x}) = \boldsymbol{\theta}^T\mathbf{x}$ to the ys, where $\boldsymbol{\theta}$ is the vector of the as-yet-unknown parameters.

One subtle point is that because we usually want to have a parameter θ_0 that doesn't multiply any of the independent variables we write $\mathbf{x}$ as $(1, x_1, \ldots, x_M)^T$, where T means transpose, so it has dimension $M+1$.

The cost function remains the same as in Chapter 2, i.e. the quadratic function:

$$J(\boldsymbol{\theta}) = \frac{1}{2N}\sum_{n=1}^{N}\left(h_\theta(\mathbf{x}^{(n)}) - y^{(n)}\right)^2.$$

As in Chapter 2 we can differentiate this cost function with respect to each of the θs, set the result equal to zero, and solve for the θs. Easy. However, although there is still technically an analytic expression for the vector $\boldsymbol{\theta}$ it involves matrix inversion so in practice you might as well use gradient descent.

Finding the parameters numerically using gradient descent

Although numerical methods are not needed when you have linear regression in a single dimension you will need to use some numerical method for anything more complicated. Batch gradient descent and stochastic gradient descent have both been mentioned in Chapter 2. To use these methods you will need

$$\frac{\partial J}{\partial \boldsymbol{\theta}} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}^{(n)}\left(h_\theta(\mathbf{x}^{(n)}) - y^{(n)}\right).$$

The Batch Gradient Descent Algorithm

Step 1: Iterate

New $\boldsymbol{\theta}$ = Old $\boldsymbol{\theta}$ $-\beta\, \partial J/\partial \boldsymbol{\theta}$
I.e. New $\boldsymbol{\theta}$ = Old $\boldsymbol{\theta}$ $-\beta/N \sum_{n=1}^{N} \mathbf{x}^{(n)} \left(h_\theta(\mathbf{x}^{(n)}) - y^{(n)}\right)$

Step 2: Update

Update all θ_k simultaneously. Repeat until convergence

Linear regression is such a common technique that I'm going to skip examples and go straight to something different, regression used for classification.

6.4 Logistic Regression

Suppose you want to classify an email as to whether or not it is spam. The independent variables, the xs, might be features such as number of !s or number of spelling mistakes in each email. And your ys will all be either 0, not spam, or 1, spam. Linear regression is not going to do a good job of fitting in this case. See Figure 6.1, would you want to fit a straight line through this data? It would be nonsense.

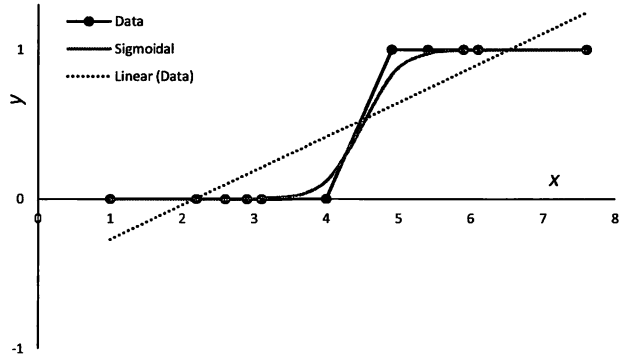


Figure 6.1: Regression for a classification problem.

One way to see why this is nonsense is to add another dot to this figure, a dot way off to the right, with $y = 1$. It is clearly not conflicting with the

data that's already there, if it had $y = 0$ then that might be an issue. But even though it seems to be correctly classified it would cause a linear fit to rotate, to level out and this would affect predictions everywhere.

Also sometimes the vertical axis represents a probability, the probability of being in one class or another. In that case any numbers that you get that are below zero or above 1, which you will get with a linear fit, will also be nonsense.

The conclusion is that for classification problems we need something better than linear. We tend to use a sigmoidal function, such as the logistic function,

$$h_\theta(x) = \frac{1}{1 + e^{-\theta_0 - \theta_1 x}},$$

or, since we are going to go immediately to multiple regression,

$$h_\theta(\mathbf{x}) = \frac{1}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}}}.$$

This function is shown in the figure. We would fit this function to the data and then given a new data point (email) we would determine whether it was spam or not according to a threshold for h_θ. We might have a threshold of 0.5, so that anything above that level would go into our spam box. Or we might use a different threshold. If we are worried that genuine emails are going into our spam box (we have friends who are very bad at speeling and use lots of exclamation marks) then the threshold might be 0.8, say.

The cost function

An important difference when we want to do logistic regression instead of linear regression is in the choice of cost function.

There are some obvious properties for a cost function: It should be positive except when we have a perfect fit, when it should be zero; It should have a single minimum. Our previous, quadratic, cost function satisfies the first of these but when h_θ is the logistic function it does not necessarily have a single minimum.

However, the following cost function does have these properties.

$$J(\boldsymbol{\theta}) = -\frac{1}{N} \sum_{n=1}^{N} \left(y^{(n)} \ln \left(h_\theta(\mathbf{x}^{(n)}) \right) + (1 - y^{(n)}) \ln \left(1 - h_\theta(\mathbf{x}^{(n)}) \right) \right). \tag{6.1}$$

So why does this cost function work? Remember that y can only take values 0 or 1. When $y = 1$ the function $J(\boldsymbol{\theta})$ is smoothly, monotonically decreasing to a value of zero when h_θ is also one. And when $y = 0$ the cost function is also zero when $h_\theta = 0$. But perhaps most importantly this cost

function has a nice interpretation in terms of maximum likelihood estimation for classification problems as explained in Chapter 2.

We no longer have an analytic solution for the minimum of the cost function so we have to solve numerically. But rather beautifully we find that with the new definition for h_θ and the new cost function we still have

$$\frac{\partial J}{\partial \boldsymbol{\theta}} = \frac{1}{N}\sum_{n=1}^{N} \mathbf{x}^{(n)} \left(h_\theta(\mathbf{x}^{(n)}) - y^{(n)} \right).$$

This means that our gradient descent algorithm remains, surprisingly, unchanged.

6.5 Example: Political speeches again

In the last chapter I used some data from the speeches and writings of politicians to determine the nature of an unseen speech. I'm not sure that it quite went according to plan, but was nevertheless a fascinating exercise, I thought. I'm going to use the same data, the same eight speeches, here but in a different way, using a regression method.

Again I shall take speeches/writings by $N = 8$ politicians. And I shall label each politician as either 0 for left wing or 1 for right wing. These are the $y^{(n)}$ for $n = 1, \ldots, N$. But instead of looking at the frequency of individual words as before I shall look at the *types* of words used. Are the words positive words, negative words, irregular verbs, etc.? For a total of M features. So the n^{th} politician is represented by $\mathbf{x}^{(n)}$, a vector of length $M + 1$. The first entry is 1. The second entry would be the fraction of positive words the n^{th} politician uses, the third entry the fraction of negative words, and so on.

But how do I know whether a word is positive, negative, etc.? For this I need a special type of dictionary used for such textual analysis.

One such dictionary, easily found online, is the Loughran–McDonald dictionary. This is a list of words classified according various categories. These categories are: Negative; Positive; Uncertainty; Litigious; Constraining; Superfluous; Interesting; Modal; Irregular Verb; Harvard IV; Syllables. This list is often used for financial reporting (hence the litigious category). Most of the category meanings are clear. Modal concerns degree from words such as "always" (strong modal, 1) through "can" (moderate, 2) to "might" (weak, 3). Harvard IV is a Psychosociological Dictionary.

The results of the analysis of the speeches is shown in Table 6.1. The fractions of positive, negative, etc. words is very small because the vast

majority of words in the dictionary I used are not positive, negative, etc.

	Benn	Churchill	Corbyn	JFK	M&Es	May	Thatcher	Trump
Negative	0.501%	1.048%	1.216%	0.562%	2.067%	0.395%	1.626%	1.261%
Positive	0.213%	0.380%	0.517%	0.243%	0.760%	0.137%	1.018%	1.048%
Uncertainty	0.395%	0.441%	0.274%	0.137%	0.289%	0.091%	0.425%	0.289%
Litigious	0.152%	0.061%	0.274%	0.213%	0.441%	0.091%	0.304%	0.228%
Constraining	0.000%	0.030%	0.137%	0.122%	0.304%	0.122%	0.046%	0.122%
Superfluous	0.000%	0.000%	0.000%	0.000%	0.030%	0.000%	0.000%	0.000%
Interesting	0.030%	0.061%	0.000%	0.000%	0.152%	0.030%	0.106%	0.061%
Modal	1.307%	1.975%	1.945%	0.881%	1.276%	0.380%	2.097%	2.112%
Irregular verb	0.228%	0.745%	0.608%	0.334%	0.988%	0.289%	0.942%	0.699%
Harvard IV	1.352%	3.206%	4.513%	1.869%	7.765%	1.596%	4.270%	4.255%
Syllables	1.52	1.44	1.57	1.41	1.68	1.68	1.50	1.49

Table 6.1: Results of the analysis of political speeches.

Because I was only using eight politicians for the training I did not use all 11 of these categories for the regression fitting. If I did so then I would probably get a meaningless perfect fit. (Twelve unknowns and eight equations.) So I limited the features to just positive, negative, irregular verbs and syllables.

I found that the θs for positive words, irregular verbs and number of syllables were all positive, with the θ for negative words being negative. I leave the reader to make the obvious interpretation.

However, you should know the disclaimer by now!

Finally I thought I would classify my own writing. So I took a sample from that other famous political text *Paul Wilmott On Quantitative Finance, second edition*. And I found that I was as right wing as Margaret Thatcher. Take from that whatever you want.

But seriously, if you were to do this for real you would use the above methodology but improve on its implementation in many ways. A couple of obvious improvements would be:

- More training data, meaning many, many more speeches/writings from a larger selection of politicians

- Use a better dictionary, one more suited to classifying politicians instead of one for analysing speeches about the growth in sales of widgets

6.6 Other Regression Methods

You can go a lot further with regression methods than the techniques I've covered here. Just briefly...

The usual suspects

There are many (basis) functions in common use for fitting or approximating functions. You will no doubt have used some yourself. For example, Fourier series, Legendre polynomials, Hermite polynomials, radial basis functions, wavelets, etc. All might have uses in regression. Some are particularly user friendly being orthogonal (the integral of their products over some domain, perhaps with a weighting function, is zero).

Polynomial regression

Rather obviously fit a polynomial in the independent variables. However, the higher the degree of the polynomial the greater the danger of overfitting.

You can use OLS again to find the parameters by treating each non-linear term as if it were another independent variable. So a polynomial fit in one dimension becomes a linear fit in higher dimensions. That is, instead of

$$\theta_0 + \theta_1 x + \theta_2 x^2$$

think of

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2.$$

Because of the obvious correlation between x and x^2 it can be tricky to interpret exactly what the coefficients in the polynomial mean.

Ridge regression

I mentioned in Chapter 2 how one sometimes adds a regularization term to OLS:

$$J(\boldsymbol{\theta}) = \frac{1}{2N}\left(\sum_{n=1}^{N}\left(h_\theta(x^{(n)}) - y^{(n)}\right)^2 + \lambda|\bar{\boldsymbol{\theta}}|^2\right).$$

And similarly for other cost functions. Here I am using $\bar{\boldsymbol{\theta}}$ to mean the vector with the first entry being zero (i.e. there is no θ_0). And for comparison with below, this is the L^2 or Euclidean norm here. This extra penalty term has the effect of reducing the size of the (other) coefficients.

Why on earth would we want to do this? Generally it is used when there is quite a strong relationship between several of the factors. Fitting to both height and age might be a problem because height and age are strongly related. An optimization without the regularization term might struggle because there won't, in the extreme case of perfect correlation, be a unique solution. Regularization avoids this and would balance out the coefficients of the related features.

Lasso regression

LASSO stands for Least Absolute Shrinkage and Selection Operator. This is similar to ridge regularization but the penalty term is now the L^1 norm, the sum of the absolute values rather than the sum of squares.

Not only does Lasso regression shrink the coefficients it also has a tendency to set some coefficients to zero, thus simplifying models. To check this last point, plot $|\bar{\theta}| = \text{constant}$ in θ space (stick to two dimensions!) and draw comparisons between minimizing the loss function with penalty term and minimizing the loss function with a constraint (see Chapter 2 and Lagrange multipliers).

Further Reading

Applied Regression Analysis, 3rd Edition by Norman Draper and Harry Smith published by Wiley covers everything you'll need for linear regression. But it ain't cheap.

After you've mastered linear regression go on to nonlinear with another Wiley book *Nonlinear Regression Analysis and Its Applications* by Douglas Bates and Donald Watts.

Chapter 7

Support Vector Machines

7.1 Executive Summary

Support vector machines are a supervised-learning classification technique. You have classified data represented by vectors of features. This method divides data according to which side of a hyperplane in feature space each point lies.

7.2 What Is It Used For?

Support Vector Machines are used for

- Classifying data based on a set of numerical features
- Example: Identify plant types from features of their petals
- Example: Estimate the risk of prostate cancer from MRI images
- Example: Find likely customers from subjects mentioned in their Twitter posts

The Support Vector Machine (SVM) is another supervised-learning classification technique. There's a little bit more mathematics to it than the methods we've covered so far, hence me saving it until now. It's still implementable in Excel, at least as far as learning the technique is concerned if you don't have too much data. As with all the methods we discuss if you want to use them professionally then you are going to have to code them up properly.

7.3 Hard Margins

In Figure 7.1 are shown sample vectors divided into two classes (the circles and the diamonds). There are very clearly two groups here. These classes can be divided using a straight line, they are linearly separable. This dividing straight line can then tell us to which class a new, unclassified, sample belongs, depending on whether it is on the circle or the diamond side of the line. With such a clear distinction between these groups it should be easy to find such a line. It is, but it is so easy that there are many lines that will do the job, as illustrated in the figure. So which is the best one?

One definition is that the best line is the one that has the largest margins between it and the samples. This is shown as the bold line in the figure.

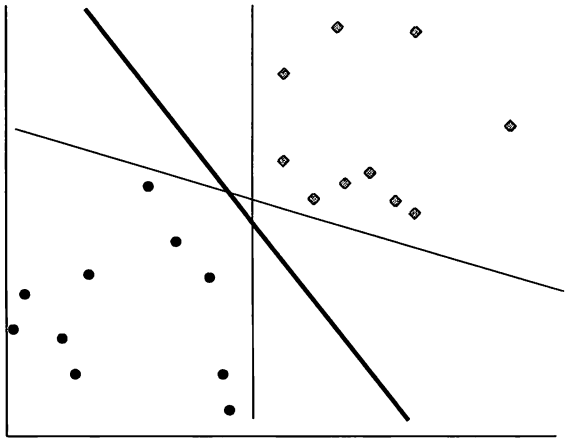


Figure 7.1: Two classes and three possible borders between them.

These margins are shown in Figure 7.2, where I've also shown the vector that is orthogonal to the hyperplane dividing the two classes. I say hyperplane because we will generally be in many dimensions, here of course this hyperplane is just the straight line.

Our goal is to find the hyperplane such that the margins are furthest apart. The cases that define the margins are called the support vectors.

You'll notice that this section is called Hard Margins. This is a reference to there being a clear boundary between the two classes. If one sample strays into the wrong region, a diamond over to the circle side, say, then we will need to do something slightly different from what immediately follows. And we'll look at that in the Soft Margins and Kernel Trick sections.

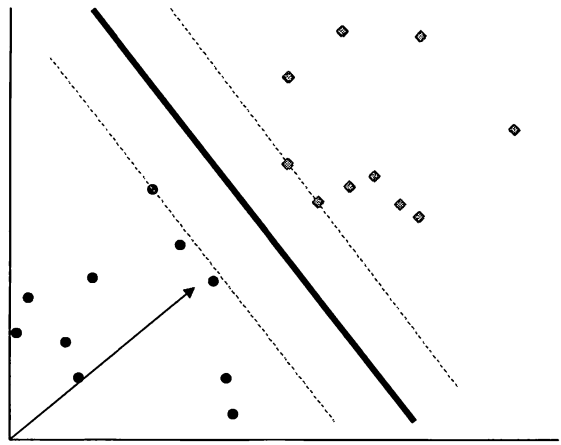


Figure 7.2: Margins and the vector orthogonal to the dividing hyperplane.

Throughout this book I am trying to be as consistent as possible in my notation. So I am going to use notation that is similar to, but subtly different from, that in the previous chapter. For example I am going to use $\boldsymbol{\theta}$ to denote the vector shown in Figure 7.2 that is orthogonal to the hyperplane. This is not quite the same $\boldsymbol{\theta}$ as in Chapter 6, because it doesn't have that first entry, the θ_0, I'm going to keep that separate. It therefore has M dimensions. Similarly $\mathbf{x}^{(n)}$ is the M-dimensional vector representing the n^{th} sample. I shall use $y^{(n)}$ as the class label, being $+1$ for all diamonds and -1 for all circles. Again this is slightly different to earlier when we had 0 and 1 as being the labels. We now want some symmetry to keep the mathematics elegant and so we use instead labels ± 1.

The dividing hyperplane is all points such that

$$\boldsymbol{\theta}^T \mathbf{x} + \theta_0 = 0,$$

where we have to find the vector $\boldsymbol{\theta}$ and the scalar θ_0.

Now you could ask, "Why not just have $\theta_0 = 1$, Paul?" After all we could simply rescale the above equation. And I would answer, "Because I want the margins to be represented by $\mathbf{x}^{\pm}$ where

$$\boldsymbol{\theta}^T \mathbf{x}^{\pm} + \theta_0 = \pm 1." \tag{7.1}$$

And that's where the scaling happened. The $+$ refers to the margin close to the diamonds, and the $-$ to the margin close to the circles.

From (7.1) we have

$$\boldsymbol{\theta}^T (\mathbf{x}^{+} - \mathbf{x}^{-}) = 2. \tag{7.2}$$

Also, for the diamonds (labelled $y^{(n)} = +1$) we will have

$$\boldsymbol{\theta}^T \mathbf{x}^{(n)} + \theta_0 \geq 1 \tag{7.3}$$

and for the circles (labelled $y^{(n)} = -1$)

$$\boldsymbol{\theta}^T \mathbf{x}^{(n)} + \theta_0 \leq -1. \tag{7.4}$$

Multiplying these last two equations by the label $y^{(n)}$ they can both be expressed as simply

$$y^{(n)} \left(\boldsymbol{\theta}^T \mathbf{x}^{(n)} + \theta_0 \right) - 1 \geq 0. \tag{7.5}$$

We also know from Equation (7.2) that (you might have to refresh your memory about the meaning of the inner product of two vectors in terms of distances and angles)

$$\text{margin width} = \frac{2}{|\boldsymbol{\theta}|}.$$

And so maximizing the margin is equivalent to finding $\boldsymbol{\theta}$ and θ_0 to minimize $|\boldsymbol{\theta}|$ or

$$\min_{\boldsymbol{\theta}, \theta_0} \frac{1}{2} |\boldsymbol{\theta}|^2.$$

Of course, subject to the constraint (7.5) for all n.

Once we have found $\boldsymbol{\theta}$ and θ_0 and given a new, unclassified, data point, $\mathbf{u}$, say, we just plug everything into

$$\boldsymbol{\theta}^T \mathbf{u} + \theta_0, \tag{7.6}$$

and depending on whether this is positive or negative we have a diamond or a circle. This is known as the primal version of the classifier.

We shall take a break from the mathematics here and implement a simple example in Excel.

7.4 Example: Irises

There is a popular data set often used in machine learning, a dataset of measurements for different types of iris, the flower. You can get the dataset here: `https://www.kaggle.com/uciml/iris`. I'm not usually keen on using the same data as everyone else. Everyone perfecting their algorithms on the same data is obviously dangerous.

There are three types of iris in the data: Setosa; Versicolor; Virginica. And there are 50 samples of each, with measurements for sepal length and width, petal length and width. (My daughter had to remind me what a sepal was.) The first few lines of raw data in the csv file look like this:

```
SepalLength,SepalWidth,PetalLength,PetalWidth,Name
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5,3.6,1.4,0.2,Iris-setosa
```

In Figure 7.3 are shown the 50 samples for each type of iris, the data being just petal length and sepal width. (I can't plot in four dimensions!)

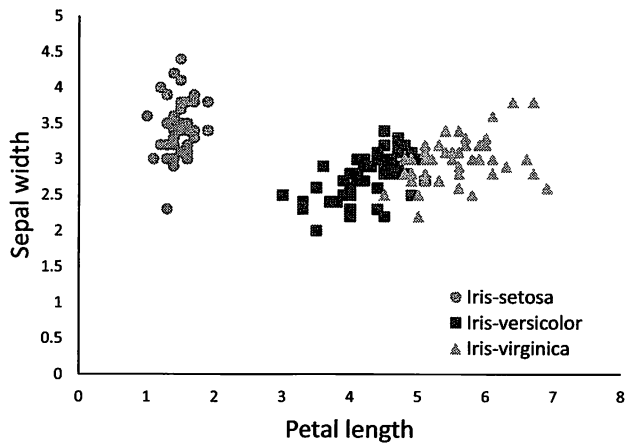


Figure 7.3: Sepal width versus petal length.

For illustrating the SVM method I am going to make this as simple as possible and just use the setosa and versicolor varieties since eyeballing them shows a very clear demarcation. And just in two dimensions, sepal width and petal length, so I can draw the pictures easily. Any results you see below are only in those two dimensions. I would encourage my readers to repeat the analyses but in the full four feature dimensions of the dataset.

Results are shown in Figure 7.4. Ok, not the most challenging example.

I found the hyperplane for the two-dimensional, two-class, iris problem to be

$$0.686 \times \text{sepal width} - 1.257 \times \text{petal length} + 1.057 = 0.$$

The two margins are given by the same equation but with a plus or minus one on the right.

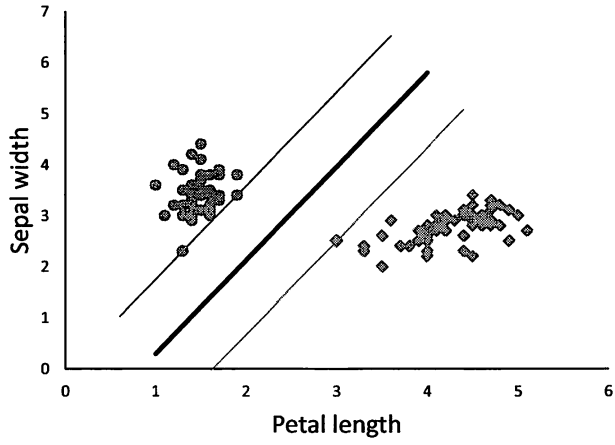


Figure 7.4: The samples, the hyperplane and the margins.

Now given a new iris with values for sepal width and petal length we can determine whether it is setosa or versicolor by plugging the numbers into Expression (7.6), or equivalently the left-hand side of the above, and seeing whether the number that comes out is positive or negative. A positive number for the left-hand side means you have a setosa. If the number has absolute value less than one it means we have an iris in the middle, the no-man's land. We will still have a classification but we won't be confident in it.

As a little experiment see what happens if you move one of the data points, say a setosa, clearly into the versicolor group. When you try to solve for the dividing hyperplane you will find that there is no feasible solution.

7.5 Lagrange Multiplier Version

We can take the mathematics further. Because we have a constrained optimization problem (with inequality constraints) we can recast the problem using Lagrange multipliers, the αs ≥ 0 below, as finding the critical points of

$$L = \frac{1}{2}|\boldsymbol{\theta}|^2 - \sum_{n=1}^{N} \alpha_n \left(y^{(n)} \left(\boldsymbol{\theta}^T \mathbf{x}^{(n)} + \theta_0 \right) - 1 \right). \tag{7.7}$$

To find the critical points we differentiate with respect to the scalar θ_0 and with respect to the vector $\boldsymbol{\theta}$. Differentiation with respect to a vector just means differentiating with respect to each of its entries, and the end

result is pretty much what you would expect. Setting these derivatives to zero you end up with

$$\sum_{n=1}^{N} \alpha_n y^{(n)} = 0 \tag{7.8}$$

and

$$\boldsymbol{\theta} - \sum_{n=1}^{N} \alpha_n y^{(n)} \mathbf{x}^{(n)} = 0.$$

The second of these gives us $\boldsymbol{\theta}$ (albeit in terms of the αs):

$$\boldsymbol{\theta} = \sum_{n=1}^{N} \alpha_n y^{(n)} \mathbf{x}^{(n)}, \tag{7.9}$$

which means that our vector orthogonal to the hyperplane is just a linear combination of sample vectors.

Substituting Equation (7.9) into (7.7) and using (7.8) results in

$$L = \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y^{(i)} y^{(j)} \mathbf{x}^{(i)^T} \mathbf{x}^{(j)}. \tag{7.10}$$

We want to maximize L over the αs, all greater than or equal to zero, subject to (7.8). This is known as the dual problem.

Once we have found the αs the dual version of the classifier for a new point $\mathbf{u}$ is then just

$$\sum_{n=1}^{N} \alpha_n y^{(n)} \mathbf{x}^{(n)^T} \mathbf{u} + \theta_0,$$

whether this is greater or less than zero.

The maximization of L will return almost all αs as zero. Those that aren't zero correspond to the support vectors, the cases that define the margins.

Also note that if the problem is not linearly separable then the Lagrange formulation is not valid. If that is the case then solving the dual problem might give you some αs but if you were to do a sanity check to see whether all your sample points are correctly classified then you'd find some that aren't. I.e. the dual problem has given a solution to a problem that doesn't have a solution.

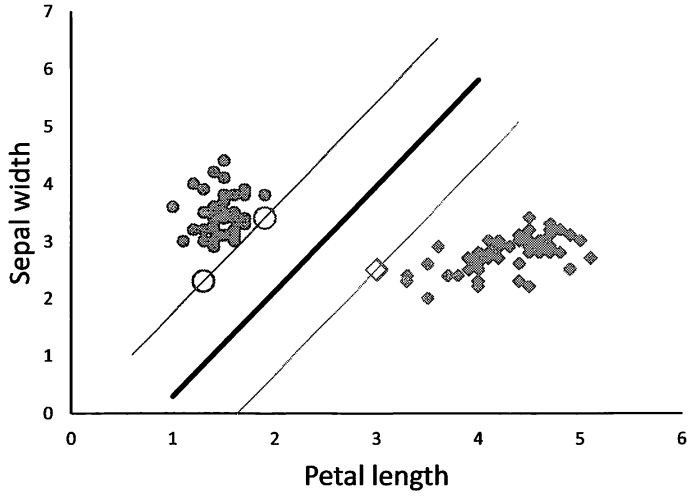


Figure 7.5: The support vectors and the margins.

When we solve the dual problem for the two-dimensional, two-class, iris classification we find that α for the versicolor support vector shown as the hollow diamond in Figure 7.5 is 1.025, for the upper hollow circle is 0.810, and for the lower hollow circle 0.215. And indeed all other αs are zero.

The end result is quite simple in that the decision boundary is defined using just a small subset of the sample vectors. This makes classification of new data very quick and easy.

7.6 Soft Margins

If there is clear water between your groups then SVM works very well.

What if our data is not linearly separable? I'm going to explain two possible ways to address this situation. (In practice you might use both methods simultaneously.) In the first method we try to do the best we can while accepting that some data points are in places you'd think they shouldn't be. It's all about minimizing a loss function.

In the second method we transform our original problem into higher dimensions, thereby possibly giving us enough room to find genuinely linearly separable data. So, to the first method.

Let me introduce the function

$$J = \frac{1}{N}\sum_{n=1}^{N} \max\left(1 - y^{(n)}\left(\boldsymbol{\theta}^T\mathbf{x}^{(n)} + \theta_0\right), 0\right) + \lambda|\boldsymbol{\theta}|^2. \tag{7.11}$$

This we want to minimize by choosing $\boldsymbol{\theta}$ and θ_0. Comments...

- This looks quite a lot like Equation (7.7), but without the different αs, with a maximum function, and a change of sign
- The parameter λ is a weighting between the margin size and whether or not the data lies on the correct side of the margin
- In the language of optimization the first term is the loss function and the second term is the regularization
- The maximum function in the first term has the effect of totally ignoring how far into the correct region a point is and only penalizes distance into the wrong region, measured from the correct margin. The plot of the maximum function against its argument is supposed to look like a hinge, giving this loss function the name hinge loss

Gradient descent

Because this function is convex in the θs we can apply a gradient descent method to find the minimum.

For example, with stochastic gradient descent pick an n at random and update according to

$$\text{New } \theta_0 = \text{Old } \theta_0 - \beta \begin{cases} -y^{(n)} & \text{if } 1 - y^{(n)}\left(\boldsymbol{\theta}^T\mathbf{x}^{(n)} + \theta_0\right) > 0 \\ 0 & \text{otherwise} \end{cases}$$

and

$$\text{New } \boldsymbol{\theta} = \text{Old } \boldsymbol{\theta} - \beta \begin{cases} 2\lambda\boldsymbol{\theta} - \frac{1}{N}y^{(n)}\mathbf{x}^{(n)} & \text{if } 1 - y^{(n)}\left(\boldsymbol{\theta}^T\mathbf{x}^{(n)} + \theta_0\right) > 0 \\ 2\lambda\boldsymbol{\theta} & \text{otherwise} \end{cases}.$$

Technically you might want to call this sub-gradient descent because the derivatives aren't defined at the hinge.

In Figure 7.6 I've taken the earlier iris data and moved one setosa deep into versicolor territory. The resulting margins have become wider, and are softer since a few data points are within those margins.

You can also use the soft-margin approach even if you have linearly separable data. It can be used to give wider margins but some points will now lie within the margins.

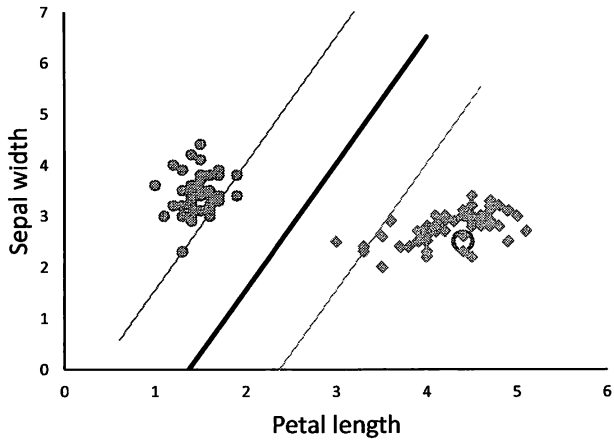


Figure 7.6: Soft margins.

7.7 Kernel Trick

We might have data that is not linearly separable but which nonetheless falls neatly into two groups, just not groups that can be divided by a hyperplane. A simple example would be that shown in Figure 7.7.

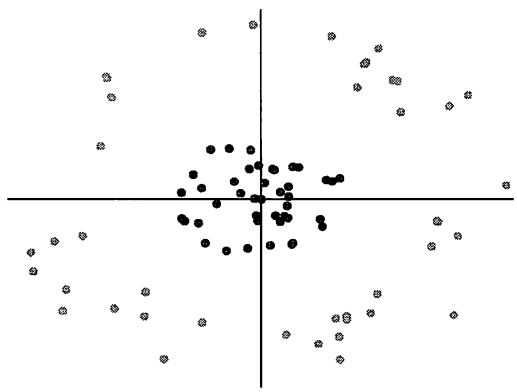


Figure 7.7: Two clear groups, but not linearly separable.

In this problem we could change to polar coordinates in which case the data becomes trivially linearly separable.

But there's another thing we could do and that is to go into higher dimensions.

For the data in Figure 7.7 we could go to three dimensions via the transformation

$$(x_1, x_2) \rightarrow (x_1, x_2, x_1^2 + x_2^2).$$

This would have the effect of lifting up the outer group of dots higher than the inner group in the new third dimension, making them linearly separable.

But once we start moving into higher dimensions we find that the training procedure gets more time consuming. Is there any way to take advantage of higher dimensions without suffering too much from additional training time? Yes, there is. And it follows on from two observations.

First observation: Thanks to the dual formulation, in (7.10), we can see that finding the αs depends only on the products $\mathbf{x}^{(i)^T}\mathbf{x}^{(j)}$.

Second observation: If you plug (7.9) into (7.6) you can see that the classification of new data only depends on the products $\mathbf{x}^{(i)^T}\mathbf{u}$.

So there is something special about this product.

If we use $\phi(\mathbf{x})$ to denote a transformation of the sample points (into higher dimensions) then the above observations show us that all we need to know for training and for classifying in the higher-dimensional space are

$$\phi(\mathbf{x}^{(i)})^T\phi(\mathbf{x}^{(j)})$$

and

$$\phi(\mathbf{x}^{(i)})^T\phi(\mathbf{u}),$$

for our transformed data.

So rather than needing to know the exact data points we actually only need a lot less information: The products of the vectors. And this leads on to an idea that exploits these observations to address problems that are not immediately linearly separable. And this idea is powerful because we can go to many, perhaps even infinite, dimensions without adding much numerical complexity or programming time.

The function $\phi(\mathbf{x})$ is known as a feature map. And what is clever about the feature-map transformation is that you don't need to explicitly know what the map is in order to exploit it!

The trick in Kernel Trick is to transform the data from our M-dimensional space into a higher-dimensional space where the data is linearly separable.

Let me show how this idea works with an example. Specifically let's work with the example of $\phi(\mathbf{x})$ transforming a two-dimensional vector $(x_1, x_2)^T$ into a six-dimensional vector

$$(b, \sqrt{2ab}\, x_1, \sqrt{2ab}\, x_2, ax_1^2, ax_2^2, \sqrt{2}\, ax_1x_2)^T,$$

where a and b are constant parameters. (Obviously this transformation was not chosen at random, as we shall see.)

So

$$\phi(\mathbf{x})^T\phi(\mathbf{x}') =$$

$$b^2 + 2ab\, x_1x_1' + 2ab\, x_2x_2' + a^2\, x_1^2x_1'^2 + a^2\, x_2^2x_2'^2 + 2a^2x_1x_2x_1'x_2'.$$

This can be written as

$$(ax_1x_1' + ax_2x_2' + b)^2 = \left(a\mathbf{x}^T\mathbf{x}' + b\right)^2. \tag{7.12}$$

Underwhelmed? Then I haven't explained properly! The key point is that we have gone from two to five dimensions — the constant in the first position doesn't really count as a dimension — opening up all sorts of possibilities for finding linearly separable regions but *we didn't need to know the details of the transformation* $\phi(\mathbf{x})$. All we needed was that the product in the new variables is a function of the product in the original variables. That function is (7.12).

And all we do now with SVM in the higher-dimensional space is to replace all instances of

$$\mathbf{x}^T\mathbf{x}'$$

with the kernel function

$$K(\mathbf{x},\mathbf{x}') = \left(a\mathbf{x}^T\mathbf{x}' + b\right)^2.$$

Computationally speaking there is little extra work to be done, just an addition and raising to a power.

Of course, this is not the only kernel for which this trick works. Here are a few more:

$$K(\mathbf{x},\mathbf{x}') = \left(a\mathbf{x}^T\mathbf{x}' + b\right)^p, \text{ polynomial kernel,}$$

$$K(\mathbf{x},\mathbf{x}') = \exp\left(-\frac{|\mathbf{x}-\mathbf{x}'|^2}{2\sigma^2}\right), \text{ Gaussian or Radial Basis Function kernel,}$$

$$K(\mathbf{x},\mathbf{x}') = \exp\left(-\frac{|\mathbf{x}-\mathbf{x}'|}{2\sigma}\right), \text{ exponential kernel,}$$

and so on. Google SVM + kernel + polynomial + gaussian + "Inverse Multiquadric Kernel" and you should hit some lists.

After swapping the kernel function into (7.10) the quantity to maximize becomes

$$L = \sum_{n=1}^{N}\alpha_n - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y^{(i)}y^{(j)}K\left(\mathbf{x}^{(i)},\mathbf{x}^{(j)}\right). \tag{7.13}$$

The observant will look at the above examples of kernel functions and say "I can see the product in the polynomial kernel, but I don't see it in the Gaussian kernel." Well, it is in there but we need to do a little bit of work to see it.

First of all, what do we mean by the product of two vectors in the present context? Let's write $\phi(\mathbf{x})$ in long hand:

$$\phi(\mathbf{x}) = (\phi_1(x_1, \ldots, x_n), \phi_2(x_1, \ldots, x_n), \ldots, \phi_m(x_1, \ldots, x_n)) .$$

Notice how we've gone from n to $m > n$ dimensions. So the product we are interested in is

$$\sum_{i=1}^{m} \phi_i(x_1, \ldots, x_n)\phi_i(x'_1, \ldots, x'_n).$$

The key element in this is that the sum is made up of products of the *same* function of the two initial coordinates $(x_1, \ldots, x_n)$ and $(x'_1, \ldots, x'_n)$. So our goal is to write the Gaussian kernel in this form. Here goes

$$\exp\left(-\frac{|\mathbf{x} - \mathbf{x}'|^2}{2\sigma^2}\right) = \exp\left(-\frac{\mathbf{x}^T\mathbf{x} - 2\mathbf{x}^T\mathbf{x}' + \mathbf{x}'^T\mathbf{x}'}{2\sigma^2}\right)$$

$$= \exp\left(-\frac{\mathbf{x}^T\mathbf{x}}{2\sigma^2}\right) \exp\left(\frac{\mathbf{x}^T\mathbf{x}'}{\sigma^2}\right) \exp\left(-\frac{\mathbf{x}'^T\mathbf{x}'}{2\sigma^2}\right) .$$

The first and last terms in this are of the form that we want. But not; yet, the middle. That is easily dealt with by expanding in Taylor series. The middle term is then

$$\sum_{i=0}^{\infty} \frac{1}{i!} \left(\frac{\mathbf{x}^T\mathbf{x}'}{\sigma^2}\right)^i .$$

And the job is done, because each of these terms is a polynomial kernel. So indeed the Gaussian kernel does have the required product form, albeit as an infinite sum and therefore in infinite dimensions.

Note that some kernels take you from a finite-dimensional space to another finite-dimensional space with higher dimension. The polynomial kernel is an example of this. However others take you to an infinite-dimensional space.

If we go back to the iris problem in which I moved one setosa into the versicolor region then using a kernel might be able to make this a separable problem. However in so doing the boundary of the two regions would inevitably be very complex. We might find that classifying unclassified irises would not be very robust. It would almost certainly be better to treat the odd iris as an outlier, a one off. So if you have data that is only linearly inseparable because of a small number of outliers then use soft margins. If however the original problem is not linearly separable but there is structure then use a kernel.

Further Reading

You can often find inexpensive, often self published, monographs on machine-learning topics, sometimes they will be a student's thesis. Sadly SVM doesn't seem to be such a topic and most books are expensive. So if you could only afford one book it might be *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond* by Bernhard Schölkopf and Francis Bach published by MIT Press. It's quite advanced though.

And, of course, *The Nature of Statistical Learning Theory* by one of the founders of SVM, Vladimir Vapnik.

Chapter 8

Self-Organizing Maps

8.1 Executive Summary

A self-organizing map is an unsupervised-learning technique. We start with vectors of features for all our sample data points. These are then grouped together according to how similar their vectors are. The data points are then mapped to a two-dimensional grid so we can visualize which data points have similar characteristics.

8.2 What Is It Used For?

Self-organizing maps are used for

- Grouping together data points according to similarities in their numerical features and visualizing the results
- Example: Group people according to the contents of their supermarket baskets
- Example: Group countries together according to demographics and economic statistics

A self-organizing map (SOM) is a simple unsupervised-learning method that both finds relationships between individual data points and also gives a nice visualization of the results. Some people think of this technique as a version of a neural network. The end result of the method is a typically two-dimensional picture, consisting of a usually square array of cells as in Figure 8.1. Each cell represents a vector and individual items will be placed into one of the cells. Thus each cell forms one group and the closer that two cells are to one another the more they are related.

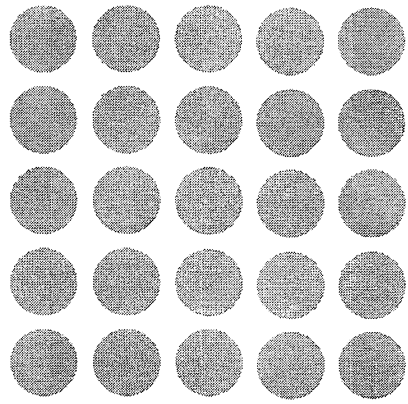


Figure 8.1: An array of cells.

8.3 The Method

The data

Each item that we want to analyze and group is represented by a vector, $\mathbf{x}^{(n)}$, dimension M, of numerical data. There will be N such items, so $n = 1$ to N.

As is often the case we shall be measuring and comparing distances. So we need to make sure that the magnitude of distances are similar for all elements. Just as we've done before we should translate and scale, to either make mean and standard deviations the same, or the minimum and maximum, for all features.

Each group/cell/node is represented by a vector $\mathbf{v}^{(k)}$, also of dimension M. And we'll have K such groups. For visualization purposes in two dimensions it is usual to make K a square number, 25, 64, 100.... These $\mathbf{v}$s are often called weights.

But we are getting ahead of ourselves. Before we go on to talk about two-dimensional arrays and visualization let's talk about how we determine in which group we put item n. We have the vector $\mathbf{x}^{(n)}$ and we want to decide whether it goes into Group A, B, C, D, ..., this is represented in Figure 8.2.

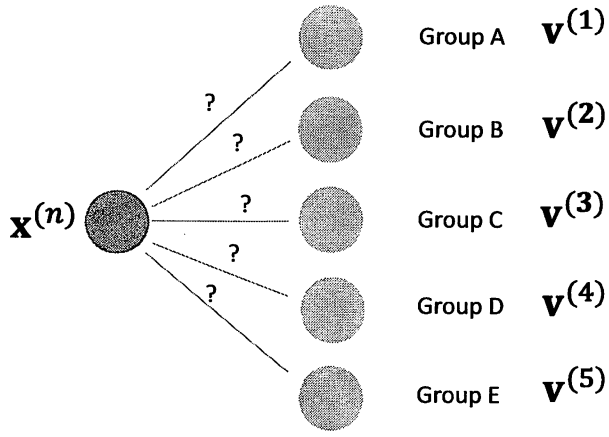


Figure 8.2: Which group should item n go into?

Remember that all of $\mathbf{x}^{(n)}$ and the $\mathbf{v}$s are vectors with M entries. All we do is to measure the distance between item n and each of the K cell vectors $\mathbf{v}$:

$$|\mathbf{x}^{(n)} - \mathbf{v}^{(k)}| = \sqrt{\sum_{m=1}^{M} \left(x_m^{(n)} - v_m^{(k)}\right)^2}.$$

The k for which the distance is shortest

$$\underset{k}{\operatorname{argmin}} |\mathbf{x}^{(n)} - \mathbf{v}^{(k)}|,$$

is called the Best Matching Unit (BMU) for that data point n.

But how do we know what the vs are?

Finding the BMU is straightforward. But that's assuming we know what the $\mathbf{v}$s are. We could specify them manually: Classify dogs according to different characteristics. But, no, instead we are going to train the system to find them.

We begin by redistributing our $\mathbf{v}$ vectors in a square array, as shown in Figure 8.3. However unless there is some relationship between cells then this redistribution is without any meaning. More anon.

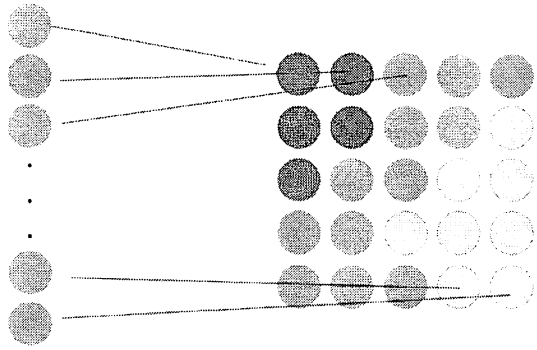


Figure 8.3: Redistributing the vectors into an array.

We are now ready to bring everything together... via learning.

8.4 The Learning Algorithm

The Self Organizing Maps Agorithm

Step 0: Pick the initial weights

We need to start with some initial $\mathbf{vs}$, which during the learning process will change. As always there are several ways to choose from, perhaps pick K of the N items, or randomly generate vectors with the right order of magnitudes for the entries.

Step 1: Pick one of the N data vectors, the $\mathbf{xs}$, at random

Suppose it is $n = n_t$. Find its BMU.
We are going to be picking from the $\mathbf{xs}$ at random as we iterate during the learning process. Keep track of the number of iterations using the index t. So t starts at 1, then next time around becomes 2, and so on.

Step 2: Adjust the weights

Change all of the weights $\mathbf{v}^{(k)}$ for $k = 1$ to K slightly to move in the direction of $\mathbf{x}^{(n_t)}$. The closer a cell is to the BMU that we've just found the more the weight is adjusted. This is where it gets interesting. Let's see the mathematical details.
What do we mean by closer? This is where the topography of the two-dimensional array comes in, and the distribution of cells becomes meaningful. Look at Figure 8.4 which shows the current BMU and surrounding cells.

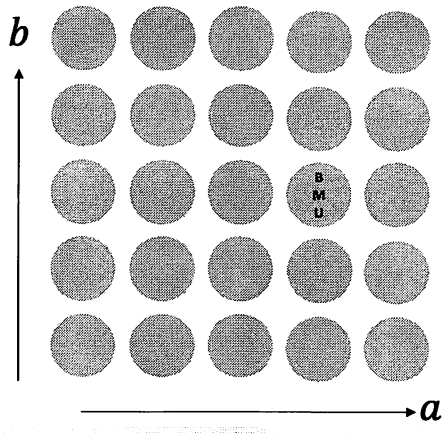


Figure 8.4: The BMU and neighbouring cells.

Measure the distance between the BMU cell and other nodes, according to the obvious

$$\text{Distance} = D = \sqrt{(a - a_{\text{BMU}})^2 - (b - b_{\text{BMU}})^2}. \qquad (8.1)$$

This distance is going to determine by how much we move each $\mathbf{v}^{(k)}$ in the direction of our randomly chosen item $\mathbf{x}^{(n_t)}$. The as and bs are integers representing the position of each node in the array.
Note that this is not the same distance we measured when we were comparing vectors. It is the distance between the cells, as drawn on the page and nothing to do with the vectors that these cells represent. And here is the updating rule:

$$\mathbf{v}^{(k)} \leftarrow \mathbf{v}^{(k)} + \beta\left(\mathbf{x}^{(n_t)} - \mathbf{v}^{(k)}\right) \quad \text{for all} \quad 1 \leq k \leq K.$$

Here β is the learning rate. It is a function of the distance, D, in (8.1) and the number of iterations so far, t.
I have to confess that the next part is where I personally start to get a bit disappointed. And that's because, like so much of numerical analysis, what you do is a bit arbitrary. So, for example, we might choose the function $\beta(D,t)$ to be

$$\beta(D,t) = \eta(t) \exp\left(-\frac{D^2}{2\sigma(t)^2}\right)$$

where

$$\eta(t) = \eta_0 \exp\left(-\frac{t}{\tau_\eta}\right)$$

and

$$\sigma(t) = \sigma_0 \exp\left(-\frac{t}{\tau_\sigma}\right).$$

This can be interpreted as:

- The adjustment to each weight decreases with time (i.e. iteration)
- The adjustment to each weight decreases with distance from that iteration's BMU
- The effective range over which each weight is adjusted decreases with iteration

See what I mean by a bit arbitrary?
If entries in the weights are mean zero, range of one, then a suitable η_0 might be around 0.2. And σ_0 might be around $\frac{1}{2}\sqrt{K}$. There are typically two phases during the learning process. The first is a topological ordering where adjacent cells change to have similar weights and then convergence.

Step 2: Iterate

Go back to Step 1 above and repeat until convergence.

8.5 Example: Grouping shares

Suppose you want to classify constituents of the S&P500 index. You could try:

1. Grouping according to sector, market capitalization, earnings, gender of the CEO,...
2. Measure expected returns, volatilities and correlations
3. Principal Components Analysis of returns

But that's what everyone does. The second of these is the popular (at least in the text books) Modern Portfolio Theory (MPT).

MPT is a clever, Nobel-Prize winning, way of deciding which stocks to buy or sell based on three types of statistical information: Each stock's expected return; Each stock's volatility; The correlations between all stocks' returns. In this method one aims to maximize the expected return of a portfolio of stocks while holding its volatility, or risk, constant. To achieve this it exploits correlations between stocks. If two stocks both have a good expected growth but those returns are uncorrelated then spreading your money across a portfolio of the two of them will be better than holding just one. And so on for portfolios of any number of stocks. The problem with this elegant theory is that in practice the parameters, especially the correlations, are very unstable.

Perhaps SOM might be a more stable method? It is also nonlinear generally, so might capture something that MPT doesn't.

However, having asked that question I'm not going to fully answer it. I shall use SOM on stock returns data but won't go too far in the direction of portfolio selection. That is more for a finance research paper than for illustrating machine learning.

What I am going to do is a slightly unusual example of self-organized maps. Usually one would have significantly different features for each stock (just like we had different crimes for each local authority when we did K Means). Perhaps we would have the sector, market capitalization, earnings, gender of the CEO, etc. mentioned above. But I want to tie this in more closely to the MPT of classical quantitative finance. For that reason I am going to use annual stock price returns as a feature. So for each stock I will have a vector of five features, the last five annual returns. I have chosen annual data because it fits in with the sort of timescale over which portfolio allocation methods are used, one doesn't usually reposition one's portfolio every day.

I will use annual returns for each of 476 constituents of the S&P index.

("Why not 500?" you ask? "Because some stocks left the index and others joined during this period," I reply. Since some stocks have left the

index there will be a survivorship bias in my results. But if you're worried about biases then the choosing-fun-examples bias is probably going to be more important, and it's throughout this book.)

These returns are the $\mathbf{x}^{(n)}$s. So the features of the stock are returns and the m^{th} entry in each $\mathbf{x}^{(n)}$ is the return over the m^{th} year.

And I will have $K = 25$.

A note on scaling Because I've chosen features which are very similar, just returns over different years, scaling is not as crucial as it would be with features that are significantly different. So in what follows I have left all data unscaled.

Let's see what the algorithm gives us.

In Figure 8.6 we see how many of the 476 stocks appear in each node, with the corresponding contour plot in Figure 8.6. Clearly some nodes are more popular than others. In those nodes there are a lot of stocks that move closely in sync.

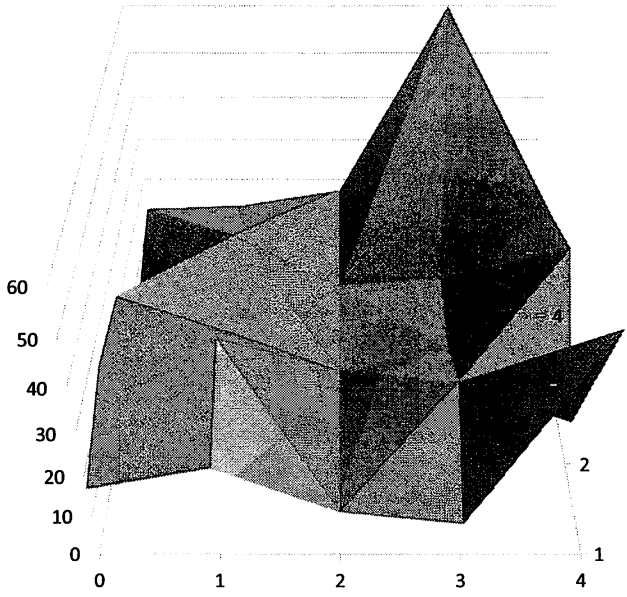


Figure 8.5: Number of stocks per node.

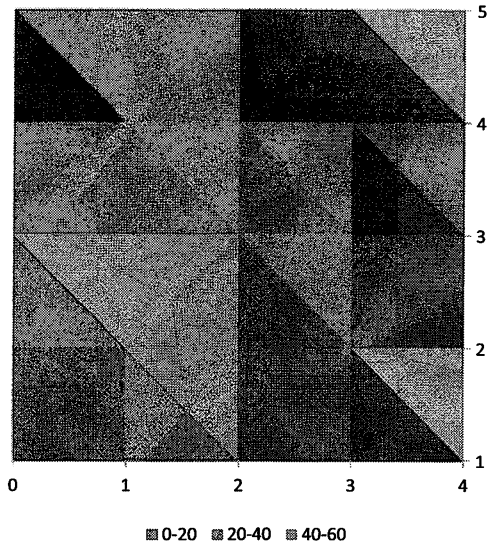


Figure 8.6: Number of stocks per node. Contour plot.

The most popular node, the one with the most stocks, is number 20. Its constituents are shown in Table 8.1.

ADM	BAX	DVN	GRMN	KMX	NI	PWR	WDC
ADS	BEN	EBAY	HAL	KSU	NOV	PXD	WMB
AKAM	BWA	EOG	HES	LNC	NRG	QCOM	WYNN
AMG	CF	ETN	HOG	LUK	NSC	RRC	XEC
APA	CHK	FCX	HP	MOS	NTAP	STX	
APC	CMI	FLS	HST	MRO	OKE	UAL	
ARNC	COP	FMC	IP	MU	PKG	UNP	
AXP	DISH	FTI	KMI	NBL	PNR	URI	

Table 8.1: Constituents of Node 20.

Inspired by MPT one could easily use the growth and volatility of each stock together with the map structure to optimize a portfolio in a way that would be different from classical methods. For example diversification could be achieved by not buying stocks that are in nodes that are close to each other. In Figure 8.7 is shown the average in each node of the ratio of the stocks' growths to the volatilities. Perhaps these numbers will be useful. I shall leave these possibilities as an exercise for the reader. (But please don't

put any of your hard-earned cash on any investments based on my analysis.)

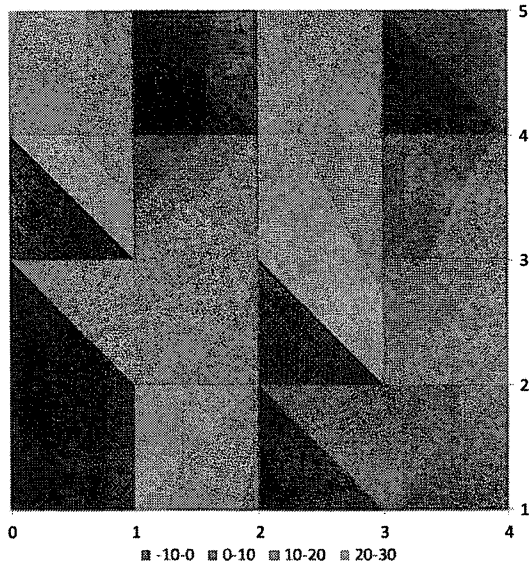


Figure 8.7: Average growth/volatility for each Node.

I want to finish this example by looking at what this solution has to say about financial sectors. In Figure 8.8 I show the number of stocks from each sector in each cell.

Node	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Consumer Discretionary	1	4	2	0	4	3	3	2	4	3	3	0	0	1	5	0	2	1	7	6	2	1	14	3	3
Consumer Staples	0	2	7	1	0	1	2	0	1	1	0	2	1	1	2	0	0	0	0	1	0	1	4	4	1
Energy	1	0	0	0	0	0	0	0	3	3	0	0	0	0	2	0	0	0	0	18	1	0	0	1	1
Financials	0	1	0	0	0	1	4	4	1	1	0	1	0	2	1	0	7	2	0	5	25	3	3	1	1
Health Care	0	0	1	0	4	8	4	0	2	0	2	0	0	1	3	4	1	3	1	1	1	0	8	11	3
Industrials	0	2	0	1	0	3	6	4	6	2	1	0	0	0	1	1	6	2	0	11	10	2	4	1	1
Information Technology	3	0	0	0	3	6	11	2	2	1	1	1	0	0	3	2	5	0	1	8	9	3	0	1	1
Materials	1	0	0	1	1	0	1	1	0	2	0	1	0	0	0	0	0	2	1	6	3	1	1	1	0
Real Estate	6	8	6	1	0	0	0	0	0	0	4	0	0	1	0	1	0	0	0	1	0	0	0	1	2
Telecommunication Services	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
Utilities	2	6	7	1	0	0	0	0	1	0	0	2	3	2	0	0	0	0	0	2	1	0	0	1	0

Figure 8.8: Number of stocks from each sector in each cell.

Figure 8.9 shows this as a 3D plot. Node numbers from 1 to 25 run left to right and sectors go into the page.

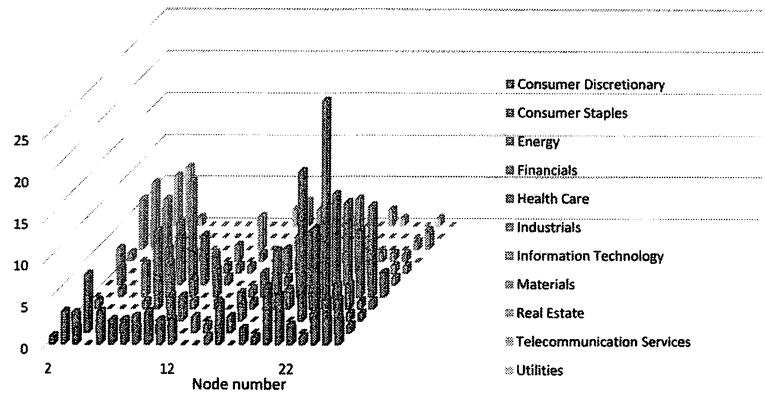


Figure 8.9: Sectors in each node.

It's not easy to appreciate this in black and white but you can get a rough idea of what is going on, especially if you also look at the contour map in Figure 8.10.

There are clearly some sectors that are very focused: Financials are concentrated around Node 21; Real Estate around Nodes 1–3. Energy around Node 20.

There are some sectors that are very spread out, such as Consumer Discretionary.

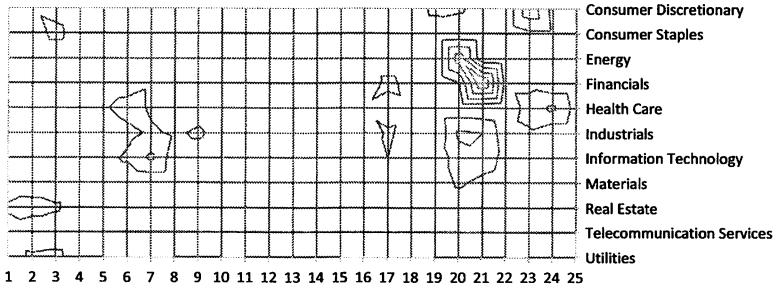


Figure 8.10: The number of stocks from each sector in each cell. The contour map.

Note that I never told the algorithm anything about sectors. But it does look as if SOM has found something rather sector-like, but only for some sectors.

8.6 Example: Voting in the House of Commons

At time of writing the UK is going through some political turmoil. (As you are reading this in the future, how did it all turn out?) So I couldn't resist doing something with voting in Parliament. I downloaded voting data from `https://www.publicwhip.org.uk`.

I took the last 18 months of voting records of Members of Parliament, who voted, in which direction and who abstained or wasn't present.

There are 650 members of Parliament, the MPs. However, the data gets a bit confusing. Some MPs don't appear in the data because Sinn Fein MPs never attend the House of Commons, they don't fancy swearing allegiance to the Queen apparently. And quite a few MPs have been kicked out of the Labour party, and other MPs have become independent. Every time something like that happens the MP gets a new ID Number which complicates things a bit. I haven't dealt with the redesignation of such MPs properly, but it would be interesting to look at their voting before and after changing allegiance.

Each entry in the vector represents an individual's position on a specific vote. I have used $+1$ to represent a vote in favour, -1 for a vote against, and zero for abstention or not being present. Now obviously turning such data into numbers has to be meaningful for this method to work. Fortunately the ordering no, abstention, aye is quite well represented by -1, 0, 1. If we had used data for sock colour instead then replacing red, blue, yellow, striped, spotted,... with numbers would have been silly.

The results are shown in Figures 8.11 and 8.12. The first figure shows that there are two main groupings, surprise surprise.

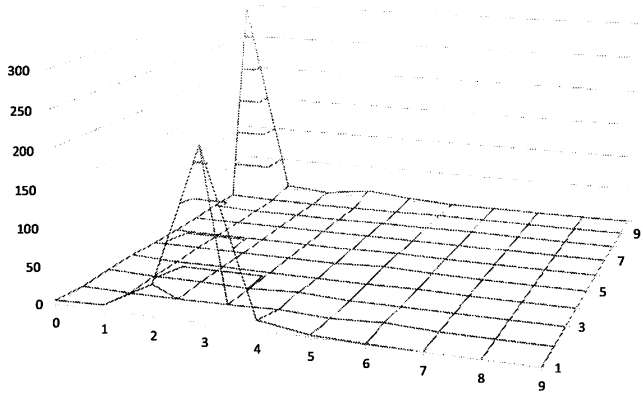


Figure 8.11: Number of MPs per node.

The second figure shows the breakdown of the nodes by political party. Conservative and Labour are predominantly in completely different areas of the grid. However Labour are a little bit more spread out. Interestingly, there are members of the main parties scattered about, not being where you'd expect them to be. It would be interesting to look at those MPs who don't seem to be affiliated with the correct party and why. Note that nodes 11, 21, 31, etc. are all close together in the grid.

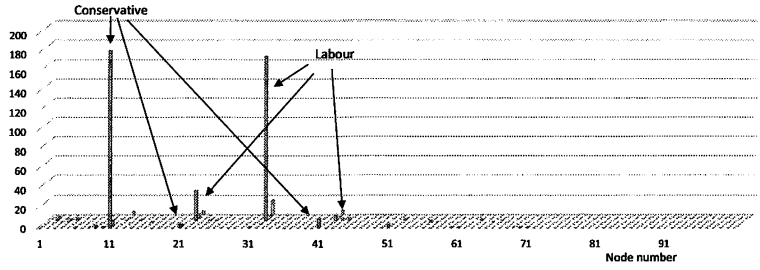


Figure 8.12: Political affiliation and nodes.

Obviously I have used this data for illustrative purposes. If this were being done for real I would take a lot more care, such as testing the results, and also probably break down the data according to things like the subject of the vote.

Further Reading

You might want to take a look at the Springer book *Self-Organizing Maps: Third Edition* by the creator of the subject Teuvo Kohonen.

There is a paper by Claus Huber, "R Tutorial on Machine Learning: How to Visualize Option-like Hedge Fund Returns for Risk Analysis," on SOM applied to hedge funds, including R code, in *Wilmott* magazine January 2019 pp36–41. It is also available for free on `wilmott.com`.

Chapter 9

Decision Trees

9.1 Executive Summary

Decision trees are a supervised-learning technique. Really they are just flowcharts. They can be used for classification and for regression, so you'll also hear them referred to as Classification and Regression Trees (or CART). If used for classification then we start with labelled data points with multiple features. These features describe each data point according to their attributes (tall or short, man or woman, or numerical values). The method uses a hierarchy of divisions of the data points according to these attributes. Trees can also be used for regression if the data has associated numerical values.

9.2 What Is It Used For?

Decision trees are used for

- Classifying data according to attributes that can be categories or numerical
- Example: Predict the number of votes in the Eurovision Song Contest by looking at type of song, beats per minute, number of band members, etc.
- Example: Forecast IQ based on books read, type, subjects, authors, etc.

You will no doubt have seen decision trees before, although perhaps not in the context of machine learning. You will definitely have played 20 Questions

before. "Is the actor male?" Yes. "Is he under 50?" No... "Is he bald?" Yes. "Bruce Willis?" Yes. If the answer to the first question had been "No" then you would have taken a different route through the tree.

In 20 Questions the trick is to figure out what are the best Yes/No questions to ask at each stage so as to get to the right answer as quickly as possible, or at least in 20 or fewer questions. In machine learning the goal is similar. You will have a training set of data that has been classified, and this is used to construct the best possible tree structure of questions and answers so that when you get a new item to classify it can be done quickly and accurately. So decision trees are another example of supervised learning. The questions in a decision tree do not have to be binary, and the answers can be numerical.

I would not recommend using Excel for decisions trees when you have a real problem, with lots of data. It's just too messy. That's because you don't know the tree structure before you start, making it difficult to lay things out in a spreadsheet.

First some jargon and conventions. The tree is drawn upside down, the root (the first question) at the top. Each question is a condition or an internal node that splits features or attributes. From each node there will be branches or edges, representing the possible answers. When you get to the end of the path through the tree so that there are no more questions/decisions then you are at a leaf. There are also the obvious concepts of parent and child branches and nodes. See Figure 9.1.

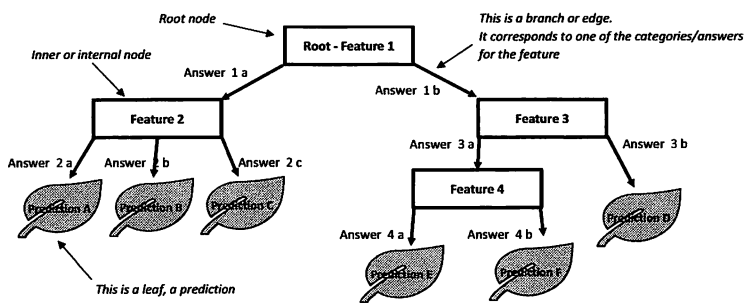


Figure 9.1: A schematic diagram showing the characteristics of a decision tree.

You can use decision trees to classify data, such as whether or not a mushroom is edible given various physical features. Those features could have binary categories such as with/without gills, or multiple categories,

such as colour, or be numerical, height of mushroom, for example. Decision trees also can be used for regression, when you have numerical data, how much is a car worth based on make, model, age, etc.

Building or growing a tree is all about choosing the order in which features of your data are looked at and what the conditions are. There might be some pruning to be done and you'll need to know when to stop.

9.3 Example: Magazine subscription

I am going to work with member data from my own website now. I shall be taking data for a small subset of members to figure out which people are likely to subscribe to our magazine. (And along the way I'll be shamelessly promoting both the website and the magazine.) I will not be violating any data-protection laws.

There are three features of `wilmott.com` members we will look at: Employment Status; Highest Degree Level; CQF Alumnus (whether or not they have the Certificate in Quantitative Finance qualification). And the classification is whether or not they are magazine subscribers. The goal is to use information about those features for new members to determine whether or not they will be subscribers of the magazine.

ID	Employment Status	Degree Level	CQF Alumnus	Wilmott Magazine Subscriber
1	Self Employed	Postgraduate	No	No
2	Self Employed	Postgraduate	Yes	Yes
3	Employed	Postgraduate	Yes	Yes
4	Student/Postdoc.	Postgraduate	No	Yes
5	Student/Postdoc.	Undergraduate	Yes	Yes
6	Student/Postdoc.	Undergraduate	Yes	No
7	Employed	Undergraduate	Yes	Yes
8	Self Employed	Postgraduate	No	No
9	Self Employed	Undergraduate	No	Yes
10	Student/Postdoc.	Undergraduate	Yes	No
11	Self Employed	Undergraduate	Yes	Yes
12	Employed	Postgraduate	Yes	Yes
13	Employed	Undergraduate	No	Yes
14	Student/Postdoc.	Postgraduate	Yes	No
15	Employed	Postgraduate	No	Yes
16	Student/Postdoc.	Postgraduate	No	No
17	Self Employed	Postgraduate	No	No

Figure 9.2: The raw data.

I am only going to work with a small subset of the full magazine dataset, so just 17 lines. See Figure 9.2. Out of these 17 members there are ten who

are magazine subscribers.

The obvious use of any results that we find is in helping decide who to target for subscriptions.

Just looking at this table we cannot immediately see any single question, whether they are employed etc., that will determine whether or not someone is a magazine subscriber. It's not as if having a postgraduate degree means you do subscribe and not having one means you don't. So we are going to have to ask *several* questions, using more of the data in the table, to pin down people's subscription status. But in what order should we ask the questions? What is, in some sense to be decided, most efficient?

Now I must confess that I have tweaked the data a bit. That is so that I can get a single decision tree that shows as many outcomes as possible. You will shortly see what I mean. (Obviously the real data shows simply that the better educated a person is then the more likely they are to subscribe. Haha!)

I'm first going to explain how the final decision tree works, but *how I constructed it* in an optimal fashion will come a bit later.

Suppose I start by asking members of `wilmott.com` ("Serving the Quantitative Finance Community Since 2001") about their employment status and whether or not they are magazine subscribers. They can answer Employed, Self Employed or Student and Yes or No. According to the data in Figure 9.2 we would get results that could be represented by Figure 9.3. This is the root of our decision tree.

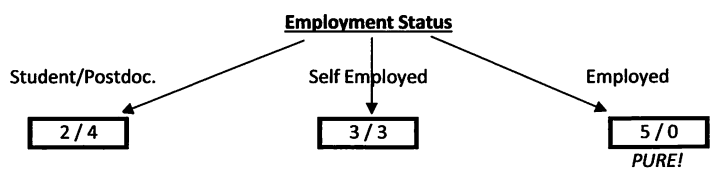


Figure 9.3: Responses to question about employment.

We read this as follows. In the boxes are two numbers. The number on the left is the number of people who are magazine subscribers and the number on the right those who aren't. Is this useful?

Yes and no. If they say they are Employed then this is very useful because five employed people say they are magazine subscribers and none say they aren't. That is very useful information because as soon as we know someone is employed we know that they will be subscribers. And no more questions need be asked.

If only it were true that all employed people subscribed to the magazine. Sadly, this data has been massaged as I mentioned. In practice you would

have a lot more data, you'd be unlikely to get such a clear-cut answer, but you'd have a lot more confidence in the results. However even with these numbers we can start to see some ideas developing. When you get an answer that gives perfect predictability like this it is called pure. That is the best possible outcome for a response. The worst possible outcome is if you get the answer Self Employed because there is no information contained in that answer, it is 50:50 whether or not a self-employed person is a subscriber. In fact we seem to have gone backwards, at least before we had asked any questions we knew that ten out of 17 people were subscribers, now we are at a coin toss.

If someone is a Student/Postdoc. then two out of six are subscribers. In terms of information, two out of six and four out of six are equally useful.

We are now going to look at the sub trees. We can ask the Student/Postdoc. whether they are a CQF Alumnus. Out of the six Students/Postdocs there are four who are alumni and two who aren't. See Figure 9.4. The left branch of the CQF Alumnus, the one labelled Yes, has four people at the end, the No branch has two. Out of the four CQF Alumni there is only one magazine subscriber. You see those numbers in the box, a 1 and a 3. And the two non alumni are equally split between subscribing and not. So a 1 and a 1.

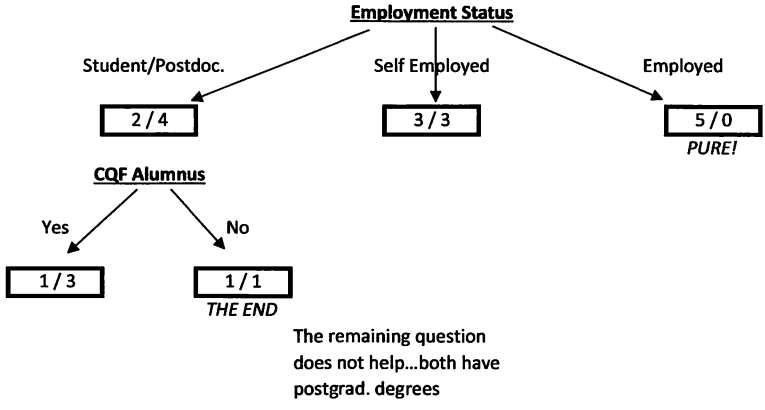


Figure 9.4: The first sub tree.

We can move further down the tree. We can ask those two non-alumni Students/Postdocs what their highest degree is. Unfortunately that does not help here because both have the same level, they have postgraduate degrees. There's nothing we can do with their answers to the three questions that will separate these two individuals.

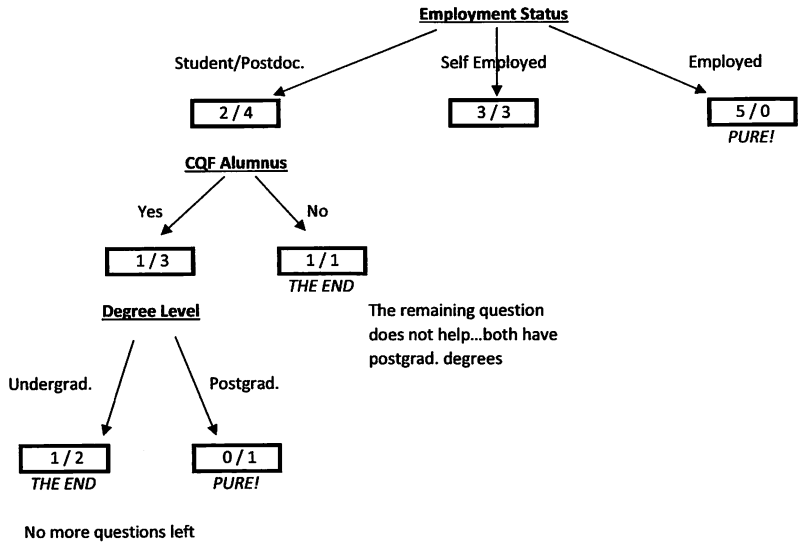


Figure 9.5: Moving further down the tree.

Moving to the CQF Alumnus Students/Postdocs we now look at the answers to the question about their highest degree. We see in Figure 9.5 that if they have a postgraduate degree then they will not be a subscriber. The split is pure (it's actually just a single, non-subscribing sample). If their highest degree is undergraduate then we can't be so sure, there's a one-in-three chance of them being a subscriber. But this is the end of the line, there are no more questions left. We have to accept the probabilistic result.

We can continue like this to fill in the rest of the tree, as shown in Figure 9.6. Luckily the rest of the tree results in pure splits.

If we are given a new data point, such as Self Employed with a Postgraduate Degree and who is a CQF Alumnus then we just run through our tree and we will see that such a person will be, like all the best people, a magazine subscriber. Although with so little data we would not be confident in that conclusion.

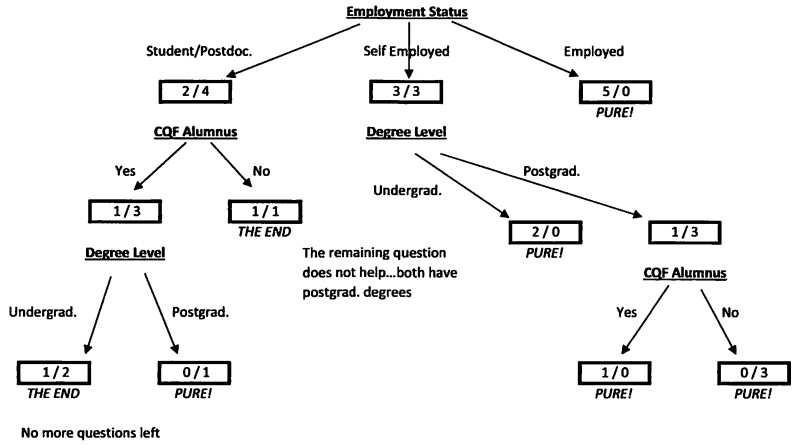


Figure 9.6: The final tree.

Some observations...

- Ideally you will end with a leaf that is a pure set, which classifies perfectly
- However if you have an impure set you might find that the remaining questions do not help to classify
- Or you might run out of questions
- You should always keep track of the numbers (in the boxes) because that will give you a probabilistic classification and a degree of confidence
- Notice how the same questions can appear at different places in the tree, although there will be no path along which you get asked the same question twice
- Questions don't have to be binary, yes/no
- And there don't have to be just two classifications (here whether or not they are magazine subscribers)

That's what decision trees look like and how they work. But you should be asking how did I construct the tree? Or more specifically how did I know in which order to ask the questions? Why did I start with Employment Status as the root, and not another attribute? Which attributes best split the data? There was a clue in my occasional use of the word information above. We are going to take inspiration from Information Theory.

9.4 Entropy

In layman's terms you want to find the attribute that gives you the highest information gain. I reckon that out of Employment Status, Highest Degree Level and whether or not a person is a CQF Alumnus the attribute that does the splitting the best, that gives the highest information gain, is Employment Status. But what makes me think that?

We need some way of looking at which attribute is best for splitting the data based on the numbers in each class. Referring to Figure 9.3 we need some numerical measure that gives us how efficient the split is based on the numbers 2 / 4, 3 / 3 and 5 / 0. But first we need to see what the Employment Status attribute is competing with. In Figure 9.7 we see the splits you get from having a root that is either CQF Alumnus or Highest Degree Level.

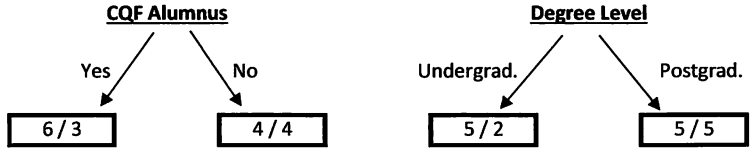


Figure 9.7: Other possible roots.

It looks like both of these splits do a very poor job. Both of them have one branch that is a coin toss. And in both cases the other branch is not much better. But we need to quantify this.

The uncertainty in classification is going to be measured by the entropy. We saw this, and the justification for it, in Chapter 2. All we need to do to determine the best attribute to start with, the root of our tree, is to measure the entropy for each attribute and choose the one with the lowest value. Usually this is done by measuring the information gain.

To measure the gain we need a starting point to measure gain relative to. And that would be the raw data before it is split. We calculate the entropy for the original data, that is ten magazine subscribers and seven non subscribers:

$$-\sum p\log_2(p) =$$

$$-\frac{10}{10+7}\log_2\left(\frac{10}{10+7}\right) - \frac{7}{10+7}\log_2\left(\frac{7}{10+7}\right)$$
$$= 0.977.$$

It could be worse, i.e. closer to 1, but not much.

Now we go to the data in Figures 9.3 and 9.7 and measure the entropy for each branch of each attribute.

For the Student/Postdoc. branch of Employment Status we have

$$-\sum p\log_2(p) = -\frac{2}{2+4}\log_2\left(\frac{2}{2+4}\right) - \frac{4}{2+4}\log_2\left(\frac{4}{2+4}\right) = 0.918.$$

For the Self Employed branch of Employment Status we have a rather obvious 1, since there is an equal number of subscribers and non. And for the Employed branch the entropy is zero since the split is pure.

Now we measure the average entropy for Employment Status as

$$\frac{6}{17}\times 0.918 + \frac{6}{17}\times 1 + \frac{5}{17}\times 0 = 0.677.$$

That's because six of the 17 are Student/Postdoc., another six of the 17 are Self Employed and five are Employed.

Thus the information gain thanks to this split is

$$\text{Information gain for Employment Status} = 0.977 - 0.677 = 0.300.$$

We do exactly the same for the Highest Degree Level and CQF Alumnus attributes:

$$\text{Information gain for Highest Degree Level} = 0.034,$$

$$\text{Information gain for CQF Alumnus} = 0.021.$$

And thus Employment Status having an information gain of 0.300 is the easy victor, beating Highest Degree Level and CQF Alumnus. And so it becomes the root of the decision tree.

Having established the root attribute we now repeat this process at each branch. For example we go to the left-most branch, the one labelled Student/Postdoc. in Figure 9.3 and consider which of the two remaining attributes will give the larger information gain for the seven cases on that branch. And we find it is CQF Alumnus. And so on.

The Decision Tree Algorithm

Step 1: Pick an attribute

Take one of the (remaining) attributes and branches and split the data.

Step 2: Calculate the entropy

For each branch calculate the entropy. Then calculate the average entropy over both/all branches for that attribute.
Return to Step 1 until all attributes have been examined.

Step 3: Set attribute to minimize entropy

Choose as the node the attribute that minimizes the entropy (or equivalently maximizes the information gain relative to the unsplit data).
Move down/across the tree and return to Step 1.

This algorithm is known as ID3, ID standing for Iterative Dichotomiser. There are other algorithms one can try. For example C4.5 is an algorithm created by the inventor of ID3. One of the problems addressed by C4.5 concerns how the algorithm deals with how many branches come from each attribute. In my example I have two attributes that split into two branches, this is a binary classification, and one attribute that has three branches, there are three answers to the question. How does this affect entropy? It's easy to see that the more branches leaving a node the more likely it is to be favoured, the entropy will be smaller. Just take the above example and give every line of data the name of the `wilmott.com` member, Andrea, Bruce, Charles, David,... If we use the name in the classification then each data point will get its own leaf and the entropy will be a perfect zero. But it almost certainly won't be much use when a Zachary comes along.

A simple modification to allow for the number of branches is to divide the information gain by another entropy measure, the split entropy, defined by

$$-\sum \frac{N_i}{N} \log_2 \left(\frac{N_i}{N} \right).$$

Where the sum is over all the relevant branches, N is the number of samples in the parent node and N_i is the number of samples at the end of each branch. So for the data in Figure 9.3 we would divide the information gain,

0.300, by

$$-\frac{6}{17}\log_2\left(\frac{6}{17}\right)-\frac{6}{17}\log_2\left(\frac{6}{17}\right)-\frac{5}{17}\log_2\left(\frac{5}{17}\right)=1.58.$$

This gain ratio allows for different numbers of partitions.

There are also different measures of uncertainty. A common one, besides entropy, is Gini impurity. Gini impurity measures how often something would be incorrectly labelled if that labelling was random. If we have probabilities p_i for being in set i then the probability of mislabelling is $1-p_i$. The Gini impurity is the average of this over all elements in the set:

$$\sum p_i(1-p_i)=1-\sum p_i^2.$$

9.5 Overfitting And Stopping Rules

If you have many attributes it is possible that by the time you get down to the leaf level there are very few records. The danger here is that your results are not representative of other, non-training, data. In other words, you have overfitted and all that your decision tree is doing is memorizing the data it has been given. To reduce the chance of this happening it is common to introduce stopping rules that say that when you get down to a certain number of records (either an absolute number or as a fraction of the whole dataset) you stop splitting the data and create a leaf. This way your results can remain statistically significant. This just means that you end up with a reliable probabilistic classification rather than an unreliable, possibly deterministic, one.

This is also the classic problem of fitting training data well, but doing badly on test data.

A stopping rule can also help when you have a new data point that you want to classify but there was no data point in the training set with the same features.

9.6 Pruning

Another, similar, method for avoiding overfitting is to prune your tree. You might find that part of the tree is unreliable, perhaps because you have tried classifying data from a validation set that you had kept back. In that case you just cut off the most unreliable nodes, replacing them with a leaf. The leaf would then either be classified according to whichever classification has the most samples in that leaf, or left as a probabilistic classification.

9.7 Numerical Features

What can we do if the answers to our questions are not categories but numerical? Suppose that we have data for people's heights, and whether or not they are magazine subscribers, such as that shown in Figure 9.8. The height data here is entirely made up but to make things interesting I have naturally assumed that magazine subscribers tend to be taller, and more attractive.

ID	Employment Status	Degree Level	CQF Alumnus	Heights	Wilmott Magazine Subscriber
1	Self Employed	Postgraduate	No	174.6	No
2	Self Employed	Postgraduate	Yes	173.0	Yes
3	Employed	Postgraduate	Yes	185.2	Yes
4	Student/Postdoc.	Postgraduate	No	169.5	Yes
5	Student/Postdoc.	Undergraduate	Yes	179.9	Yes
6	Student/Postdoc.	Undergraduate	Yes	167.9	No
7	Employed	Undergraduate	Yes	177.7	Yes
8	Self Employed	Postgraduate	No	175.7	No
9	Self Employed	Undergraduate	No	176.5	Yes
10	Student/Postdoc.	Undergraduate	Yes	162.5	No
11	Self Employed	Undergraduate	Yes	178.4	Yes
12	Employed	Postgraduate	Yes	178.2	Yes
13	Employed	Undergraduate	No	173.0	Yes
14	Student/Postdoc.	Postgraduate	Yes	174.7	No
15	Employed	Postgraduate	No	188.0	Yes
16	Student/Postdoc.	Postgraduate	No	163.2	No
17	Self Employed	Postgraduate	No	159.5	No

Figure 9.8: Same data as before but with added (fictitious) height information.

One simple way of tackling this classification problem within a decision tree is to choose a threshold s for the height so that being above or below that threshold determines the branch. The level of s can be chosen to maximize the entropy gain. With this data, and if we used the height as the root attribute, then the entropy gain is maximized by a threshold of 176cm.

In principle you could have something more complicated than a simple threshold for determining the branch.

Now you can mix categories and numerical data and all of the above explanation for building your decision tree carries over.

9.8 Regression

In the above I have focused on using decision trees for classification problems. However the method can also be used for regression.

When used for regression our goal is to come up with a numerical value or prediction based on features that can be either categories or numbers. How much is the car worth give its model (category), manual or automatic (category), age (numerical), mileage (numerical), etc.

Having both categories and numerical data for our features will not cause a problem, we just use the threshold method described above (or something similar but more complicated) to turn numerical feature data (mileage) into categories (above/below 40,000 miles, say).

However we do need to use a measure other than entropy to tell us how informative our different types of data are. This is because our dependent variables are no longer classes (yes/no to magazine subscription) but are numerical (£6,250).

ID	Age	Seat	Transmission	Mileage	Price
1	6	2	Manual	19,042	£ 3,150
2	5.5	3	Automatic	12,851	£ 6,400
3	6	5	Manual	36,282	£ 5,300
4	6	5	Manual	16,621	£ 5,250
5	6	5	Manual	35,685	£ 5,700
6	5	3	Manual	27,314	£ 5,050
7	6	5	Manual	42,143	£ 5,000
8	6	3	Manual	42,780	£ 3,650
9	5	3	Manual	37,413	£ 4,950
10	6	5	Automatic	74,848	£ 4,850
11	4.5	5	Manual	48,572	£ 6,400
12	5	3	Manual	28,602	£ 4,900
13	5	5	Manual	10,663	£ 5,950
14	5	5	Manual	56,775	£ 5,250
15	7	5	Manual	63,000	£ 2,600
16	4	5	Automatic	34,303	£ 7,150
17	5	5	Automatic	25,584	£ 5,900
18	6	5	Manual	9,499	£ 6,250

Figure 9.9: Auction prices for Peugeot Partner Tepee.

Let's look at the data shown in Figure 9.9. Here we have recent auction prices for several Peugeot Partner Tepees (no, me neither, but I think they are cars or vans) together with various features, some are categories such as transmission type and some are numerical, mileage for example. I have tweaked this data only a tiny amount and simply to illustrate several things that can happen in building the tree.

The dependent variable will be $y^{(n)}$, representing the value of the n^{th} car, say. For each attribute we are going to look at the sum of squared error across the branches. But, crucially, each branch will have a different model for the forecast.

To see what I mean let's start with perhaps the simplest attribute, whether the car is manual or automatic transmission.

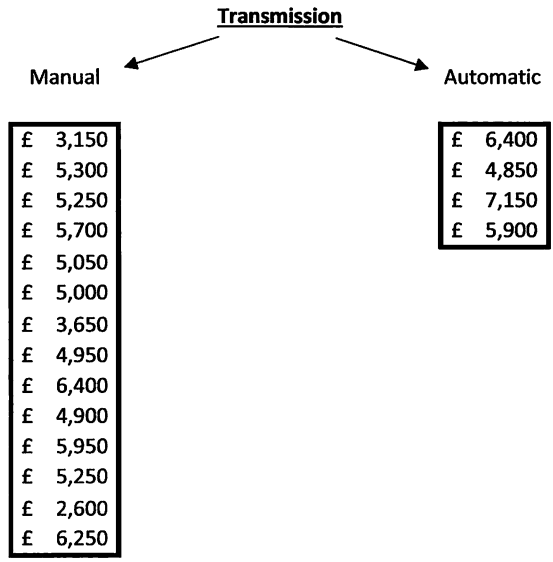


Figure 9.10: The transmission attribute and the data.

In Figure 9.10 we see how the car prices depend on the type of transmission. If the car is manual then the average price is £4,957, and £6,075 if automatic. If we had no other information (such as mileage, age, etc.) so that this was the end of the branches then those would be our forecasts for any other Peugeot Partner Tepee we might come across. I.e. the model we use is the average at each leaf, but it's a different average for each leaf. We'll see something a bit more sophisticated than the average shortly.

But how good is that attribute at predicting compared with say predicting price based on number of seats or mileage? We need to know this if we are build our tree most efficiently. To figure that out we need something to replace entropy. And the easiest quantity to use is the sum of squared errors:

$$\sum_{\text{Manual}} \left(\bar{y}_{\text{Manual}} - y^{(n)}\right)^2 + \sum_{\text{Auto}} \left(\bar{y}_{\text{Auto}} - y^{(n)}\right)^2.$$

This is to be interpreted as the sum of the squared differences between each individual price and the average *for that category* over all the manual-transmission cars plus the same for automatic transmission. This is a simple standard-deviation type of measure. In other words

$$(3150-4957)^2+(5300-4957)^2+\cdots+(6250-4957)^2$$
$$+(6400-6075)^2+\cdots+(5900-6075)^2=18,916,785.$$

But how good is that, i.e. how low, compared to a similar calculation for, say, the mileage of the car? Since mileage is a numerical quantity we introduce a threshold.

Let's go over to just symbols. We want to calculate

$$\min_s\left(\min_y\sum_{x_m^{(n)}<s}\left(y-y^{(n)}\right)^2+\min_y\sum_{x_m^{(n)}\geq s}\left(y-y^{(n)}\right)^2\right). \tag{9.1}$$

And now this is interpreted as follows. The threshold for the mileage, say, is s. And we are looking at prices for cars under and over the mileage of s. $x_m^{(n)}$ means the numerical value of the m^{th} attribute (mileage) of the n^{th} data point (car). The minimization over y just means each branch has its own model i.e. value. Rather obviously the minimization over y will result in the value for each y being the average of values on that branch. And then finally we want to find the threshold that gives us the best split of the data.

Let's look at the example.

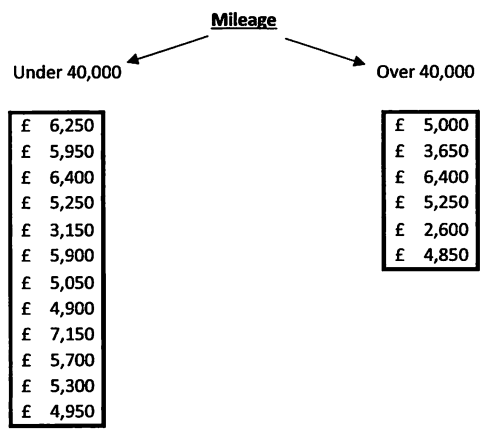


Figure 9.11: Mileage attribute, threshold 40,000, and the data.

In Figure 9.11 we see the data divided according to the mileage attribute with the threshold of 40,000. The mean value (the y) for under 40,000 miles is £5,496 and £4.625 for above. The sum of squared errors, as in expression (9.1), is 19,771,041. This is worse than using the transmission type for regression. But that's because I just picked 40,000 at random. We can minimize this error by choosing a threshold of 60,000 when the squared error becomes 17,872,343. This is now better than using transmission but not by much.

Anyway, we can continue this process by looking for the best attribute to be the root of our decision tree. And then work our way down the tree looking at the remaining attributes exactly as in the classification problem.

To summarize: Choose an attribute, m, and for that attribute find a threshold s to minimize (9.1). The m gives us which feature to choose and the s tells us where the split is.

I don't know whether this will help or confuse but there are three minimizations of the standard deviation happening here:

1. There is the minimization within each node, finding the best model. Here this is trivially the mean of the data in each node

2. Then there is the minimization by finding the best split

3. Then we choose which feature gives us the lowest error and this becomes the one we use to move further down the tree

If we have a classification problem then we don't need the first of these.

If our features are non numeric then we don't need to do the second of these.

Linear regression

We have possibly thrown away quite a lot of information by just using a prediction that is the mean on that branch, or in the leaf. If a numerical attribute divides into leaves then we could relate the forecast to the numerical values of that attribute.

Let's look at car price versus age now. Our data is just that in Figure 9.12 that hasn't been greyed out.

ID	Age	Seat	Transmission	Mileage	Price
1	6	2	Manual	19,042	£ 3,150
2	5.5	3	Automatic	12,851	£ 6,400
3	6	5	Manual	36,282	£ 5,300
4	6	5	Manual	16,621	£ 5,250
5	6	5	Manual	35,685	£ 5,700
6	5	3	Manual	27,314	£ 5,050
7	6	5	Manual	42,143	£ 5,000
8	6	3	Manual	42,780	£ 3,650
9	5	3	Manual	37,413	£ 4,950
10	6	5	Automatic	74,848	£ 4,850
11	4.5	5	Manual	48,572	£ 6,400
12	5	3	Manual	28,602	£ 4,900
13	5	5	Manual	10,663	£ 5,950
14	5	5	Manual	56,775	£ 5,250
15	7	5	Manual	63,000	£ 2,600
16	4	5	Automatic	34,303	£ 7,150
17	5	5	Automatic	25,584	£ 5,900
18	6	5	Manual	9,499	£ 6,250

Figure 9.12: We will do a regression on this data.

A simple splitting of the data into two branches using a threshold of 6.2 years gives the lowest sum of squared errors at 15,616,176. (This happens to be our best error yet, so really ought to be the root.) But we are lumping together a lot of data points and throwing out any nuance. You'll see that nuance now when I plot value against age.

What if instead of splitting the data at a threshold we fit a straight line to the data? We'd get the results in Figure 9.13 and an error of 12,457,500.

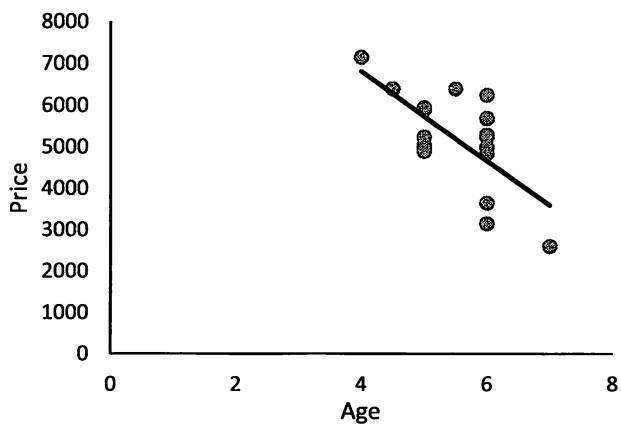


Figure 9.13: A straight line fitted to the data.

And finally, in the spirit of *both* splitting and linear regression, split at 5.2 years and fit *two* straight lines, a different one above and below the split, gives the results shown in Figure 9.14, with an error of just 8,609,273. Of course this is no more than a more complicated fitting function than a straight line.

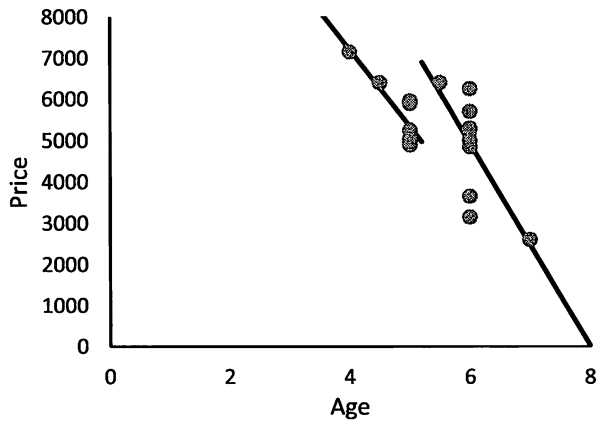


Figure 9.14: Splitting plus a linear fit.

You should see what I mean now, by fitting with a straight line or lines I've kept more information from the data and got a better forecast.

You can get more complicated by making decisions based on (linear perhaps) combinations of features. (Age minus some parameter times the number of seats.)

9.9 Looking Ahead

What I have described is a greedy algorithm, one where at each stage we choose which feature to split on based on a measure (entropy or standard deviation) at that point in the tree. There is no consideration for what will follow further down the tree. But this could be a problem, it might lead to sub-optimal splitting.

We sort of saw this in the car example. Right at the end I found that a linear fit to the age data gave the lowest standard deviation. But what if I'd split on, say, mileage higher up the tree? Or just split on age but using a threshold and a constant value (the mean) in the node? And then I'd gone on to split on transmission etc. further down. If I'd done that then I would

have missed out the rather good linear fit to age.

One can prevent this by looking ahead to see what happens to splits further down the tree i.e. the sub-trees that are produced.

9.10 Bagging And Random Forests

Decision trees can have a tendency to overfit. Think of the rituals that a footballer might go through before a match based on experiences from the past when he won or lost. Lucky socks? Check. Leave dressing room left foot first? Check. No curry the night before? Check. To reduce overfitting and variance in predictions one can use various methods for aggregating results over many trees.

Bootstrap aggregation or bagging for short is a method in which one samples, with replacement, from a full data set and then applies whatever machine-learning technique you are using to get a forecast, and then repeats with another random sample. One does this many times and the final forecast will be either the majority vote in a classification problem or an average for a regression problem.

Random forests is the same idea with one twist. In deciding which node to use for a split you don't look at the best feature for splitting over *all* possible features. Instead you consider a *random subset* of features and choose the best feature from that subset.

Further Reading

The self-published *Decision Trees and Random Forests: A Visual Introduction For Beginners* by Chris Smith is cheap and cheerful. Very much for beginners. There are other machine-learning books in the same series.

Another self-published book *Tree-based Machine Learning Algorithms: Decision Trees, Random Forests, and Boosting* by Clinton Sheppard is light on the mathematics but does have plenty of Python code.

Chapter 10

Neural Networks

10.1 Executive Summary

A neural network is a type of machine learning that is meant to mimic what happens in the brain. Signals are received by neurons where they are mathematically manipulated and then passed on to more neurons. The input signal might pass through several layers of neurons before being output, as a forecast regression or classification. It can be used for either supervised or unsupervised learning.

10.2 What Is It Used For?

Neural networks are used for

- Modelling complex relationships between inputs and outputs
- Example: Recognising images including handwriting
- Example: Enhancing grainy images
- Example: Translating from one language to another
- Example: Advertising. If advertisers can find a way to use neural networks they will!

10.3 A Very Simple Network

You will have seen pictures like Figure 10.1, representing a typical neural network. This is a good illustration of the structure of a very simple

feedforward network. Inputs go in the left of this picture and outputs come out of the right. In between, at each of the nodes, there will be various mathematical manipulations of the data. It's called feedforward because the data and calculations go in one direction, left to right. There is no feedback or recurrence like we'll be seeing later.

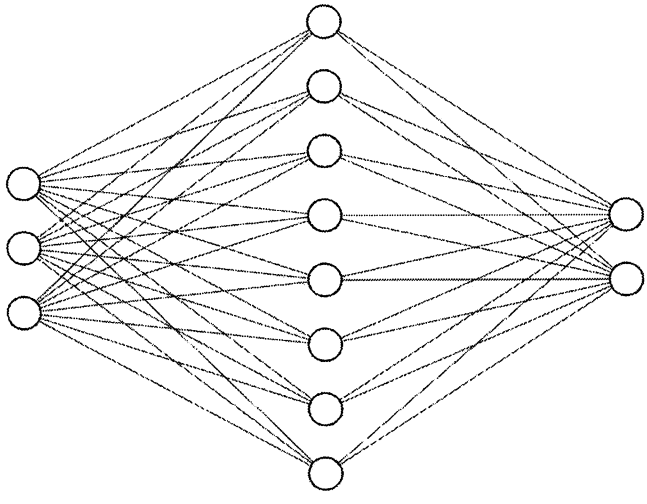


Figure 10.1: A typical neural network. Here there is an input layer, an output layer and one hidden layer in between.

In this particular example we have an input being an array or vector with three entries, so three *numerical* quantities. These are manipulated in the hidden layer of eight nodes before being passed out to an output vector/array with two entries. This would be an example of trying to predict two quantities from a three-dimensional input. The outputs will be numerical also but these can still be used for classification.

I'll explain the mathematical manipulations shortly. But let me now just say that there doesn't have to be a single hidden layer. You can have as many as you want or need, and each with as many nodes as you want or need. However, the number of input and output nodes will depend on what data you have and what you want to forecast. The layout of the network is called the architecture.

10.4 Universal Approximation Theorem

The Universal Approximation Theorem says that, under quite weak conditions, as long as you have enough nodes then you can approximate any continuous function with a single-layer neural network to any degree of accuracy. The neural-network architecture for this simple task is shown in Figure 10.2. There is a single input, think x, and a single output, think y. To get a better approximation you will just need more nodes in the hidden layer.

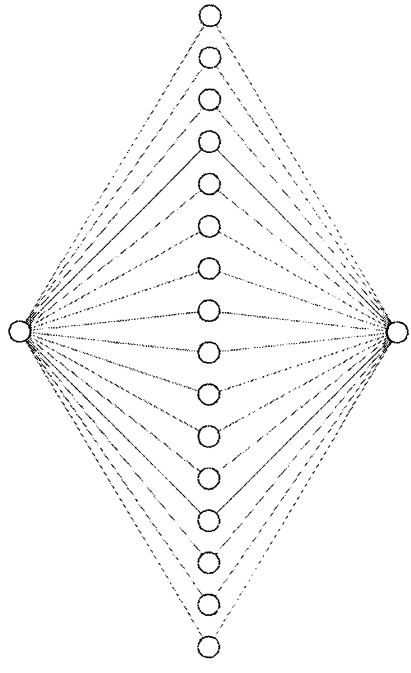


Figure 10.2: A neural network to represent the Universal Approximation Theorem.

This is one of the most important uses for a neural network, function approximation. But usually our problems are not as straightforward as this implies. Instead of having a single independent variable, the x, we will typically have many, a whole array. And instead of a single output, the y, there could be many outputs. And most importantly instead of having a known function that we want to approximate we have a whole load of data, the inputs and the outputs, and we want our network to figure out what happens at the nodes to give the best approximation, one that can be used on new, unseen, data. With such complicated problems we'll need the richer

structure that you can get by having more than one hidden layer.

Ok, now it's time to tell you what goes on in the hidden layer, what are the mathematical manipulations I've mentioned.

10.5 An Even Simpler Network

Figure 10.3 shows just about the simplest network, and one that can still be used to describe the manipulations going on.

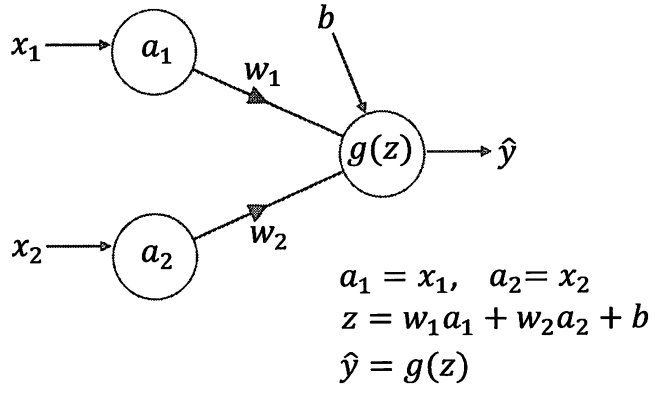


Figure 10.3: The manipulations in the classical neural network.

Here we have two inputs on the left, x_1 and x_2, and a forecast/output on the right, $\hat{y}$.

The inputs are passed unchanged into the first nodes:

$$a_1 = x_1 \quad \text{and} \quad a_2 = x_2.$$

Then each of these node values is multiplied by a weight, the ws. And then a bias, b, is added:

$$z = w_1 a_1 + w_2 a_2 + b.$$

That is, just a linear combination.

This result is then acted on by a function $g(z)$ to give the output,

$$\hat{y} = g(z).$$

This is a very, very simple transformation of the data, here used to give some function of a two-dimensional input.

The function $g(z)$ will be chosen, it is called an activation or transfer function. In neural networks we specify the activation function but the weights, the ws, and the bias, b, will be found numerically in order to best fit our forecast output with our given data.

10.6 The Mathematical Manipulations In Detail

Now let's do that with a larger network, showing all the sub- and superscripts we'll be needing.

In Figure 10.4 is a neural network with a single hidden layer. For more layers the following is no different. Just keep an eye out for the superscript denoting which layer we are looking at.

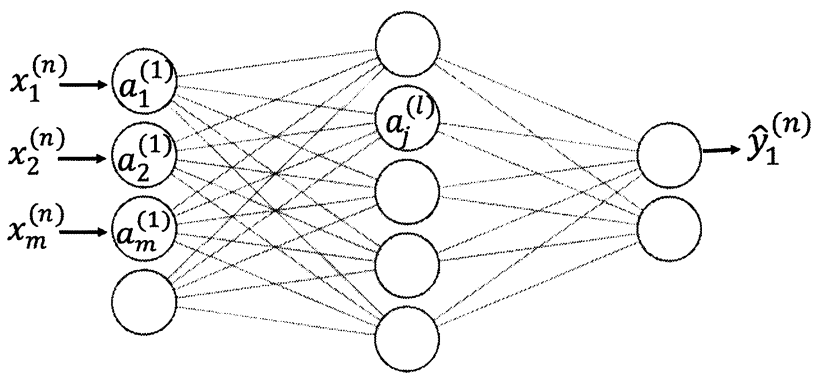


Figure 10.4: The quantities being input, output and in the nodes.

Notation

I show three types of quantities in the figure. The inputs are the xs. The outputs are the $\hat{y}$s. The values in each of the nodes are the as. Let's look at the sub- and superscripts, etc., and what they mean.

As before the superscript (n) refers to the n^{th} data point. The subscript m on x is the position in the vector of features, of which there will be M. So our inputs will be vectors $\mathbf{x}^{(n)}$.

These quantities are then input into the first node. The input layer of nodes contain values $a_m^{(1)}$, as shown by the direction of the arrows. We put the input values, the xs, unchanged into the first layer:

$$a_m^{(1)} = x_m.$$

The nodes in the next layers have the values $a_j^{(l)}$. Here l represents the number of the layer, out of L layers, and the j represents the specific node. Note that the number of nodes in each hidden layer can be anything. However the input layer has the same number of nodes as features, and the output layer the number of nodes needed for our forecast.

We continue like this until the output. The (n) means the output associated with the n^{th} data point. That will be another vector $\hat{\mathbf{y}}^{(n)}$, with usually a different dimension from that of the input vector. And the hat in this just means that this is the forecast value, as opposed to the actual value for the dependent quantity from the training data.

As I showed above, two things happen to the numerical quantities as we go through the network.

Propagation

In going from one layer to the next the first thing that happens is a linear combination of the values in the nodes. Figure 10.5 is similar to Figure 10.3 except with more nodes and sub- and superscripts everywhere.

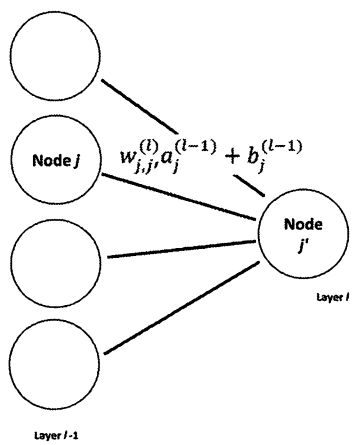


Figure 10.5: A detail of the network and a formula.

I have labelled the two layers as $l-1$ and l. Also notice that the general node in the left-hand layer is labelled j and one in the right-hand layer, layer l, is labelled j'.

We want to calculate what value goes into the j'^{th} node of the l^{th} layer.

First multiply the value $a_j^{(l-1)}$ in the j^{th} node of the previous, $(l-1)^{\text{th}}$, layer by the parameter $w_{j,j'}^{(l)}$ and then add another parameter $b_{j'}^{(l)}$. Then we add up all of these for every node in layer $l-1$.

This is just

$$\sum_{j=1}^{J_{l-1}} w_{j,j'}^{(l)} a_j^{(l-1)} + b_{j'}^{(l)} \tag{10.1}$$

where J_l means the number of nodes in layer l. I'll call this expression $z_{j'}^{(l)}$.

This is a bit hard to read, such tiny fonts, and anyway it's much easier to write and understand as

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}, \tag{10.2}$$

with the matrix $\mathbf{W}^{(l)}$ containing all the multiplicative parameters, i.e. the weights $w_{j,j'}^{(l)}$, and $\mathbf{b}^{(l)}$ is the bias. The bias is just the constant in the linear transformation. (Sometimes the bias is represented by an extra node at the top of the layer, containing value 1, with lines coming down to the next layer. This way of drawing the structure is exactly equivalent to what we have here.)

This just means that the first manipulation, to get to the $\mathbf{z}$, is to take a linear combination of values in the previous layer. But remember, we don't specify what the weights and biases are, they are to be found during the training of the network.

If that were it then it wouldn't be interesting. But we still have to perform the other simple transformation.

The activation function

The activation function gets its name from a similar process in physical, i.e. inside the brain, neurons whereby an electrical signal once it reaches a certain level will fire the neuron so that the signal is passed on. If the signal is too small then the neuron does not fire.

Applying the same idea here we simply apply a function to expressions (10.1) or (10.2). Let's call that function $g^{(l)}$. It will be the same for all nodes in a layer but can differ from layer to layer. And we *do* specify this function.

Thus we end up with the following expression for the values in the next

layer

$$\mathbf{a}^{(l)} = g^{(l)}(\mathbf{z}^{(l)}) = g^{(l)}\left(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}\right). \tag{10.3}$$

The function of a vector just means taking the function of each entry.

So a signal is passed from all the nodes in one layer to the next layer where it goes through a function that determines how much of the signal to pass onwards.

And so on, all the way through the network, up to and including the output layer which also has an associated activation function.

This can be interpreted as a regression on top of a regression on top of a ...

You can see that what comes out of the right is just a messy function of what goes in the left. I use messy in the technical mathematical sense that if you were to write the scalar $\hat{y}$s explicitly in terms of the scalar xs then it wouldn't look very pretty. It would be a very long expression, with lots of sub- and superscripts, and summations. But a function it most definitely is.

Classification problems

I've mentioned this before, but it's worth repeating, classification is different from regression and function fitting.

Suppose you want to classify fruit. You have peaches, pears, plums, etc. Your raw data for the $\mathbf{x}$s might be numerical quantities representing dimensions, shape, colour, etc. But what will the dependent variable(s) be? You could have a single dependent y which takes values 1 (for peach), 2 (for pear), etc. But that wouldn't make any sense when you come to predicting a new out-of-sample fruit. Suppose your output prediction was $\hat{y} = 1.5$. What would that mean? Half way between a peach and a pear perhaps? That's fine if there is some logical ordering of the fruit so that in some sense an peach is less than a pear, which is less than a plum, and so on. But that's not the case.

It makes more sense to output a vector $\hat{\mathbf{y}}$ with as many entries as there are fruit. The input data would have a peach as being $(1, 0, \ldots, 0)^T$, a pear as $(0, 1, \ldots, 0)^T$ and so on. An output of $(0.4, 0.2, \ldots, 0.1)^T$ would then be useful, most likely a peach but with some pear-like features.

10.7 Common Activation Functions

Here are some of the most common activation functions.

Linear function

Not interesting would be

$$g(x) = x.$$

There'll be problems with this activation function when it comes to finding the parameters because its gradient is one everywhere and this can cause problems with gradient descent. And it also rather misses the essential non-linear transformation nature of neural networks.

Step function/Hard limit

The step function behaves like the biological activation function described above.

$$g(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}.$$

The signal either gets through as a fixed quantity, or it dies. This might be a little bit too extreme, leaving no room for a probabilistic interpretation of the signal for example. It also suffers from numerical issues to do with having zero gradient everywhere except at a point, where it is infinite. This messes up gradient descent again. Probably best to avoid.

Positive linear/ReLU function

$$g(x) = \max(0, x).$$

ReLU stands for Rectified Linear Units. It's one of the most commonly used activation functions, being sufficiently non linear to be interesting when there are many interacting nodes. The signal either passes through untouched or dies completely. (Everyone outside machine learning calls it a ramp function.)

Saturating linear function

$$g(x) = \begin{cases} 0 & x < 0 \\ x & 0 \leq x \leq 1 \\ 1 & x > 1 \end{cases}.$$

Similar to the step function but not so extreme.

Sigmoid or logistic function

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

This is a gentler version of the step function. And it's a function we have found useful previously. It could be a good choice for an activation function if you have a classification problem.

The tanh function can also be used, but this is just a linear transformation of the logistic function.

Softmax function

We saw the softmax function in Chapter 2. The softmax function takes an array of K values $(z_1, z_2, \ldots, z_K)$ and maps them onto K numbers between zero and one, and summing to one. It is thus a function that turns several numbers into quantities that can be perhaps interpreted as probabilities.

$$\frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}.$$

It is often used in the final, output, layer of a neural network, especially with classification problems.

Hidden layer versus output layer

One can be quite flexible in choosing activation functions for hidden layers but more often than not the activation function in the final, output, layer will be pretty much determined by your problem.

10.8 The Goal

Our goal is to ultimately fit a function. But I haven't yet said much about the function we are fitting. Typically it won't be the sine function we'll be seeing in a moment as our first example. It will come from data. For each input independent variable/data point of features $\mathbf{x}^{(n)}$ there will be a corresponding dependent vector $\mathbf{y}^{(n)}$. This is our training data.

Our neural network on the other hand takes the $\mathbf{x}^{(n)}$ as input, manipulates it a bit, and throws out $\hat{\mathbf{y}}^{(n)}$. Our goal is to make the $\mathbf{y}^{(n)}$ and $\hat{\mathbf{y}}^{(n)}$ as

close to each other as possible. And we do that by choosing the parameters $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$, for each layer.

This is just a fancy regression. And so you would rightly expect some discussion of cost functions and numerical methods. Soon.

10.9 Example: Approximating a function

Let's look at the Universal Approximation Theorem in action. Take a neural network with one hidden layer and a variety of numbers of nodes in that layer and use it to fit $\sin(2x)$. In Figure 10.6 we see how well we can fit this function using two, four, six and eight nodes.

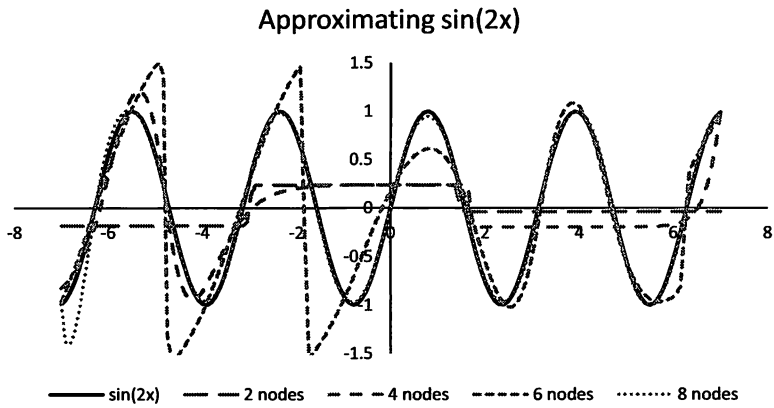


Figure 10.6: Approximating a sine wave with a one-hidden-layer network.

I used a sigmoidal activation function in the hidden layer. Obviously the more nodes we have the better. With eight nodes the fit is excellent.

How did I derive these results? Well, I confess that since the network layout, the architecture, here is quite simple I used Excel and Solver, to find the weights and the biases. And then I checked it using the NeuroLab Python library, see `https://pythonhosted.org/neurolab/`.

But that is cheating, according to the premise of this book. So what is going on inside the code that finds the parameters? First we need to decide on a cost function.

10.10 Cost Function

As in simpler forms of regression we need to have a measure of how good a job our algorithm is doing at fitting the data. Thus we need a cost function.

A common one, and the one I used in the sine fitting above, is ordinary least squares. The cost function is then

$$J = \frac{1}{2} \sum_{n=1}^{N} \sum_{k=1}^{K} \left(\hat{y}_k^{(n)} - y_k^{(n)} \right)^2 . \tag{10.4}$$

The notation is obvious, I hope. $y_k^{(n)}$ is the dependent data for the n^{th} data point, the k representing the k^{th} node in the output vector, and $\hat{y}_k^{(n)}$ is similar but for the forecast output.

(In the above sine wave example we only had one output so $K = 1$.)

Classification

In classification problems we still associate our classes with numbers or vectors. If we are labelling emails as spam or not we might label non-spam emails as 0 and spam emails as 1. That would be a binary classification, and would require a single output.

If we have types of animal such as mammal, reptile, amphibian, etc. then we use the vector approach, using a vector with dimension the same as the number of classes, just as described above for fruit, $(1, 0, \ldots, 0)^T$, etc.

The cost function commonly used for such an output is similar to the one we used in Chapter 6 for logistic regression. And this is, for a binary, yes/no, classification

$$J = -\sum_{n=1}^{N} \left(y^{(n)} \ln \left(\hat{y}^{(n)} \right) + (1 - y^{(n)}) \left(1 - \ln \left(\hat{y}^{(n)} \right) \right) \right) .$$

But if we have three or more classes then we have to sum over all of the K outputs, K being the number of classes:

$$J = -\sum_{n=1}^{N} \sum_{k=1}^{K} \left(y_k^{(n)} \ln \left(\hat{y}_k^{(n)} \right) + (1 - y_k^{(n)}) \left(1 - \ln \left(\hat{y}_k^{(n)} \right) \right) \right) . \tag{10.5}$$

This is related to cross entropy again.

To these cost functions could be added a regularization term, as discussed previously, of the form

$$\frac{\lambda}{2} |\mathbf{W}|^2 .$$

So we now have something to minimize, but how do we do that numerically?

10.11 Backpropagation

We want to minimize the cost function, J, with respect to the parameters, the components of $\mathbf{W}$ and $\mathbf{b}$. To do that using gradient descent we are going to need the sensitivities of J to each of those parameters. That is we want

$$\frac{\partial J}{\partial w_{j,j'}^{(l)}} \quad \text{and} \quad \frac{\partial J}{\partial b_{j'}^{(l)}}.$$

If we can find those sensitivities then we can use a gradient descent method for finding the minimum. But this is going to be much harder here than in any other machine-learning technique we have encountered so far. This is because of the way that those parameters are embedded within a function of a function of a... To find the sensitivities requires differentiating that function of a function of a... And that's going to be messy, involving repeated use of the chain rule for differentiation, unless we can find an elegant way of presenting this. Fortunately we can.

And this is made relatively painless by introducing the quantity

$$\delta_{j'}^{(l)} = \frac{\partial J}{\partial z_{j'}^{(l)}}.$$

Remember what z is, it's the linear transformation of the values in the previous layer, but *before* going into the activation function.

This idea is called backpropagation. Backpropagation is rather like calculating the error between the $\mathbf{y}$ and the $\hat{\mathbf{y}}$ in the output layer and assigning that error to the hidden layers. So the error in effect propagates back through the network. This is slightly strange because although there is an obvious meaning for the error in the output layer (it's just the difference between the actual value of $\mathbf{y}$ and the forecast value $\hat{\mathbf{y}}$) there is no correct $\mathbf{a}$ in the hidden layer with which to make a comparison. But that doesn't matter. Backpropagation is going to tell us all we need in order to find our parameters.

I'm sorry but we're going to have another one of those mini networks. See Figure 10.7. This shows pretty much all we need in the following. Our first goal is to find the sensitivity of the cost function to $w_{j,j'}^{(l)}$.

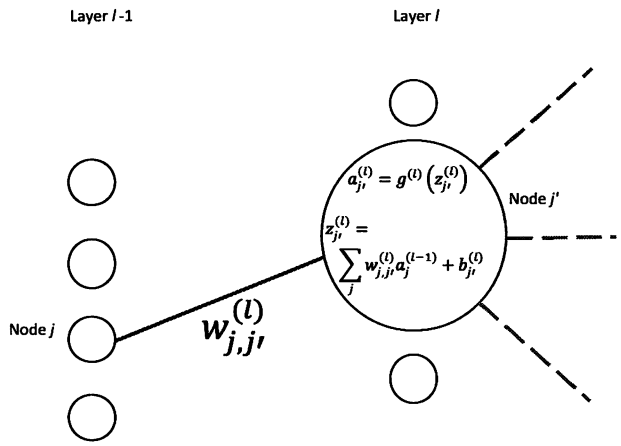


Figure 10.7: The network showing the key formulæ.

Let's play around with the chain rule:

$$\delta_j^{(l-1)} = \frac{\partial J}{\partial z_j^{(l-1)}} = \sum_{j'} \frac{\partial J}{\partial z_{j'}^{(l)}} \frac{\partial z_{j'}^{(l)}}{\partial z_j^{(l-1)}}.$$

But

$$z_{j'}^{(l)} = \sum_j w_{j,j'}^{(l)} a_j^{(l-1)} + b_{j'}^{(l)} = \sum_j w_{j,j'}^{(l)} g^{(l-1)}\left(z_j^{(l-1)}\right) + b_{j'}^{(l)}.$$

And so

$$\begin{aligned}\delta_j^{(l-1)} &= \left.\frac{dg^{(l-1)}}{dz}\right|_{z_j^{(l-1)}} \sum_{j'} \frac{\partial J}{\partial z_{j'}^{(l)}} w_{j,j'}^{(l)} \\ &= \left.\frac{dg^{(l-1)}}{dz}\right|_{z_j^{(l-1)}} \sum_{j'} \delta_{j'}^{(l)} w_{j,j'}^{(l)}. \end{aligned} \tag{10.6}$$

Let's see what we have achieved so far, and what we haven't. Equation (10.6) shows us how to find the δs in a layer if we know the δs in all layers to the *right*. Fantastic.

But what is so great about these δs? That's actually the easy bit:

$$\begin{aligned}\frac{\partial J}{\partial w_{j,j'}^{(l)}} &= \frac{\partial J}{\partial z_{j'}^{(l)}} \frac{\partial z_{j'}^{(l)}}{\partial w_{j,j'}^{(l)}} \\ &= \delta_{j'}^{(l)} a_j^{(l-1)}. \end{aligned} \tag{10.7}$$

And that's done it! The sensitivity of the cost function, J, to the ws can be written in terms of the δs which in turn are backpropagated from the network layers that are just to the right, one nearer the output.

And sensitivity of the cost function to the bias, b? That's even easier,

$$\frac{\partial J}{\partial b_{j'}^{(l)}} = \delta_{j'}^{(l)}.$$

In the above we have the derivative of the activation function with respect to z. This will be a simple function of z depending on which activation function we use. If it is ReLU then the derivative is either zero or one. If we use the logistic function then we find that $g'(z) = g(1-g)$, rather nicely.

The last hidden layer is different because it feeds into the output. If the cost function is quadratic, for example, then we have instead

$$\delta_j^{(L)} = \left.\frac{dg^{(L)}}{dz}\right|_{z_j^{(L)}} (\hat{y}_j - y_j).$$

(To avoid confusion, if there is a single output then you can drop the j subscripts.)

The Backpropagation Algorithm

Step 0: Initialize weights and biases

Choose the weights and biases, typically at random. The size of the weights should decrease as the number of nodes increases.

Step 1: Pick one of the data points at random

Input the vector $\mathbf{x}$ into the left side of the network, and calculate all the zs, as, etc. And finally calculate the output $\hat{y}$. (This might also be a vector.)

Step 2: Calculate contribution to the cost function

You don't actually need to know the actual value of the cost function to find the weights and biases but you will want to calculate it so you

can monitor convergence.

Step 3: Starting at the right, calculate all the δs

For example, if using the quadratic cost function in one dimension, then

$$\delta^{(L)} = \left.\frac{dg^{(L)}}{dz}\right|_{z_j^{(L)}} (\hat{y} - y) .$$

Moving to the left

$$\delta_j^{(l-1)} = \left.\frac{dg^{(l-1)}}{dz}\right|_{z_j^{(l-1)}} \sum_{j'} \delta_{j'}^{(l)} w_{j,j'}^{(l)} .$$

Step 4: Update the weights and biases using (stochastic) gradient descent

$$\text{New } w_{j,j'}^{(l)} = \text{ Old } w_{j,j'}^{(l)} - \beta \frac{\partial J}{\partial w_{j,j'}^{(l)}} = \text{ Old } w_{j,j'}^{(l)} - \beta \delta_{j'}^{(l)} a_j^{(l-1)}$$

and

$$\text{New } b_{j'}^{(l)} = \text{ Old } b_{j'}^{(l)} - \beta \frac{\partial J}{\partial b_{j'}^{(l)}} = \text{ Old } b_{j'}^{(l)} - \beta \delta_{j'}^{(l)} .$$

Return to Step 1.

10.12 Example: Character recognition

Now for a proper example. I am going to use a neural network to recognise handwritten characters. This is a good, robust test of the technique. It's also relatively easy, and hence commonly found in text books, because of the ease of access to data and there is sample Python code all over the internet.

The inputs, the xs, are just many examples of handwritten numbers from 0 to 9. Each of these is represented by a vector of dimension 784. That just means that the digits have been pixelated in a square of 28 by 28. Each entry is a number between 0 and 255, representing how grey each pixel is.

And where did I get this data from? There is a very famous data set, the Modified National Institute of Standards and Technology (MNIST) database of 60,000 training images and 10,000 testing images. These are

samples handwritten by American Census Bureau employees and American high school students. Easy-to-read versions of this data can be found at `http://yann.lecun.com/exdb/mnist/`.

Here's one example, in Figure 10.8, this is the first line of the raw MNIST training data. It is interpreted as follows. The first number in the top left corner, here 5, is the number represented. The rest of the numbers are the greyness of each pixel.

IPython console

Console 1/A

```
5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,3,18,18,18,126,136,175,26,166,255,247,127,0,0,0,0,0,0,0,0,0,0,0,0,30,3
6,94,154,170,253,253,253,253,253,225,172,253,242,195,64,0,0,0,0,0,0,0,0,0,0,0,49,238,253,253,2
53,253,253,253,253,253,251,93,82,82,56,39,0,0,0,0,0,0,0,0,0,0,0,0,18,219,253,253,253,253,253,1
98,182,247,241,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,80,156,107,253,253,205,11,0,43,154,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,14,1,154,253,90,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,139,253,190,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,11,190,253,70,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,35,241,225,160,108,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,81,240,253,253,119,25,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,45,186,253,253,150,
27,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,16,93,252,253,187,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,249,253,249,64,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,46,130,183,253
,253,207,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,39,148,229,253,253,253,250,182,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,24,114,221,253,253,253,253,201,78,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,23,6
6,213,253,253,253,253,198,81,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,18,171,219,253,253,253,253,195,
80,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,55,172,226,253,253,253,253,244,133,11,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,136,253,253,253,212,135,132,16,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

Figure 10.8: The digitized number 5.

And this is what that 5 looks like, Figure 10.9.

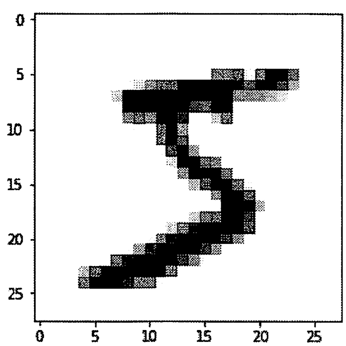


Figure 10.9: The sample 5.

The network that I use has 784 inputs, just one hidden layer with a sigmoidal activation function and 50 nodes, and 10 outputs. Why 10 outputs

and not just the one? After all we are trying to predict a number. The reason is obvious, as a drawing it is not possible to say that, say, a 7 is mid way between a 6 and an 8. So this is a classification problem rather than a regression.

10.13 Training And Testing

For the first time in this book I'm actually going to do both the training and the testing.

The MNIST training set consists of 60,000 handwritten digits. The network is trained on this data by running each data point through the network and updating the weights and biases using the backpropagation algorithm and stochastic gradient descent. If we run all the data through once that is called an epoch. It will give us values for the weights and biases. Although the network has been trained on all of the data the stochastic gradient descent method will have only seen each data point once. And because of the usually small learning rate the network will not have had time to converge. So what we do is give the network another look at the data. That would then be two epochs. Gradually the weights and biases move in the right direction, the network is learning. And so to three, four, and more epochs.

We find that the error decreases as the number of epochs increases. It will typically reach some limit beyond which there is no more improvement. This convergence won't be monotonic because there will be randomness in the ordering of the samples in the stochastic gradient descent.

In Figure 10.10 is a plot of the error, measured by the fraction of digits that are misclassified, against the number of epochs.

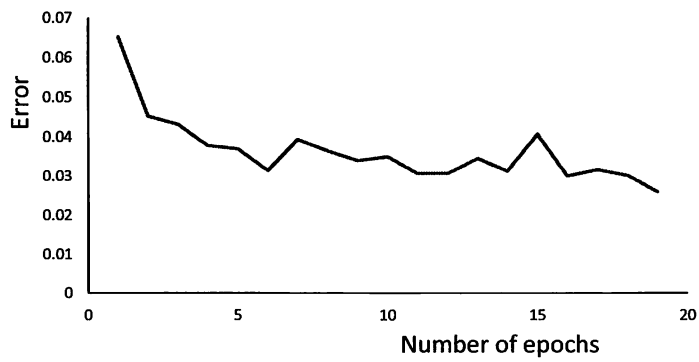


Figure 10.10: Error versus number of epochs, training set.

After 19 epochs we are getting a 97.4% accuracy.

So the network has learned, the error is decreasing. But what has it learned? Has it learned the right thing?

We want the network to learn to recognise handwritten digits, but we don't want it to memorize the data we have given it. We don't want to overfit. That's why we hold back some data for testing purposes. And obviously the data we hold back will be a random subset of the entire data set.

Let's look at how well the trained network copes with the test data. In Figure 10.11 we see the error versus epoch for the test data as well as for the training data.

Clearly the network is not doing so well on the test data, with only a 95% accuracy. That's almost double the error on the training data. But it's still pretty good for such a simple network.

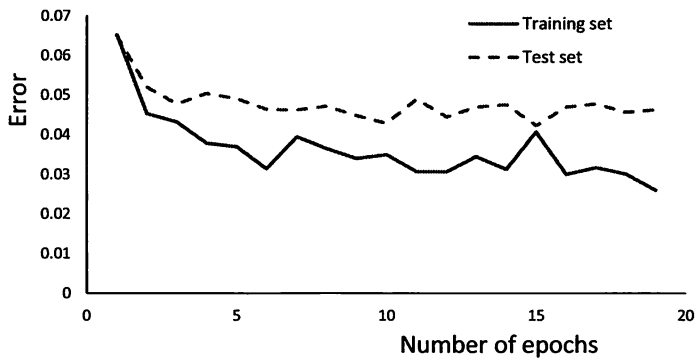


Figure 10.11: Error versus number of epochs, training and testing sets.

In practice one would conduct further experiments, varying the number of nodes in the hidden layer and also varying the number of hidden layers. As a rule the more neurons you have then the more epochs you'll need for training.

Also with more neurons you often find that for the test data the error gets worse with increasing number of neurons. This is a sign that you have definitely overfitted the data. And it's really easy to overfit if you have a lot of neurons. Take a look in Chapter 2 to see the discussion of training, test and validation datasets.

Once one has an idea of accuracy versus architecture one can then look at speed of prediction for new samples. The time of training is not necessarily important but if speed of prediction is important then one will need an architecture that is fast. It's very easy to estimate the time taken for an

architecture. For example suppose one has 784 inputs, 50 nodes in one hidden layer and ten outputs then the time taken will be proportional to

$$784 \times 50 + 50 \times 10 = 39,700.$$

But if we had, say, 784 inputs, one hidden layer of 30, another of 20, and ten outputs then the time would be proportional to

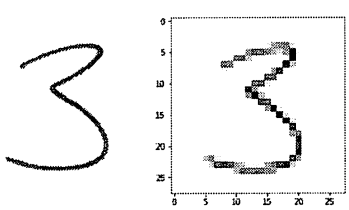
$$784 \times 30 + 30 \times 20 + 20 \times 10 = 24,320.$$

The same number of nodes, but only two thirds of the time.

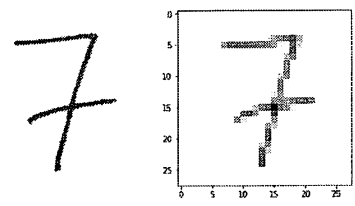
(This is assuming that different activation functions all take roughly the same time to compute.)

My digits

I showed a few of my handwritten digits to the trained network. In Figure 10.12(a) is my number 3 together with the digitised version. It got this one right. Figure 10.12(b) is my seven, it got this one wrong. Perhaps because it's European style, with the bar.



(a) My number 3.



(b) My number 7.

Figure 10.12: My numbers.

The output from the network is a ten-dimensional vector, and by looking at the entries we can see what the network thinks of my handwriting. This output suggests that the network sees my number 7 as 84.2% number 2, 6.8% number 7 and 5.7% number 4. You can see where it's coming from.

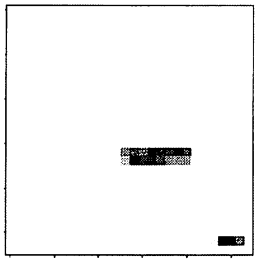
Nonsense

Often one sees tests of character-recognition networks using input noise to see which number is forecast. I thought I'd try something similar but different.

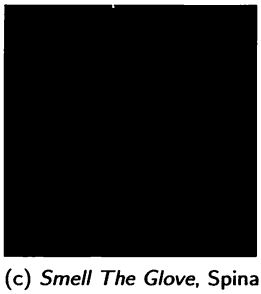
So as a final experiment I gave the network a couple of famous album covers and a *Wilmott* magazine cover, all converted to 28 × 28, just to see how these are interpreted. These are shown in Figure 10.13.

The White Album was interpreted by the network as a five, *Smell The Glove* was a zero, and the *Wilmott* magazine (featuring Ed Thorp, see the next chapter) was seen as a three.

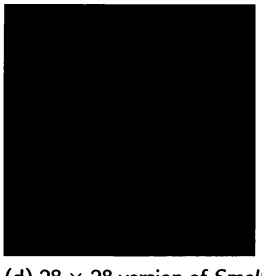
(a) *The White Album*, The Beatles.



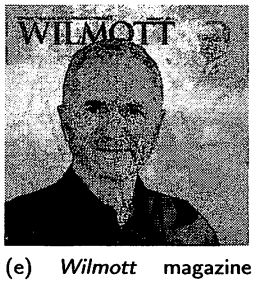
(b) 28 × 28 version of *The White Album*.



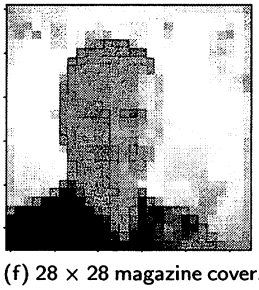
(c) *Smell The Glove*, Spinal Tap.



(d) 28 × 28 version of *Smell The Glove*.



(e) *Wilmott* magazine, January 2017.



(f) 28 × 28 magazine cover.

Figure 10.13: Covers

10.14 More Architectures

Autoencoder

The autoencoder is a very clever idea that has outputs the same as the inputs. Whaaat? The idea is to pass the inputs through a hidden layer (or more than one) having significantly lower dimension than the input layer and then output a good approximation of the inputs. It's like a neural network version of Principal Components Analysis in which the original dimension of the data is reduced (to the number of nodes in the hidden layer).

Training is done exactly as described above, just that now the ys are the same as the xs.

You can see the idea illustrated in Figure 10.14.

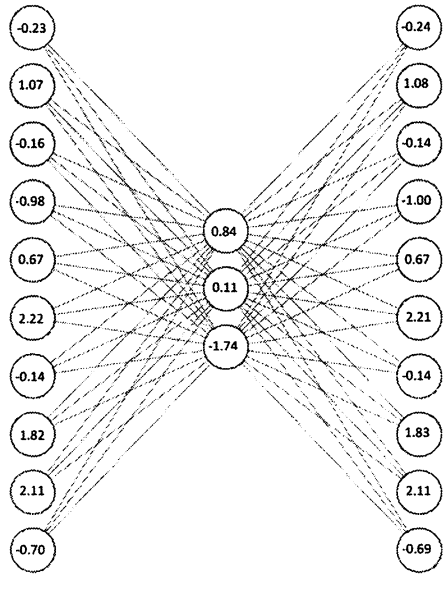


Figure 10.14: Illustrating the autoencoder.

The data goes in the left-hand side (the figure just shows one of the samples). It goes through the network, passing through a bottleneck, and then comes out the other side virtually unchanged. You'll see in the figure that the output data is slightly different from the input since some information has been lost. But we have reduced the dimension from ten to three in this example.

Now if you want to compactly represent a sample you just run it through

the autoencoder to the bottleneck and keep the numbers in the bottleneck layer (here the 0.84, 0.11, -1.74). Or if you want to generate samples then just put some numbers into the network at the bottleneck layer and run to the output layer.

There doesn't have to be just the one hidden layer. As always there can be any number. But the lowest dimension is the dimension of the smallest layer.

Radial basis function network

The radial-basis-function network uses typically the Gaussian function instead of a linear transformation. So in a node in a hidden layer you would see

$$w_i \, e^{-b|\mathbf{x}-\mathbf{c}_j|^2}$$

instead of, say, a sigmoidal function. And then the output layer would add up all of these, perhaps with another activation function transformation as well.

You can probably see overlaps with self-organizing maps, support vector machines and K nearest neighbours.

Recurrent neural network and long short-term memory

More complicated network architectures have feedback loops. These recurrent networks (RNN) have connections such that the output from a neuron feeds back into this neuron as well as being passed on. Suitably constructed such networks have a memory that means they can cope well with data that is sequential in nature, where the order matters, such as speech.

Such networks are often represented by something like Figure 10.15.

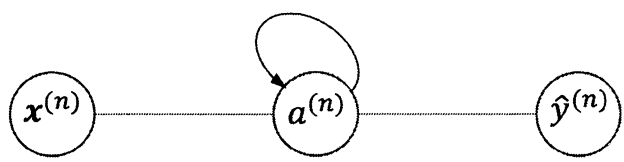


Figure 10.15: A schematic of a recurrent neural network.

This can be unpacked as shown in Figure 10.16.

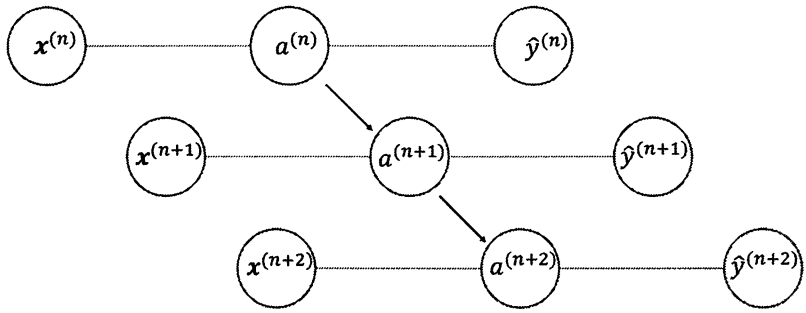


Figure 10.16: The RNN unpacked.

These networks can be complemented with long short-term memory (LSTM) building blocks. These units of a network have gates that can pass information on, reject it, or decide how much to remember.

10.15 Deep Learning

You will often hear the phrase Deep Learning. Different people mean different things when they use the phrase, there doesn't seem to be a generally recognised definition. But what all the different interpretations have in common is that neural networks are involved and that they have a large number of neurons and hidden layers.

As the amount of data that we have access to explodes and the speed of computation increases so it makes sense to explore the possibilities of networks that are deeper and deeper. Why stop at L layers? When you get more data you might as well look at the performance of $L + 1$ layers and more sophisticated architecture.

Further Reading

The first version of the Universal Approximation Theorem was by George Cybenko in 1989, but only for activation functions with sigmoidal-type asymptotic behaviour. The theorem applies even if the activation function is not monotonic. See "Approximations by superpositions of sigmoidal functions," *Mathematics of Control, Signals, and Systems* 2(4), 303—314.

A simple, straightforward book that explains what is going on and has Python code for character recognition is *Make Your Own Neural Network* by Tariq Rashid. The code actually works, helped me produce some of the results above, and almost made we want to learn Python. Almost. Thank you, Tariq.

Quite a detailed book describing many architectures is *Neural Network Design* by Martin Hagan, Howard Demuth, Mark Beale and Orlando De Jesus.

Chapter 11

Reinforcement Learning

11.1 Executive Summary

Reinforcement learning is one of the main types of machine learning. Using a system of rewards and punishments an algorithm learns how to act or behave. It might learn to play a game or move around an environment. The key to reinforcement learning is that the game or environment is not explicitly programmed. There is no rule book to which we can refer. This means that the algorithm has to learn the consequences of taking actions from taking those actions. The algorithm learns by trial and error.

Unlike the topics in the previous couple of chapters you can do plenty of reinforcement learning in a spreadsheet, at least to get you started.

11.2 What Is It Used For?

Reinforcement learning is used for

- Learning how to interact with an environment especially when the environment or the result of interactions aren't known in advance
- Example: Learn how to optimally bid at an auction
- Example: Learn which adverts to present to individual consumers
- Example: Learn how to play video games

I have saved what might be the most interesting methodology to last. It's interesting if you have children or dogs. While unsupervised learning is about finding relationships in data, and supervised learning is about deciding what something is, reinforcement learning is about teaching the machine to

do something. And it really is inspired by behavioural psychology. Sit, down, fetch, roll over, are all commands that with the right sort of reinforcement any child will eventually understand. We want an algorithm to learn what actions to take so as to maximize some reward (and/or minimize punishment).

Because you want the machine to learn to attain some goal it is very common to see the method used for playing games. And our examples here will also be from, or related to, games. But the trick with reinforcement learning is that you don't necessarily tell the machine explicitly what the goal of the game is or even what the rules are. It will learn by trial and error as it interacts with its environment.

You can see this happening if you give a two-year old the TV remote control. They've seen you use the remote control to switch on the TV and change channels so they know of some link there. But when first given the remote they will press buttons at random until something happens. There will be a phase during which they home in on the most important buttons, and after a while they learn which is the button to switch the TV on, to change channels, to start a DVD. And thereafter they will use a subset of buttons reliably. (Mind you, which adult knows what more than a 20% subset of the buttons do as well?) Funnily though they never seem to learn how to switch the TV off.

Another way of looking at trial and error is in terms of exploiting and exploring. On one hand you want the machine to take advantage of/exploit everything it has learned in order to win the game, say, but on the other hand it won't learn anything unless it has done plenty of exploration beforehand. Getting a balance between exploiting and exploring will be important to our learning algorithms.

11.3 Going Offroad In Your Lamborghini 400 GT

The structure of this chapter is to alternate motivating examples with mathematics. The motivating examples are Noughts & Crosses, fruit machines, a simple maze and Blackjack.

You will be seeing some rigorous and fun mathematics based around these simple examples. However, and I want to emphasis this, when it comes to reinforcement learning proper as applied to real problems we won't necessarily have that much in the way of rigorous underpinnings. Our *examples* all have elegant formulations that inspire the mathematics. But *real-world problems* typically don't.

Anyway, elegance is not the point of reinforcement learning proper. Reinforcement learning comes into its own when we have a problem that doesn't have any such elegant formulations. The problem might be too messy, too complicated, too uncertain, etc.

To me this is a slightly strange topic. We do lots of mathematics but in practice that mathematics is going to ultimately be irrelevant. At least that's how I see it. An analogy comes from motoring.

You design, build and test a beautiful new car. It is designed to look good parked outside Harrods. It is built carefully by skilled craftsmen. And it is tested on very straight, very smooth motorways. It is a dream of a car. It goes into production. But the new owners decide that they want to use the car for offroading in Wales. What could possibly go wrong?

The designing, building and testing are what we will do on our simple, elegant problems. That's not exactly pushing reinforcement learning to its limits. And then we let the algorithm loose in a totally new and very complicated terrain. That *is* reinforcement learning. And quite frankly we don't know whether or not it's going to work. That is, either until it beats the professionals at Go or it gets stuck fording a stream in Wales.

And this is why I keep referring to reinforcement learning proper. The "proper" is to emphasise that we typically use reinforcement learning when we have little clue as to what the environment is, or it's just too complicated to represent compactly.

I end the chapter, and the main body of the book, with the example of Blackjack. It is perfect for trying out these techniques. And we are going to treat the game as if we have no idea as to what the game is all about. That means that we won't be relying on any of the aforementioned foundations. Of course, there's the usual disclaimer, do not take any of the results here into a casino and expect to win. You really mustn't rely on my programming.

Oh, and I've also saved this method until last because it's a bit heavier on the mathematics. And it's the longest chapter.

11.4 Jargon

I'm going to explain some preliminary, basic, jargon while referring to a few common games.

- **Action:** What are the decisions you can make? Which one-armed bandit do you choose in a casino? Where do you put your cross in the game of Noughts and Crosses? How many, and which, cards do you exchange in a game of draw poker? Those are all example of actions.

- **Reward/Punishment:** You take an action and you might get rewarded. You press a button on the wall in Doom and the BFG is revealed. You take your opponent's piece in checkers. It's white chocolate and you eat it. Those are examples of immediate rewards. But

there might not be any reward until the end of the game. At the end of the chess game the winner gets the $1,000 prize.

But there aren't just rewards. To win the prize you have to be the first to solve the jigsaw puzzle. Every second you take can be thought of as a punishment.

- **State:** The state is a representation of how the game is now. Think of it as a snapshot of the gameboard, for example. It might be the positions of the Os and Xs in a game of O&Xs. Or the positions of the stones in a game of kōnane. The state is an interesting concept. How much information is needed to represent a state? In O&Xs Donald Michie needed 304 matchboxes. In Blackjack, which we shall see in detail later, you need at a minimum to know the count of the cards you hold, and how many Aces, and what the dealer's upcard is. To stand a chance of winning in a casino you also need to know something about what cards have been dealt from the deck(s). But sometimes the amount of information you must store is prohibitively large. In Go there are typically 361 points, each of which could be empty, or be occupied by a white or black stone. Or the state might not even be represented by discrete quantities. At what angle should you kick the ball when taking a penalty? And for reinforcement learning proper we won't even know what information represents the state.

- **Markov Decision Process (MDP):** Markovian means that what happens going forward only depends on the current state. Chess is a Markov Decision Process, the state of the board tells you everything you need to know to make a decision about your next move, it doesn't matter how you got there. If we use s_t to denote the state at the t^{th} time step then for a Markov process we could write $\text{Prob}(s_{t+1} = s' \mid s_1, s_2, \ldots, s_t) = \text{Prob}(s_{t+1} = s' \mid s_t)$. I.e. the probability of being at some state s' at the next step only depends on the current state. Note that we can include time or time step as a component in the state.

11.5 A First Look At Blackjack

Blackjack, a game we shall look at in detail later in this chapter, is an MDP if you keep track of enough information to represent the state. That state is not represented simply by your cards. You need to keep track of a lot more information to capture the state. Part of that information is knowing the dealer's upcard. But even knowing the dealer's upcard as well as your cards is not going to totally specify the state. The reason is that what happens next, in the drawing of cards, depends on what cards are left

in the deck (or decks, plural, in casino Blackjack the dealer will start with five or more decks shuffled together). Or equivalently knowing what cards have been dealt out already. (Well, not their suits, and 10, Jack, Queen and King all count as 10s, but that's still a lot of into you need to know.) So Blackjack *is* an MDP *if* you are able to memorize all of this information. But that's unrealistic for a mere mortal... and you aren't allowed to use a computer in a casino. See *Rain Man*. At the end of this chapter I'll briefly show you how you can approximate the state sufficiently to win at Blackjack.

An exception to this is when there is an infinite number of decks being dealt from, in an online game. In that case Blackjack is an MDP if the state is represented by your cards and the dealer's upcard. That's because the probabilities for the next cards to be dealt never changes. But then you can't win if you play with an infinite deck. Confused? You will be!

Markov refers to there being no memory if your state includes enough information. So that is part of the trick of learning how to play any game. Keep track of as many variables as needed, but no more.

You can get a lot more sophisticated with different types of MDP. For example in a Partially Observable Markov Decision Process (POMDP) you only get to see some of the current state. And so you might have a probabilistic model for what you can't see.

11.6 The Classical MDP Approach In O&Xs

The state in O&Xs is represented by the positions of the Os and Xs. The order in which they were played, i.e. how you got to that state, is irrelevant. Starting with an empty grid and it is us, with Xs, to go first we could sketch something like Figure 11.1 showing the three possible moves. There are only three despite the nine cells because of symmetries.

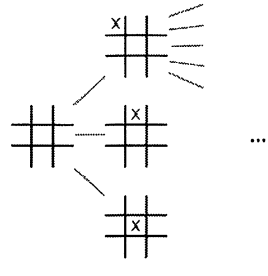


Figure 11.1: A tree structure shows what could happen next.

This structure continues for all possible plays in the game. From each

state branch out other states for the next play. Some finish when one player gets three in a row. And some end up with all cells filled and no winner. See Figure 11.2.

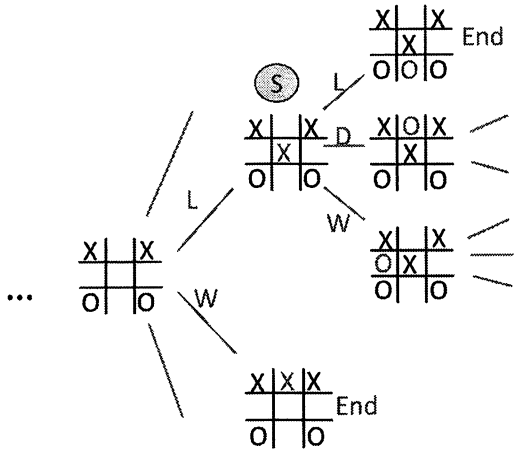


Figure 11.2: Part way through a game.

In Figure 11.2 I have labelled moves from state S as L for Lose, D for Draw and W for Win. For example at the very top right you will see that our opponent, the Os, have got three in a row. That's an L for us.

Now we ought to make some assumptions about how good our opponent is. That will help us make decisions about what actions to take.

A good opponent If we are playing against a good opponent then we must try not to get into any state from which an L follows. Referring to the figure that means that we must not put the X in the middle, that is we must try not to get to state S. That's why the line going into state S has been labelled L.

One would usually assign numerical values to winning, drawing and losing. Let's say win is $+1$, a draw is 0 and a loss -1. That means that if we are playing against a good player then State S in the figure has value -1.

Each branch coming out of a state will have a value. And if we are playing against a good player then we must assume that they will choose whatever gives us the lowest value.

Classically this is known as a minimax problem: You assume that on their move your opponent does the most to harm you, and then on your move you make the optimal move to minimize that harm.

The values will trickle back down the tree structure.

Opponent playing randomly However if our opponent is putting their O in vacant cells at random then State S is rather good. There is one losing position, -1, one draw, 0, and two (thanks to symmetry) wins, each $+1$, giving an average value for State S as 0.25.

And what if we don't know how good our opponent is? No problem. Because that's life! And that's what reinforcement learning is all about. Learning on the job, while experiencing the environment rather than having the environment programmed in.

All this is a gentle introduction to...

11.7 More Jargon

- **Value Function:** A value function measures how good or bad a state or an action is. If we are in some state now and all future states are good then the value at our present state will be high. But that value depends on what actions we take now and in the future. At each state we will have some choices to make. *Value* comes from accumulation of rewards and is how good our current position is given what we do now and in the future. Notice that I mention both state and action here. When we get to the mathematics you'll see how I distinguish between two types of value function, one specific to the state and one that is associated with both state and action. Given a state and how we decide what action to take defines our...

- **Policy:** A policy is a set of rules governing what action to take in each state. That policy might be deterministic. Or it might be random. Ultimately in reinforcement learning we want the policy to maximize value at each state. But of course *a priori* we don't know what the best policy is... that's what we are trying to get the machine to learn.

In O&Xs each state would have a value, a value that percolates down from later states in the game.

I'll be talking about the value function in relation to reinforcement learning in some detail and give some ideas about how to set it up.

So that's how you might address the game of O&Xs classically. Other games and problems can be set up in the expanding tree structure as well.

Let's start to move out of the waffly, descriptive mode, and start at least hinting at some mathematics. We shall do this by looking at an extremely popular illustrative example.

11.8 Example: The multi-armed bandit

You'll know of the one-armed bandit. It is a gambling machine found in casinos and bars. Originally these machines had a lever, the arm, that you pull, and this causes cylinders, on which there are pictures of various fruit, to spin. (Hence the name fruit machine.) If they stop on the right combination of lemons, cherries, etc. then you win a prize.

In a casino you will have many of these in a row. (So, I think multiple bandits is a much better, and less clunky, name than multi-armed bandit.) In the multi-armed bandit problem we have several such bandits each with a different probability of winning a fixed amount, the same amount for each bandit. The goal of the multi-armed bandit as a problem in reinforcement learning is to choose among the bandits, pull the lever, see if you win, try a different bandit, and by looking at your rewards learn which is the bandit with the best odds.

This problem is quite straightforward. There is first of all no state as such. But there are actions, which bandit to choose. We want to assign a value to each action. And then based on these values decide which action to take.

This is how it goes...

- There will be ten bandits. The probabilities of winning for each bandit are

Bandit 1	10%
Bandit 2	50%
Bandit 3	60%
Bandit 4	80%
Bandit 5	10%
Bandit 6	25%
Bandit 7	60%
Bandit 8	45%
Bandit 9	75%
Bandit 10	65%

Table 11.1: Table of probabilities of winning in the multi-armed bandit example.

- The value of each action, each bandit, will simply be the average reward for that bandit. This average is only updated after each pull of a lever. To be precise, it is only the chosen bandit that has its average

updated. And the average is calculated as the total actual reward so far, using randomly generated success/fail at each pull, divided by the total number of times that bandit has been chosen. To all intents and purposes the reinforcement algorithm does not *know* about the odds, only *experiences* them.

- After each pull we have to choose the next bandit to try. This is done by most of the time choosing the action (the bandit) that has the highest value at that point in time. But every now and then, let's say at a random 10% of the time, we simply choose an action (bandit) at random with all bandits equally likely.

That last bullet point describes what is known as an ϵ-greedy policy. You choose the best action so far, but you also do occasional exploration in case you haven't yet found the best policy. The random policy happens a fraction, ϵ, of the time.

I'd also like to emphasise that only you, the reader, and I know the probabilities of winning for each bandit. We will not be telling the reinforcement-learning algorithm explicitly what these probabilities are. As I said above, the algorithm only indirectly experiences them.

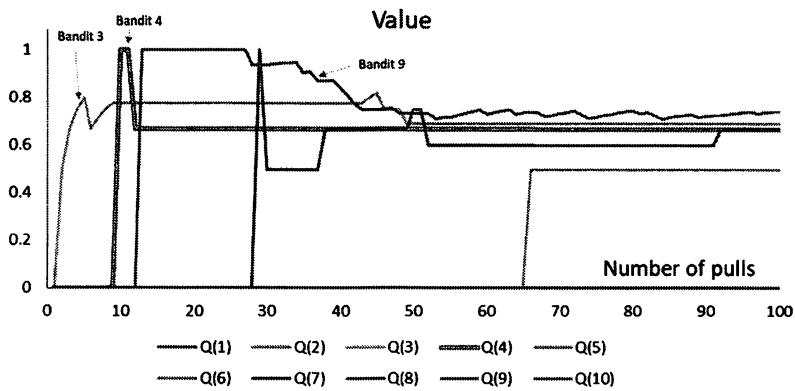


Figure 11.3: Value for each bandit versus number of pulls.

In Figure 11.3 is shown the value function for each of the ten bandits as a function of the total number of pulls so far. I have used Q to denote this value, we'll see more of Q later. And in Figure 11.4 we see the fraction of time each bandit has been chosen.

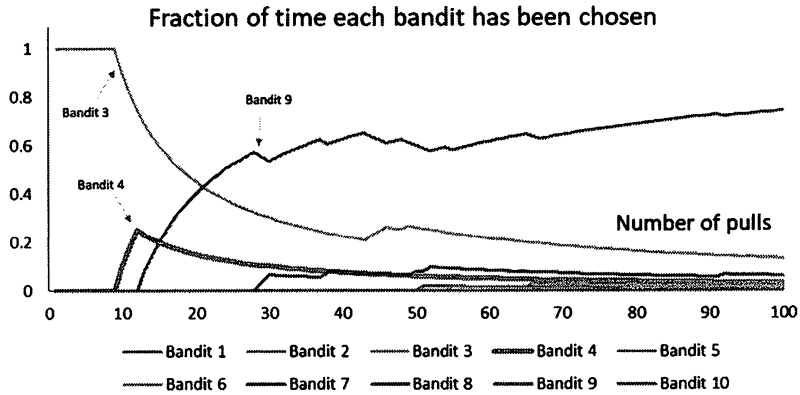


Figure 11.4: Fraction of time each bandit has been chosen.

It's hard to see what's going on in black and white with so many curves, but let me walk you through it. Bandit 3 was chosen at random for the first pull. The first pull is always random, we have no data for the value yet, and all Qs are set to zero. Bandit 3 loses. For the next pull the Qs are still all zero and so each bandit is equally likely. Coincidentally Bandit 3 is chosen again. This time Bandit 3 wins. Its Q value is now 0.5 and so is currently the best bandit. This means that it will be chosen preferentially except when the 10% random choice kicks in. Bandit 4 eventually gets picked thanks to this random choice. It wins and thus now becomes the preferred choice. It wins twice, then loses. And so it goes on. Up to 100 pulls it is looking like Bandit 9 is the best. But then...

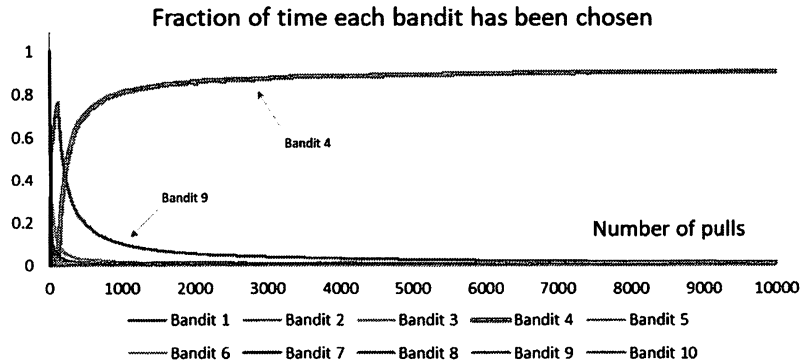


Figure 11.5: Fraction of time each bandit has been chosen, longer run.

In Figure 11.5 is shown the fraction of pulls for each bandit up to 10,000 pulls. Clearly Bandit 4 has become the best choice. And if you look at Table 11.1 you'll see that it does indeed have the highest probability of success.

The correct bandit will eventually be chosen however the evolution of the Qs and the fraction of time each bandit is chosen will depend for a while on which bandits are chosen at random. So in Figure 11.6 I show the results of doing 100 runs of 10,000 pulls each. Having 100 runs has the effect of averaging out the randomness in the experiments. Note the logarithmic scale for the horizontal axis.

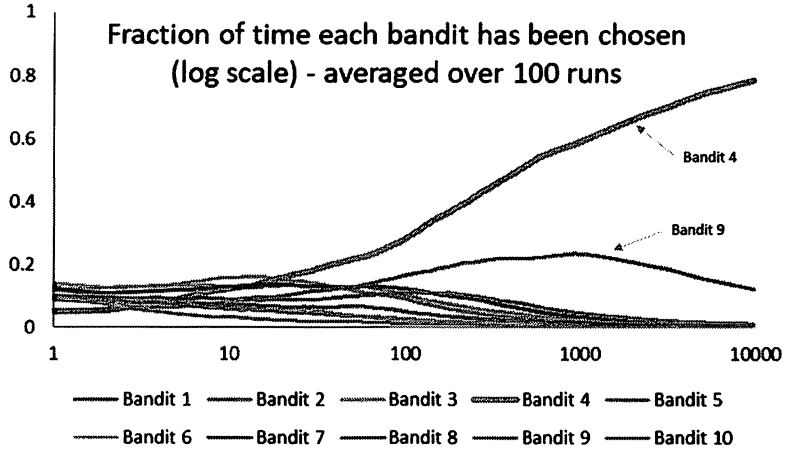


Figure 11.6: Fraction of time each bandit has been chosen (log scale) — averaged over 100 runs.

11.9 Getting More Sophisticated 1: Known environment

We've done the preparatory work, seen basic ideas of how and where reinforcement learning can be applied, now let's start putting some proper mathematical flesh on these bones.

In the next few sections we are going to do some of the mathematics behind valuation for a given policy, policy evaluation, and then find the optimal policy, the control problem. But initially we are going to restrict our attention to known environments. Known environment in an MDP means that for any state and action we have a known probabilistic description of our next state and any reward.

In a sense Blackjack is a known environment because in principle we could write down the probabilities of going from one state to another after taking a certain action. So maybe have that idea in mind while we next learn about the subject of Dynamic Programming.

This isn't really reinforcement learning yet. That comes when we move to an unknown environment.

We are going to be justifying reinforcement learning by looking at related topics. Along the way we'll be introducing things like transition probabilities. But when we finally get to reinforcement learning proper we won't be seeing them anymore. And that's simply because reinforcement learning is designed to work even if you don't know these probabilities.

Basic notation

We shall need notation to represent states, actions, rewards, and transition probabilities between states. And some other things. Everything that follows uses the standard notation seen in most of the literature. Understanding notation is half the battle of learning any new subject.

States Let's use S to denote the general state and s to mean a specific state. I'll use s' to mean the next state (after we've taken some action). And sometimes I'll give s a subscript when I want to emphasise the time or step, s_t.

Actions I'll use A to mean the general action and a to denote a specific action. Again the following action will be a' and sometimes the action will have a subscript when I want to emphasise the time or step at which the action is taken, a_t. Note that taking a certain action does not guarantee what state we'll end up in. For example, in Blackjack taking another card does not tell us what our next card count is. But we could have a simple case, like in the maze that we'll see later, where the action of moving right does tell us our next state.

Rewards Generally the reward can depend on any or all of action a, current state s and next state s'. I'll use r to denote the actual reward. Sometimes the r will have a subscript if I need to make it clear which reward I am talking about. For example r_{t+1} is the reward that comes between state s_t and state s_{t+1}.

I will also write

$$R^a_{ss'} = \mathbb{E}\left[r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\right].$$

In this expected reward we've got the before and after states, as well as the action. Reward includes punishment, that would just be a negative reward. For example if we are trying to find the shortest path from start to finish in a maze then we would have a reward of -1 for every action (step) regardless which way it is. Then maximising the sum of rewards would be the same as finding that required path.

Transition probabilities In general we will need to specify the probability of going from state s to state s' after taking action a. We denote these probabilities by

$$P^a_{ss'} = \text{Prob}\left(s_{t+1} = s' \mid s_t = s, a_t = a\right).$$

For our maze below these probabilities would be either 1 or 0, because a given action tells us exactly where we go next. But for Blackjack the action of taking a card puts us in a random next state.

Policies Given a state $S = s$ then there will generally be several possible actions. And generally speaking those actions can have probabilistic elements to them. We write such a stochastic policy as

$$\pi(a|s) = \text{Prob}(A = a \mid S = s).$$

In words, this is the probability of taking action a when in state s. I have chosen to write this using the | symbol to separate the variables, but you will often see a simple comma in the literature. I have made this choice to emphasise two things: That you first get to a state then the action comes next; In each state the choice of actions could be completely different (buy an apple or orange in the shop, or walk left or right out of the house).

I shall also use π as a subscript shortly. Then it just means "for the given policy." You'll see.

Goal or return The total reward, called the return or goal, that lies ahead of us is

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{j=0}^{\infty} \gamma^j r_{t+j+1}.$$

The parameter $\gamma \in [0, 1]$ is a discount factor that says that the sooner the reward the more we value it.

The discount factor If you come from a finance background then you will know that use of discount factors, related to interest rates, is extremely common. We talk of the time value of money and opportunity cost meaning all things being equal it's better to get money upfront. A bird in the hand is worth two in the bush. However the discount factor in reinforcement learning can seem a bit arbitrary. Non-financial reasons for introducing one are:

- It can avoid infinite sums, when adding up an infinite number of rewards. (There might be loops in a game giving rewards and rather than seeking a better solution we get caught up in this loop)
- If we aren't totally sure about the environment then there is model uncertainty in the future so best take those rewards now
- In a real-life scenario the rewards might be perishable!

And the geometric time decay (the increasing powers of γ) is the most common discounting mechanism since it has nice mathematical properties, as we shall see.

11.10 Example: A maze

I've described the concept of Markovian in reference to states and memory. A Markov chain is a sequence of events in which we move from state to state according to given probabilities.

Our example now is a maze, see Figure 11.7. This is about the smallest maze that I can use to illustrate the idea of Markov chains.

It's a simple example because first of all the reward will be -1 for every action, i.e. every step. And initially the policy will also be simple, we will move to any adjacent cell with equal probability.

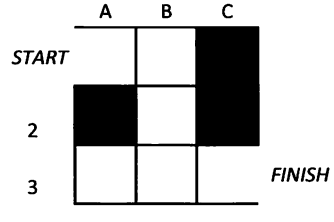


Figure 11.7: A very, very simple maze.

We start in Cell A1, can only go right from there, and when we get to Cell C3 we stop. Our state is represented by which cell we are in, B3 for example.

In a Markov chain the probability of moving from one state to another is given. But recall that in our notation above we had a potentially stochastic action (the policy, π) and a potentially stochastic transition (the transition probabilities, $P^a_{ss'}$). In the simple Markov chain here these two can be combined into one transition probability matrix, and that's because given an action (albeit random) the following state is deterministic: Move down from B1 and you always end up in B2. (Another way of thinking of this is that $P^a_{ss'}$ only contains ones and zeroes.)

To represent the (combined) transition probabilities we would need a 6 × 6 matrix (there are six cells). See Figure 11.8. In this example I am going to make a move from one cell to a neighbouring one equally likely: If there is only one neighbour then the probability of moving to it is 100%, if two neighbours then 50% each, etc. At this point there is nothing special about Cells A1 and C3, other than we can only go in one direction from A1 and we stop when we get to C3, there is no attempt to go from Start to Finish. The transition probabilities tell us where to go with no goal in mind.

		To					
		A1	**A3**	**B1**	**B2**	**B3**	**C3**
From	**A1**	0	0	1	0	0	0
	A3	0	0	0	0	1	0
	B1	0.5	0	0	0.5	0	0
	B2	0	0	0.5	0	0.5	0
	B3	0	0.333	0	0.333	0	0.333
	C3	0	0	0	0	0	1

Figure 11.8: Transition probabilities.

Solving the Markov-chain problem

If the next move we make is given by the transition matrix in Figure 11.8 then we can write down a recursive relationship for the expected number of steps it will take to get from any cell to the finish. Denote the expected

number of steps as a function of the state, i.e. the cell we are in, $v(s_t)$.

We have

$$v(\text{A1}) = 1 + v(\text{B1})$$

because whatever the number of expected steps from B1, the expected number from A1 is one more. Similarly

$$v(\text{A3}) = 1 + v(\text{B3}).$$

From Cell B1 we can go in two directions, both equally likely, and so

$$v(\text{B1}) = 1 + \frac{1}{2}\left(v(\text{A1}) + v(\text{B2})\right).$$

And so on...

$$v(\text{B2}) = 1 + \frac{1}{2}\left(v(\text{B1}) + v(\text{B3})\right),$$

$$v(\text{B3}) = 1 + \frac{1}{3}\left(v(\text{A3}) + v(\text{B2}) + v(\text{C3})\right)$$

(from B3 there are three actions you can take) and finally

$$v(\text{C3}) = 0 + v(\text{C3}).$$

You never leave Cell C3.

These can be written compactly using vector notation. Let's write v as a vector $\mathbf{v}$ with each entry representing a state, A1, A3, ..., C3. Similarly we'll write the reward as a vector $\mathbf{r}$.

In the above we have added 1 for every step in order to calculate the expected number of steps in each state. But to make this more like a reinforcement learning problem I am going to change the sign of the reward, so that we are penalized for every step we take. This ties in with the later idea of optimizing return. Maximizing the total reward when each reward is negative amounts to minimizing the number of steps. The final answer for $\mathbf{v}$ will be the negative of the expected number of steps for each state.

So the first five entries in this reward vector, $\mathbf{r}$, will be minus one, and the final entry will be zero. This just means that in going from state to state we get a reward of -1, but since we can't leave cell C3 there is a zero for that entry. And we shall write the transition matrix in Figure 11.8 as $\mathbf{P}$.

Writing the above in vector form to find $\mathbf{v}$ all we have to do is solve

$$\mathbf{v} = \mathbf{r} + \mathbf{P}\mathbf{v}. \tag{11.1}$$

This tells us the relationship between all the expected values, one for each state. It is a version of the Bellman equation. We shall be seeing different forms of the Bellman equation later.

This can be solved by putting the two terms in $\mathbf{v}$ onto one side and inverting a matrix. However since inverting matrices can be numerically

time consuming it is often easier, and certainly will be in high dimensions, to iterate according to

$$\mathbf{v}_{k+1} = \mathbf{r} + \mathbf{P}\mathbf{v}_k.$$

We've seen iterative methods in gradient descent and we are going to be seeing more iterative methods when we finally get to reinforcement learning.

We can see the iteration process in action in Figure 11.9. Here we started with the $\mathbf{v_0}$ having all entries zero.

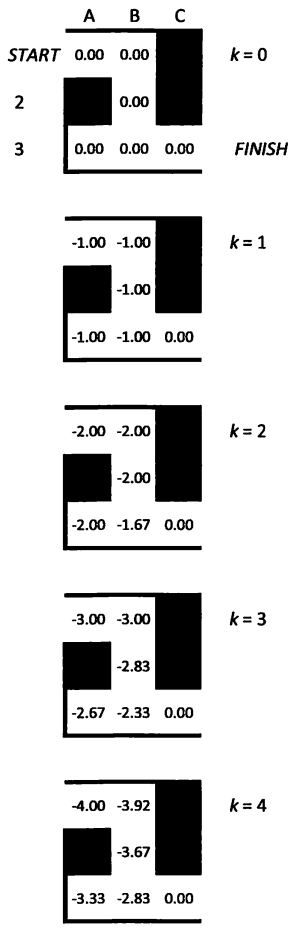


Figure 11.9: The iterative procedure for finding $\mathbf{v}$.

The final result, after many iterations, is that shown in Figure 11.10. The numbers are easy to check.

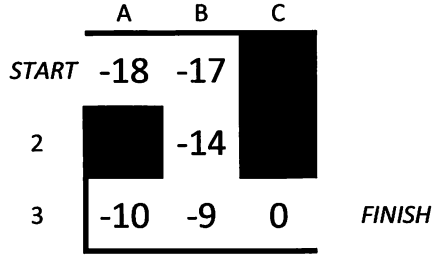


Figure 11.10: Solution of the Markov chain.

That looks to me like it's taking an awfully long time to get from A1 to C3 if you move randomly, 18 steps on average.

Of course, the big difference between Markov chains and MDPs is in the potential for us to make (optimal) decisions about which actions to take. Ultimately our goal in the maze is trying to get from start to finish asap. We want to deduce, rather trivially in this maze, that, for example, the optimal policy when in cell B3 is to move right. That will come later, after I've introduced some more notation.

11.11 Value Notation

State-value function

Let's forget the maze example for a while and set out a fairly general problem in which we have a given policy, given rewards and given probabilities for changing from state to state. Everything can be stochastic and the rewards and transition probabilities can be functions of all of s, a and s'.

In the above I used v (or $\mathbf{v}$) to represent the expected (negative of the) number of steps. More generally in an MDP we can use v to represent a value for each state. Such a state-value function is defined as

$$v_\pi(s_t) = \mathbb{E}_\pi \left[G_t \mid S = s_t \right]. \tag{11.2}$$

This function $v_\pi(s_t)$ adds up all the expected rewards, possibly discounted, and this tells us how good a particular state is. The subscript on the v is just to acknowledge that the value depends on the policy. In our maze that policy was stochastic. But this will change soon.

Expression (11.2) simply says that we take the expected value of the

sum of all *future* rewards. It is crucially important that we are adding up (expected) rewards that are yet to come. Recall the discussion of the O&Xs MDP where I mentioned valuing back down the branching tree. This seems a bit counterintuitive, surely we should add up the rewards we've collected? Well, no, because the rewards that we've collected already aren't going to, or rather shouldn't, influence our action now and in the future. It's what we might collect in the future starting from each state that will determine our behaviour now. It's no good crying over spilt milk.

Note also that if the rs are positive then our value function will decrease as we go forward in time.

We shall find it useful to introduce another value function the...

Action-value function

...defined as

$$Q_\pi(s_t, a_t) = \mathbb{E}_\pi \left[G_t \mid S = s_t, A = a_t \right].$$

This means the value when in state s_t of taking action $A = a_t$ and *afterwards* following the policy π. So that's the immediate reward plus the value that comes after.

(And that's the Q we saw in the section on the multi-armed bandit.)

This is the quantity that will later tell us what is the best action to take next, the one that maximizes $Q_\pi(s_t, a_t)$.

The relationship between the two value functions is

$$v_\pi(s_t) = \sum_a \pi(a|s_t) Q_\pi(s_t, a).$$

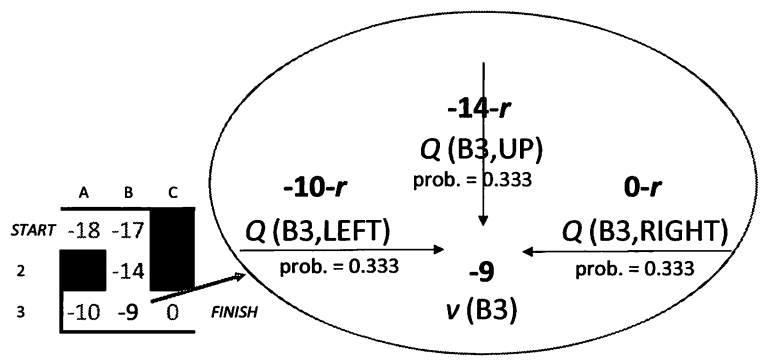


Figure 11.11: The relationship between the value functions for Cell B3 in our maze.

This relationship between the two types of value function is shown in Figure 11.11 for Cell B3 of our maze. I've written the -1 reward as $-r$ just to help emphasise what the numbers mean, that the reward is added between states, after an action.

Since our policy is stochastic with Left, Up and Right actions all having probability one third we have

$$v(\text{B3}) = \frac{1}{3}\left(Q(\text{B3}, \text{LEFT}) + Q(\text{B3}), \text{UP}) + Q(\text{B3}), \text{Right})\right)$$

$$= \frac{1}{3}\left(-10 - r - 14 - r + 0 - r\right) = -9.$$

You can already start to see the benefits of the action-value function Q from the figure. Out of the three possible actions in state B3 it's moving to the right that has the highest Q value. Of course, that's assuming that the state values in the adjacent cells are correct.

You should anticipate an iterative solution for the Q function, and thus the fastest route through the maze. And that's sort of where we are heading. Except that we will get there by going through the maze many times, using trial and error. Enough of the maze.

11.12 The Bellman Equations

We can write these two value functions recursively: The value function *now* is just the expected value of the immediate reward plus the expectation of the value function *where you go to*. Let me show you how this works.

$$v_\pi(s) = \mathbb{E}_\pi\left[G_t \mid S_t = s\right] = \mathbb{E}_\pi\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \mid S_t = s\right]$$

$$= \mathbb{E}_\pi\left[r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \cdots) \mid S_t = s\right]$$

$$= \mathbb{E}_\pi\left[r_{t+1} + \gamma G_{t+1} \mid S_t = s\right].$$

We can write this as the recursive relationship

$$v_\pi(s_t) = \mathbb{E}_\pi\left[r_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s_t\right]. \tag{11.3}$$

But since there are potentially two probabilistic elements (the policy and the transitions) we can make this more explicit using the notation from Section 11.9:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} P^a_{ss'}\left(R^a_{ss'} + \gamma v_\pi(s')\right). \tag{11.4}$$

These final results are the Bellman equation, relating the current state-value function to the state-value function at the next step.

Note that this recursive relationship between the value function now and the value function at the next step is only possible because of the simple geometric discounting of future rewards. That's what I meant when I said earlier that geometrical discounting has nice mathematical properties.

The same process also gives us the recursive equation for the action-value function

$$Q_\pi(s_t, a_t) = \mathbb{E}_\pi \left[r_{t+1} + \gamma Q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s_t, A_t = a_t \right].$$

Or more explicitly

$$Q_\pi(s, a) = \sum_{s'} P^a_{ss'} \left(R^a_{ss'} + \gamma \sum_{a'} \pi(a'|s') Q_\pi(s', a') \right).$$

11.13 Optimal Policy

The whole point of setting up this problem is to find the optimal policy at each state. So we are seeking to maximize $v_\pi(s)$ over all policies π:

$$v_*(s) = \max_\pi v_\pi(s).$$

The $*$ is used to denote optimal.

And similarly

$$Q_*(s, a) = \max_\pi Q_\pi(s, a).$$

For clarity, this expression is the expected return in state s when taking action a and from then on following the optimal policy.

There are two more Bellman equations now, one for each of v_* and Q_*, called the Bellman optimality equations:

$$v_*(s) = \max_a \sum_{s'} P^a_{ss'} \left(R^a_{ss'} + \gamma v_*(s') \right)$$

and

$$Q_*(s, a) = \sum_{s'} P^a_{ss'} \left(R^a_{ss'} + \gamma \max_{a'} Q_*(s', a') \right).$$

It's the second of these that we focus on because it explicitly distinguishes the next action:

$$v_*(s) = \max_a Q_*(s, a).$$

And if we can find the next action then in principle the job is done because of the recursive nature of the Bellman equation. This is the benefit of working with Q, isolating the immediate next action.

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \operatorname{argmax}_a Q_*(s, a) \\ 0 & \text{otherwise} \end{cases}.$$

Once we have found the optimal policy then typically the $\pi(a|s)$ becomes deterministic, with probability of one for the optimal action and zero for other actions.

For completeness, here is the action-value function Q_* for our earlier maze. That is, the action-value function for the optimal policy. Meaning the value for taking an action and *thereafter* finding the optimal policy.

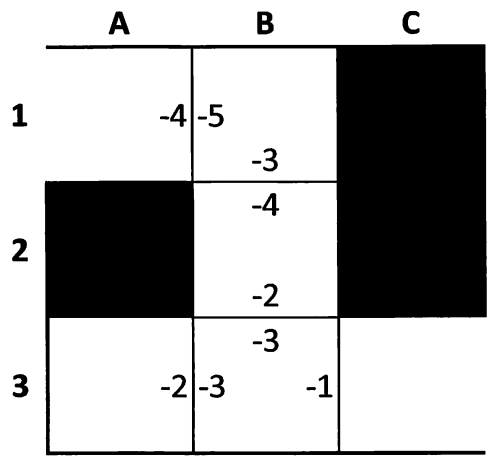


Figure 11.12: The Q_* function for our maze.

The position of the numbers in the cells just refers to the move. So the -4 at the top of Cell B2 means the value if you go up and thereafter do what is optimal.

11.14 The Role Of Probability

It's worth taking a moment out to look at the role of probability here. We started out by talking about a policy being probabilistic, hence all those earlier expectations. We've now moved on to talking about a deterministic policy. That doesn't mean all probabilistic elements have necessarily disappeared.

- There could well be elements of the environment that are always random. This is the case with the bandits and Blackjack. In both cases there will be deterministic optimal policies (based on the state in Blackjack) but the consequences still have random components. You may be on a losing streak, but that shouldn't change your policy. This is where we are in our discussion of the theory so far.

- Even if ultimately we are seeking a deterministic policy, such as in the bandit example, we might get there via a stochastic policy: We were trying to determine which bandit to bet on but we did that via a policy with occasional random choices. Think of the probabilistic element as just being a numerical method for getting to the right answer. This is something coming up later. (Keep an eye out for ϵ greedy.)

- There might genuinely be random elements that are part of the optimal policy. Rock, Paper, Scissors is the obvious example. However we aren't going to look at such cases here.

11.15 Getting More Sophisticated 2: Model free

In the first Getting More Sophisticated section I showed the mathematics for MDPs with a known environment, how to find value functions for a known policy and then how to optimize the policy. In my main example I kept things simple by focusing on a maze with a deterministic environment. However I did set up the mathematics for a known stochastic environment as well.

In the following sections we are going to move in the direction of unknown environments. We won't know *a priori* the transition probabilities or rewards. This is where reinforcement learning really starts.

A good example to look at is, of course, Blackjack. Technically the Blackjack environment is known because we could write down the transition probabilities: You hold a 12 count with the dealer's upcard being an eight, if the action is to take a card then what are the probabilities of going to 13, 14,..., bust? We could write down a largish $P^a_{ss'}$ matrix and use the above techniques but I'd rather use this as an example of an unknown environment, because then we really are in reinforcement-learning territory. Instead of going through the process of figuring out the transition probabilities, we simply play many games.

11.16 Monte Carlo Policy Evaluation

The method we'll look at first is Monte Carlo policy evaluation.

This method just plays the game, or whatever, many times and calculates expected, empirical, returns for each state over many games. You don't need complete *knowledge* of the environment, like you do for dynamic programming. All you need is to have *experience* of the environment. It's a simple method and very common.

It does have a drawback and that is that it only really works when you

have complete returns, which means that the MDP terminates: An episodic MDP has a well-defined start and end, it doesn't go on forever. You start the game, play, and then finish. That's one episode, and then you start all over again.

You couldn't easily calculate an expected return if the game never terminated.

We will use T to denote the step at which the game finishes. T can be different for each episode, i.e. each time we play the game. So now

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{T-t-1} r_T.$$

Again our value function is defined as the expected return

$$v_\pi(s_t) = \mathbb{E}_\pi \left[G_t \mid S = s_t \right]$$

for our chosen policy. But this expected value will (have to) be the empirical mean return rather than some theoretical mean, since we aren't given the probabilities.

In doing a simulation there are two important points that we have to address:

- Can we be sure that we will visit each state enough times to be able to get a decent estimate of the expectation? The standard deviation of the estimated expectation for a state decreases in proportion to the inverse of the square root of the number of episodes for which we've been in that state.
- What do we do if we visit a state more than once during the same episode? For example in a game of chess. There are two approaches...

Suppose we have the following three episodes:

$$s_1 \xrightarrow{r_1} s_2 \xrightarrow{r_2} s_1 \xrightarrow{r_3} s_3 \xrightarrow{r_4} s_1 \xrightarrow{r_5} \text{terminates},$$

$$s_2 \xrightarrow{r_6} s_4 \xrightarrow{r_7} s_2 \xrightarrow{r_8} s_1 \xrightarrow{r_9} \text{terminates},$$

$$s_1 \xrightarrow{r_{10}} s_2 \xrightarrow{r_{11}} \text{terminates}.$$

The first is interpreted as start in state s_1 and move to state s_2 picking up a reward of r_1 *en route* etc.

First-visit evaluation We calculate the average for a state by looking at total return over many episodes and dividing by the number of episodes in which you went through that state. Do not divide by number of *times* you went through the state, you might visit the state several times in one game.

For the above three episodes, without any discounting, we get:

$$v(s_1) = \frac{1}{3}\left((r_1 + r_2 + r_3 + r_4 + r_5) + (r_9) + (r_{10} + r_{11})\right).$$

The three in the denominator is because s_1 was visited in all three episodes. And I have separated the rewards from those three episodes using parentheses.

$$v(s_2) = \frac{1}{3}\left((r_2 + r_3 + r_4 + r_5) + (r_6 + r_7 + r_8 + r_9) + (r_{11})\right),$$

$$v(s_3) = \frac{1}{1}\left((r_4 + r_5) + (0) + (0)\right)$$

and

$$v(s_4) = \frac{1}{1}\left((0) + (r_7 + r_8 + r_9) + (0)\right).$$

In practice we'd have a lot more than three episodes.

Every-visit evaluation The alternative is to count every visit to a state and every reward from that state from that point until termination. This means that some rewards will count several times. For the above three episodes we would have the following.

$$v(s_1) = \frac{1}{5}\left((r_1 + r_2 + r_3 + r_4 + r_5 + r_3 + r_4 + r_5 + r_5) + \right.$$
$$\left. (r_9) + (r_{10} + r_{11})\right).$$

State s_1 is visited five times in the three episodes.

$$v(s_2) = \frac{1}{4}\left((r_2 + r_3 + r_4 + r_5) + \right.$$
$$\left. (r_6 + r_7 + r_8 + r_9 + r_8 + r_9) + (r_{11})\right).$$

And the remaining two are the same as under first-visit since neither states s_3 nor s_4 are visited more than once in any episode.

While we are just valuing a given policy, as opposed to finding the optimal policy, is doesn't matter if we don't visit all of the states. We only have to visit those states which can be reached by the given policy.

Updating the expectations

The mean value that we want to calculate is just the simple arithmetic average of all values so far. That takes the form

$$v(s_t) = \frac{1}{n(s_t)} \sum_{i=1}^{n(s_t)} G_t^{(i)} \tag{11.5}$$

where I've used $G_t^{(i)}$ to mean the return realised in the i^{th} episode in which we have experienced state s_t, and $n(s_t)$ is the total number of episodes in which we have experienced state s_t so far. At the end of the next episode in which we visit state s_t we'll get an updated expectation

$$\begin{aligned} v_{\text{new}}(s_t) &= \frac{1}{n(s_t)+1} \sum_{i=1}^{n(s_t)+1} G_t^{(i)} \\ &= v_{\text{old}}(s_t) + \beta \left(G_t^{(n(s_t)+1)} - v_{\text{old}}(s_t) \right), \end{aligned}$$

where $\beta = \frac{1}{n(s_t)+1}$.

This is a simple updating that only requires remembering the current value and the relevant number of episodes (in which we have reached that state). But it can be simplified further.

What would it mean if instead of having β depending on the number $n(s_t)$ we instead made it a small parameter? This would tie in with the sort of updating we've see in other numerical techniques, gradient descent for example. Would it be cheating? It would certainly make things a bit easier, not having to keep track of $n(s_t)$.

Well, it wouldn't matter as long as the environment is not changing, and that β did eventually decrease to zero. If the environment is stationary then either way we would converge to the right average.

But if the environment *is* changing, i.e. is non stationary, then we should use an average that takes this into account. And that what's the exponentially weighted average does. Of course, then there is the matter of choosing the parameter β that reflects the timescale over which the environment is changing, the right half life. But that's a modelling issue.

The rule for updating thus becomes

$$v(s_t) \leftarrow v(s_t) + \beta \left(G_t - v(s_t) \right). \tag{11.6}$$

The parameter β is again the learning rate.

We will see many techniques now that use this idea of a learning rate for updating our value functions as new information/samples come along.

Notice how neither of (11.5) or (11.6) contain any vs other than the one we are updating, i.e. only $v(s_t)$. That is going to be very different when we come to other methods.

11.17 Temporal Difference Learning

I said earlier that a disadvantage of Monte Carlo policy evaluation is that you have to wait until termination of an episode before updating our values.

We get in our car, start the motor and a warning sign comes up on the dashboard. In Monte Carlo policy valuation we don't let this affect our expected journey time (our value) until we get to our destination (or not).

But we've experienced this warning light a couple of times before. In temporal difference (TD) learning our expected journey time is possibly immediately changed.

Notice that in both MC and TD I am not saying that we change our policy (go to the garage for example), because at the moment we are still in policy-evaluation mode.

TD learning uses what is called online updating. This means that updating is done during the episode, we don't wait until the end to update values. In contrast offline updating is when we wait until the end of an episode before updating values.

We don't need an episode to terminate before learning from it. In fact TD learning doesn't even need the sequences to be episodic. Unlike MC we can learn from neverending sequences.

The TD target and TD error

Instead of moving $v(s_t)$ towards G_t as in (11.6), we replace G_t with

$$r_{t+1} + \gamma v(s_{t+1}).$$

This is called the TD target. And here $v(s_{t1})$ is the sample that we have found so far, not the true expected return. (Which means that this can be a source of bias.) So the updating rule is

$$v(s_t) \leftarrow v(s_t) + \beta \left(r_{t+1} + \gamma v(s_{t+1}) - v(s_t) \right).$$

The $r_{t+1} + \gamma v(s_{t+1}) - v(s_t)$ is called the TD error.

Whereas MC policy evaluation calculates the average return in the obvious way, by adding up all returns for each state over many episodes and dividing by number of episodes in which that state was experienced — the rather obvious method — TD learning trickles down from the next state to the current state while adding the immediate reward. This makes it an application of the Bellman equation.

This method differs in that the updating rule takes information not just from $v(s_t)$ but also from $v(s_{t+1})$, the next state in our episode.

11.18 Pros And Cons: MC v TD

	Monte Carlo	Temporal Difference
Pros	Easy to understand; Zero bias (uses actual returns)	Does not require termination (can use unfinished sequences); Can be more efficient than MC; Low variance
Cons	Requires termination/episodic; High variance (adds up all rewards)	Biased; Sensitive to initial values;

11.19 Finding The Optimal Policy

We now move on to the control problem, finding the optimal policy. We are going to bring together the known-environment dynamic-programming ideas and the unknown-environment policy-evaluation ideas all seen above to address the problem of control in an unknown environment.

Our first observation is that using the state-value function $v(s)$ alone is not going to work in an unknown environment. That's because we can't evaluate which is the best action to take since we don't know where the actions will take us. In Blackjack terms we don't know whether it's better to hit a 15 when the dealer shows an eight or to stand, because we don't know where hitting 15 will take us, the next state. For this reason we are going to be working with the action-value function $Q(s, a)$.

Avoiding getting stuck

One issue we will have to address frequently is how to avoid getting stuck in a policy that is not optimal. If we always choose what seems to be the best policy then we might be premature. We might not have explored other actions sufficiently, or perhaps have an inaccurate estimate of values. If we always choose what we think is optimal it is called *greedy*.

It is much better to spend some time exploring the environment before homing in on what you think might be optimal. A fully greedy policy might be be counterproductive. We might be lucky to win when we first stand on a 14 when the dealer has an eight. We would then never experience hitting 14 when the dealer has an eight.

A simple way of avoiding getting stuck is to use ϵ-greedy exploration. This means that with probability $1 - \epsilon$, for some chosen ϵ, you take the greedy action but with probability ϵ you choose any one of the possible actions, all being equally likely. In that way you will explore the suboptimal solutions, you never know they might eventually turn out to be optimal.

Of course, as our policy improves we will need to decrease ϵ otherwise we will forever have some randomness in our policy.

11.20 Sarsa

We've seen enough updating rules now that I can cut straight to the chase. The method is called sarsa, which stands for "state action reward state action."

Update the action-value according to

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \beta \left(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right). \quad (11.7)$$

This updating is done at every step of every episode. The actions that are chosen are from an ϵ-greedy policy.

Let's go through the algorithm.

The Sarsa Algorithm

Step 0: Initialize

First choose starting values for $Q(s, a)$. Although the *rate* of convergence will not be affected by this choice the time to convergence to within some error might. Perhaps make everything zero.

Step 1: Start an episode

Start a new episode, with the initial state s.

Step 2: Move to the next state

In s choose an action using the ϵ-greedy method and the current values of $Q(s, a)$. This gives you a. Take the action a and move to next state, s'. In state s' choose the next action a' based on the values of $Q(s', a')$ for the possible actions at the new state and using the ϵ-greedy method.

Step 3: Update $Q(s, a)$

Update the action-value function at (s, a) according to the next reward you get, r and the action-value function at (s', a') with the chosen a'.

Hence (11.7)

$$Q(s,a) \leftarrow Q(s,a) + \beta\left(r + \gamma Q(s',a') - Q(s,a)\right)$$

and

$$s \leftarrow s' \quad \text{and} \quad a \leftarrow a'.$$

Here I have emphasised the updating to the state and the action as well. See Figure 11.13.

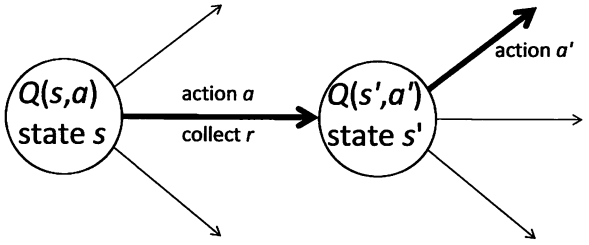


Figure 11.13: The quantities used in the sarsa algorithm.

Repeat Steps 2 and 3 until termination of the episode and then return to Step 1 for a new episode.

You can see where the name comes from, s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}.

On policy More jargon. Sarsa is an example of an on-policy method. This means that we are following a policy while also valuing or optimizing it.

Off policy We'll be looking at an off-policy method next. An off-policy method is one where you value one policy while following a different one. Again, Blackjack comes to the rescue... imagine tossing a coin to decide to hit or stand. We could follow that policy, even though it's probably going to be rubbish, and it would still give us information about whether hit or stand is better in each state and thus eventually lead to the optimal policy. The main reason for employing off-policy methods is because they can be used for exploration and ultimately finding optimality without getting tied down to anything that prematurely looks optimal.

11.21 Q Learning

A special case of an off-policy method is Q learning. In Q learning we follow one policy, the behaviour policy, which gives us our route through the states. But we learn from a different policy. We peek ahead to the next state and look at all the actions we could take, update our Q state-value function according to the best action we could have taken and then perversely possibly take a different action. I say perversely but I don't really mean it. It's like saying that you've got an idea which is the best route to the shops but you are going to experiment with a different route. Ok, you might take longer this time, but also you might find a better route. Taking a new route isn't going to make the best possible time any worse.

The Q Learning Algorithm

Step 0: Initialize

First choose starting values for $Q(s,a)$. Although the *rate* of convergence will not be affected by this choice the time to convergence to within some error might. Perhaps make everything zero.

Step 1: Start an episode

Start a new episode, with the initial state s.

Step 2: Move to the next state

In s choose an action using the behaviour policy. For example, the ϵ-greedy method and the current values of $Q(s,a)$. This gives you a. Take the action a and move to next state, s'.

Step 3: Update $Q(s,a)$

Update your current $Q(s,a)$ using the best action you could have taken:

$$Q(s,a) \leftarrow Q(s,a) + \beta\left(r + \gamma \max_{a''} Q(s',a'') - Q(s,a)\right)$$

and

$$s \leftarrow s'.$$

I've used a'' to mean all the actions you could possibly take. The action taken according to the behaviour policy would be a' which

does not appear in the updating.
In Figure 11.14 we see the algorithm in action. The actual action taken at s', a', is faint because it doesn't appear in the updating.

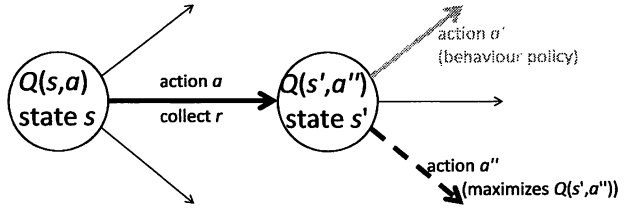


Figure 11.14: The quantities used in the Q-learning algorithm.

Repeat Steps 2 and 3 until termination of the episode and then return to Step 1 for a new episode.

11.22 Example: Blackjack

I'm going to be looking at Blackjack in detail throughout the rest of this chapter. It's a good example, often found in reinforcement-learning text books, because it is simple to understand, not trivial, but easy to code up.

First the rules of Blackjack as played in casinos.

1. There is one dealer/banker/house. In a casino the dealer will be standing at the typically kidney-shaped Blackjack table

2. There are one or more players, seated around said table

3. Each player places their bet in front of them before any cards are dealt

4. The dealer deals one card face up to each player and one to themselves. Another face-up card is then dealt to all players and another card, this time face down (the hole card), to the dealer

5. The count of a hand is the sum of the values of all the individual cards: Aces are one or 11, 2–9 count as the number of pips, 10, Jack, Queen and King count as ten

6. The goal of each player is to beat the dealer, i.e. get a higher count than the dealer

7. Each player in turn gets to take (Hit) another card, an offer that is repeated, or Stand
8. If a player goes over 21 they have bust and lose their bet immediately
9. If a player has an Ace which they are treating as 11 for the count and then a hit takes them over 21 then they simply treat that Ace as a value one going forward
10. A hand without any ace, or a hand in which aces have a value of one, is called a hard hand. A hand in which there is an ace that is counted as 11 is called a soft hand
11. If a player is dealt a natural meaning an Ace plus a 10-count card in their first two cards then, unless the dealer also has a natural, they are usually paid off at 3:2 immediately
12. A dealer's natural beats a player who has 21 with three or more cards
13. A player can double their bet before taking any cards, on condition that they then only take one extra card
14. If a player is initially dealt two cards of the same value they can be split to make two new hands. The player puts up another bet of the same size and the dealer deals another card to each of the two hands
15. If the dealer's upcard is an Ace then he can offer insurance. This is a side bet paying 2:1 if his hole card is a ten count
16. Once all players in turn have stood or gone bust it is the dealer's turn
17. The dealer in a casino does not have the flexibility in whether to hit or stand that a player has. Typically they have to hit a count of 16 or less and stand on 17 or more
18. If the player beats the dealer they get paid off at evens
19. If player and dealer have the same final count then it is a tie or push

Those are the basic rules. In some casinos these rules are fine tuned:

18. In some casinos a dealer with a count of 17 including an Ace will hit. So the dealer will hit a soft 17
19. In some casinos the dealer's hole card isn't dealt until after all players have played
20. In some casinos if a dealer has an upcard that is an Ace (sometimes also a 10) then he will look at his hole card to see if he has a natural

You can see from the rules that the player has a lot of decisions to make: Hit; Stand; Split; Take out insurance; Count Ace as one or 11; And how much to bet. This is all to the player's advantage, they can optimize their actions. The 3:2 payoff for a natural is also to the player's advantage. On the other hand the dealer is constrained to always play the same way: Hit 16 or less; Stand on 17 or more. Yet despite all this asymmetry in favour of the player the bank still has a significant edge. And that's because there is one asymmetry in favour of the house: If the player goes bust they lose their bet, even if the dealer goes bust later.

States and actions

We are to going to approach Blackjack as a problem in reinforcement learning.

(But let me be clear, this is (probably) not how you would approach this problem in practice. There is something special about the game of Blackjack that is going to be totally ignored in the approach below. And that is how from a hard card count of X and taking a card you can't go backwards to a card count of less than or equal to X. A more natural approach is to figure out what is best to do with a count of X, and once that has been done look at a count of $X-1$. That is, iteratively look at the optimal strategy working backwards. Rather like how I explained O&Xs. If you only read one thing from my further readings in this book then make it Ed Thorp's *Beat the Dealer*.)

We are going to list the states of play, actions that we can take, and try and find the optimal policy. Technically we could treat this as a dynamic programming problem in a known environment because we could, if we had the inclination, calculate the probabilities of transitions betweens states. But that is rather messy, and defeats the point of the exercise.

I'm going to simplify things a little bit below. Just so we don't get bogged down in the details. For example let's forget about doubling, splitting and insurance. I'm also not going to worry about the player getting a natural because at that point there is no decision to be made. I shall also have the dealer hitting soft 17 which is common practice.

States: How complicated do you want to get? To make this a proper MDP we need to know the following information: Our current count; Whether we have at least one Ace; The dealer's upcard; How many Aces, 2–9s, and 10-count cards have been dealt from this shoe (and how many decks were in the shoe to start with). That's a lot of information to keep track of. To simplify things for now let's assume that the dealer is using an infinite num-

ber of decks. That means that the probability of dealing any particular card never changes. The state is thus represented by our current count, whether we have at least one Ace and the dealer's upcard.

How many states are there? The dealer could have A–10. We could have 12–21 (we would always hit 11 or less). Although we would never hit 21 we need to know whether we have 21 because we might reach that state from another state. And we may or may not have an Ace that's valued at one. That makes a total of $10 \times 10 \times 2 = 200$ states. That's fewer than Noughts and Crosses.

Actions: Hit or stand. That's it.

Now let's look at the techniques I've described above.

Monte Carlo policy evaluation

Let's start with Monte Carlo policy evaluation.

To evaluate a policy we need a policy to evaluate. This is *not* about finding the optimal policy.

An obvious player policy is to do just what the dealer would do: Hit 16 or less and soft 17s, otherwise stand.

To code this up you will need

1. An implementation of the player strategy
2. An implementation of the dealer strategy
3. An implementation of dealing cards
4. An implementation of hard and soft hands
5. Two grids of 10×10, one for the state-value function for hard hands and one for soft hands
6. If you want to take arithmetic averages for finding the state-value functions you will also need two similar grids to store the number of episodes that have seen each state
7. Scissors and glue for cutting and pasting (only kidding!)

Let's play one game (an episode) and see what it looks like. We shall follow the same policy as the dealer.

Suppose the dealer has a 7 and the player is dealt A, 2, 4, 6, 5, then stands, and beats the dealer who goes bust. We can represent the state by

(Player Count, Hard or Soft, Dealer). So this episode is

$$(13, \text{Soft}, 7) \overset{r=0}{\to} (17, \text{Soft}, 7) \overset{r=0}{\to} (13, \text{Hard}, 7)$$

$$\overset{r=0}{\to} (18, \text{Hard}, 7) \quad \text{Terminates with reward of } 1.$$

There are no intermediate rewards and the final reward of 1 then gets added to the state-value function, v(Player Count, Hard or Soft, Dealer), *for each of those four states*, changing the averages for those four states.

Note that we had to run the episode all the way to termination otherwise the state-value function would miss out on the important, and here only, reward, that at the end.

Each of

$$v(13, \text{Soft}, 7), \quad v(17, \text{Soft}, 7), \quad v(13, \text{Hard}, 7) \text{ and } v(18, \text{Hard}, 7)$$

will be updated, independently of each other and other states.

This is exactly as described in Section 11.16. There is almost never any difference between first-visit and every-visit evaluation here because in Blackjack it's uncommon to revisit a state. (You'd have to have two or more Aces in your hand and then go over 21.)

In order to calculate the averages for the state-value function I chose to run all the simulations first before calculating the averages, rather than using a learning-rate parameter.

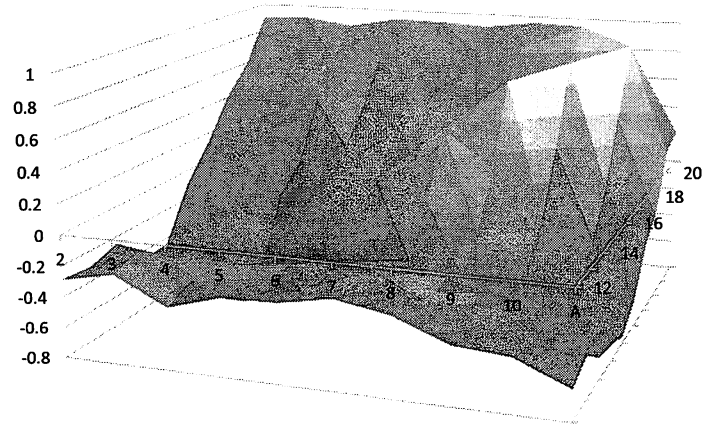


Figure 11.15: The state-value, $v(s)$, function for hard hands for the policy in which the player follows the same rules as the dealer: MC.

In Figure 11.15 is shown the state-value function for hard hands after a few thousand episodes.

Please note that I am not claiming that these are the final results, or that they are correct. Before going to a casino read the book mentioned at the end of this chapter. By the way, if you follow the dealer's strategy you are going to lose quite quickly. (Look at all the negative numbers in 11.15.)

Without knowing the transition probabilities, that is without knowing the environment (as we are assuming), then we cannot use the information in Figure 11.15 to tell us how to improve our policy from the one we have adopted. For that we need the action-value function. That will come soon.

TD learning

Changing the code to implement TD for policy evaluation is quite simple. The big difference between this and MC is that the updating takes information from the next state in the episode.

The updating rule is

$$v(s_t) \leftarrow v(s_t) + \beta \left(r_{t+1} + \gamma v(s_{t+1}) - v(s_t) \right).$$

In our problem we have $\gamma = 1$.

Instead of averaging all the rewards that each state gets, independently of other states, we use the next reward and the next state. The next state is a substitute for all the rewards that are to come (after the immediate one).

Let's see an example.

We shall follow the same policy as the dealer again. And we'll use the same example as before: The dealer has a 7 and the player is dealt A, 2, 4, 6, 5 and beats the dealer. This is represented by the episode

$$(13, \text{Soft}, 7) \overset{r=0}{\rightarrow} (17, \text{Soft}, 7) \overset{r=0}{\rightarrow} (13, \text{Hard}, 7)$$

$$\overset{r=0}{\rightarrow} (18, \text{Hard}, 7) \quad \text{Terminates with reward of } 1.$$

Going through the updating slowly we do the following, in order:

$$v(13, \text{Soft}, 7) \leftarrow v(13, \text{Soft}, 7) + \beta \left(0 + v(17, \text{Soft}, 7) - v(13, \text{Soft}, 7) \right).$$

(The $v(17, \text{Soft}, 7)$ at this point is whatever it was for the previous episode.) Then

$$v(17, \text{Soft}, 7) \leftarrow v(17, \text{Soft}, 7) + \beta \left(0 + v(13, \text{Hard}, 7) - v(17, \text{Soft}, 7) \right).$$

(Now $v(17, \text{Soft}, 7)$ is being updated, but it's still the old $v(13, \text{Hard}, 7)$.) Then

$$v(13, \text{Hard}, 7) \leftarrow v(13, \text{Hard}, 7) + \beta \left(0 + v(18, \text{Hard}, 7) - v(13, \text{Hard}, 7) \right).$$

And finally

$$v(18, \text{Hard}, 7) \leftarrow v(18, \text{Hard}, 7) + \beta\left(1 - v(18, \text{Hard}, 7)\right).$$

The state-value function for hard hands will be similar to that shown in Figure 11.15.

Now let's move on to finding the *optimal* policy for Blackjack.

Sarsa

Finding the optimal policy using sarsa first involves going over to the action-value function: Q(Player, Hard/Soft, Dealer, Hit/Stand).

The updating rule is now

$$Q(s,a) \leftarrow Q(s,a) + \beta\left(r + \gamma Q(s',a') - Q(s,a)\right),$$

with $\gamma = 1$ and the reward being zero until termination. And I have used an ϵ-greedy policy to determine which action to take in each state.

For example, suppose that the play is again

$$(13, \text{Soft}, 7) \overset{r=0}{\rightarrow} (17, \text{Soft}, 7) \overset{r=0}{\rightarrow} (13, \text{Hard}, 7)$$

$$\overset{r=0}{\rightarrow} (18, \text{Hard}, 7) \quad \text{Terminates with reward of } 1.$$

The updating would involve all of

$$Q((13, \text{Soft}, 7), \text{Hit}), \quad Q((13, \text{Soft}, 7), \text{Stand}), \quad Q((17, \text{Soft}, 7), \text{Hit}),$$

$$Q((17, \text{Soft}, 7), \text{Stand}), \quad Q((13, \text{Hard}, 7), \text{Hit}),$$

$$Q((13, \text{Hard}, 7), \text{Stand}), \quad Q((18, \text{Hard}, 7), \text{Hit})$$

$$\text{and} \quad Q((18, \text{Hard}, 7), \text{Stand}).$$

(The multiple parentheses in the above are to emphasise the difference between states and actions.) The Hits would be compared with the Stands for each state to decide on the action. And the state-action pairs that were realised would be updated.

The updating works as follows:

1. We start in state $(13, \text{Soft}, 7)$
2. Then the value of $Q((13, \text{Soft}, 7), \text{Hit})$ versus $Q((13, \text{Soft}, 7), \text{Stand})$ tells us whether to hit or stand. Although with probability ϵ we toss a coin to make that decision. We decide to hit
3. We move to state $(17, \text{Soft}, 7)$, collecting no reward

4. $Q((17, \text{Soft}, 7), \text{Hit})$ versus $Q((17, \text{Soft}, 7), \text{Stand})$ tells us whether to hit or stand. Although with probability ϵ we toss a coin to make that decision. We decide to hit

5. Etc.

I have stopped at the etc. because already we have enough information to start updating:

$$Q((13, \text{Soft}, 7), \text{Hit}) \leftarrow Q((13, \text{Soft}, 7), \text{Hit}) +$$
$$\beta \left(0 + Q((17, \text{Soft}, 7), \text{Hit}) - Q((13, \text{Soft}, 7), \text{Hit})\right).$$

This particular updating is illustrated in Figure 11.16. Each cell would contain the current estimates for the action-value function Q, as a function of the state and the action. There'd be a second grid, not shown, for the hard hands. The dashed lines represent the order in which the game is played. We go from a Soft 13 and Hit to a Soft 17 and another Hit. The second Hit takes us onto the Hard grid, that's why I've drawn it leaving the soft grid. The bent line represents the flow of information in the updating. We hit when we held Soft 13, so $Q((13, \text{Soft}, 7), \text{Hit})$ is updated. And it is updated using the action-value function where we went to, which was Soft 17, and the action we took there, a Hit, i.e. $Q((17, \text{Soft}, 7), \text{Hit})$.

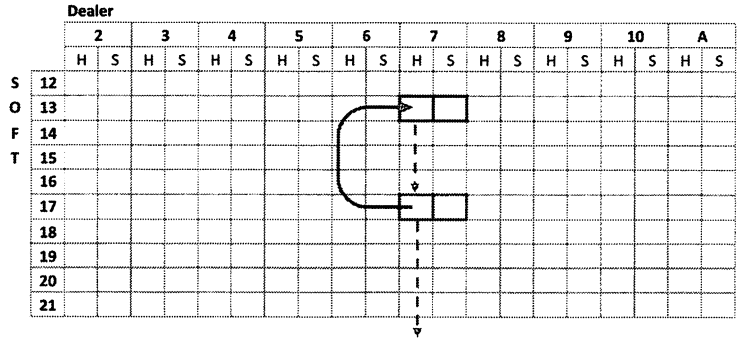


Figure 11.16: One part of the updating for a hand of Blackjack: Sarsa.

And there would be similar updating for each of the other realised state-action pairs. The final one would be

$$Q((18, \text{Hard}, 7), \text{Stand}) \leftarrow Q((18, \text{Hard}, 7), \text{Stand}) +$$
$$\beta \left(1 - Q((18, \text{Hard}, 7), \text{Stand})\right).$$

Hit or Stand		Dealer 2	3	4	5	6	7	8	9	10	A
	12	H	H	H	H	H	H	H	H	H	H
H	13	H	S	S	S	S	H	H	H	H	H
A	14	S	S	S	S	S	H	H	H	H	H
R	15	S	S	S	S	S	H	H	H	H	H
D	16	S	S	S	S	S	H	H	H	H	H
	17	S	S	S	S	S	S	S	S	S	S
	18	S	S	S	S	S	S	S	S	S	S
	19	S	S	S	S	S	S	S	S	S	S
	20	S	S	S	S	S	S	S	S	S	S
	21	S	S	S	S	S	S	S	S	S	S

Figure 11.17: Results from sarsa for hard hands. The recognised optimal strategy for hitting given by the boxes.

Results for hard hands after a few million episodes are shown in Figure 11.17. My results are labelled as H or S for Hit or Stand, and the Hits are grey. All I've done here is take the Q function and label the cell according to which is greater the Q for Hit or the Q for Stand in that state. The results are homing in on the classical, recognised optimal strategy for hitting given by the bordered boxes in that figure. I would need to run it for longer to perfect convergence. You can find the classical results in any decent book on Blackjack strategy.

Q learning

The updating rule is now

$$Q(s,a) \leftarrow Q(s,a) + \beta \left(r + \gamma \max_{a''} Q(s', a'') - Q(s,a) \right).$$

Use an ϵ-greedy policy to determine which action to take in each state. But note that the updating rule does not *explicitly* mention the action a' taken at state s because we are looking over all possible actions in state s'. Of course a' is in there *implicitly* because that's how we got to state s'.

Suppose that the play is yet again

$$(13, \text{Soft}, 7) \overset{r=0}{\rightarrow} (17, \text{Soft}, 7) \overset{r=0}{\rightarrow} (13, \text{Hard}, 7)$$

$$\overset{r=0}{\rightarrow} (18, \text{Hard}, 7) \quad \text{Terminates with reward of } 1.$$

The updating works as follows:

1. We start in state $(13, \text{Soft}, 7)$

2. Then the value of $Q((13, \text{Soft}, 7), \text{Hit})$ versus $Q((13, \text{Soft}, 7), \text{Stand})$ tells us whether to hit or stand. Although with probability ϵ we toss a coin to make that decision. We decide to hit

3. We move to state $(17, \text{Soft}, 7)$, collecting no reward

4. We take the maximum of $Q((17, \text{Soft}, 7), \text{Hit})$ and $Q((17, \text{Soft}, 7), \text{Stand})$, even though we might not take the maximizing action

5. Etc.

And so:

$$Q((13, \text{Soft}, 7), \text{Hit}) \leftarrow Q((13, \text{Soft}, 7), \text{Hit}) +$$

$$\beta \left(0 + \max \left(Q((17, \text{Soft}, 7), \text{Hit}), Q((17, \text{Soft}, 7), \text{Stand})\right)\right.$$

$$\left. - Q((13, \text{Soft}, 7), \text{Hit})\right).$$

And there would be similar updating for each of the other realised state-action pairs.

This particular updating is shown in Figure 11.18. This differs from the equivalent Sarsa diagram, 11.16, in two ways:

- First, we don't care whether our action in state Soft 17 is Hit or Stand, hence the question marks in the diagram.

- Second, the information that gets passed to $Q((13, \text{Soft}, 7), \text{Hit})$ is the maximum over the possible action-value functions in state Soft 17.

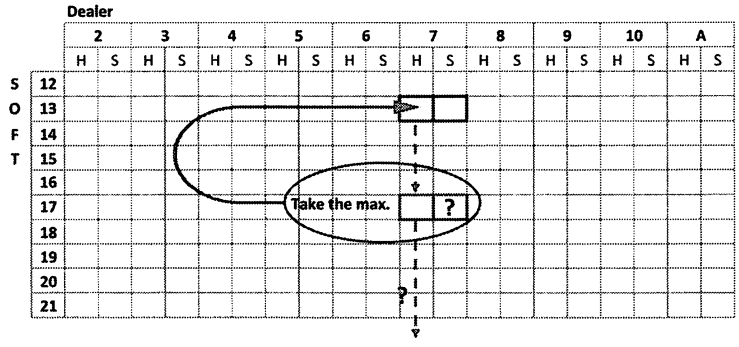


Figure 11.18: One part of the updating for a hand of Blackjack: Q learning.

How to win at Blackjack

I cannot leave you without giving a few clues as to how to actually win at Blackjack.

We've seen how you can find the optimal policy with sarsa and Q learning. But *still* that's not enough.

To win at Blackjack you need, as well as the optimum strategy,

- Finite number of decks (the fewer the better)
- Dealer deals sufficiently far down into the deck
- Player must keep track of extra variable(s)

With the infinite deck we've looked at so far the deck has no memory, the probabilities never change. With a finite number of decks the probabilities change as cards are drawn from the deck. We can take advantage of this by keeping track of more than the player's count, soft/hard and dealer's card.

The memory, as in which cards have been dealt and thus which cards are left in the shoe, is important because it changes the odds of winning. You can see this easily by considering what happens if the shoe is rich in high-count cards (because lots of twos, threes,... have been dealt). Remember that the dealer has to hit 12–16. If the deck has many tens then that means there is a higher chance than normal of him going bust.

So if, as the player, you have kept track of what has been dealt and you find the deck starting to look favourable then you simply increase your bet size.

No one expects you to keep track of everything that has been dealt. So various simple techniques have been devised to help you track the most important information. One way of doing this is to keep a running count in your head of the following cards, Aces, twos,..., sixes, tens as follows.

With a fresh shoe the running count is zero. Every time you see a two–six dealt (to other players, yourself, or to the dealer) you add 1 to that running count. And for every Ace or ten you see you subtract one. So if a round goes like this: an Ace, a two, two threes, a six, a nine, an eight, one Jack, one Queen and a seven, then the running count would go $-1, 0, 1, 2, 3, 3, 3, 2, 1, 1$, ending up at plus one. I.e. the deck is now ever so slightly better than before. And you keep this count going, only ever going back to zero for a new shoe. If the running count gets large enough relative to the number of cards left in the shoe then it's time to increase your bet size. (Of course then you get thrown out of the casino because the dealer is probably also counting cards, as it's called, and now he knows you're some kinda wise guy.)

Optimal betting

I said above that you have to adjust your bet, increasing it when the deck is favourable and decreasing the bet when it isn't. Then you might be able to win. There is some mathematics to this, going under the name of Kelly criterion and Kelly fraction. There's a simple formula, you can derive it on the back of an envelope. However since this isn't really a book about gambling I'm going to stop here, and direct you to the book listed at the end of this chapter.

11.23 Large State Spaces

My examples have all had relatively small state spaces, and just a small choice of actions. More often than not in real problems you will be faced with very large state spaces and/or actions. For example, you might be working with a complex game such as chess. Often the state space will have infinite dimensions, for example if there is a continuum of states. Or there will be a continuum of actions.

We can approach such a problem in reinforcement learning by approximating our value functions using simple or complicated functional forms with a set of parameters. These parameters are the quantities that are updated during the learning process.

We must convert the potentially infinite-dimensional states to finite-dimensional feature vectors. This process is called feature extraction.

How you best approximate the value functions is going to be problem dependent. At the simplest level you might use a linear approximation. If the problem is more complicated then perhaps approximate with a neural network.

Further Reading

There is just one book you will need for reinforcement learning, the classic and recently updated *Reinforcement Learning: An Introduction* by Richard Sutton and Andrew G. Barto.

But the one book you must get, although it's nothing about machine learning, is *Beat The Dealer* by Ed Thorp, published by Random House. It's a classic if you are into card games, or gambling, or finance, or to learn about other methods for approaching the mathematics of Blackjack. Or just because it's a great read.

Datasets

I've used a variety of data sets in this book to demonstrate the various methods. Some of these data sets are available to the public and some were private. I only occasionally adjusted the data and then only to identify and explain some idea as compactly as possible. I am not claiming that whatever method I employed on a particular problem was the best for that problem. But equally they were probably not the worst.

Good places to start looking for data are
`https://toolbox.google.com/datasetsearch`
and `https://www.kaggle.com`. You'll need to register with the latter.

Here are some datasets you might want to download to play around with. Some I used, some I didn't.

Titanic

A very popular dataset for machine-learning problems is that for passengers on the Titanic, and in particular survivorship based on information such as gender, class of ticket, etc. Go to
`https://www.kaggle.com/c/titanic`

I decided not to use this data as I thought it a bit tasteless.

Irises

The famous iris dataset that I used can be found everywhere, for example
`https://www.kaggle.com/uciml/iris`

Heights and weights

Ten thousand heights and weights of adults can be found here
`https://www.kaggle.com/mustafaali96/weight-height`

MNIST

Seventy-thousand handwritten images of digits can be found at
`http://yann.lecun.com/exdb/mnist/`
Or rather matrix representations of the images written by employees of the American Census Bureau and American high school students in CSV format.

House of Commons Voting

The site `https://www.publicwhip.org.uk` collects data for activity in the UK's Houses of Commons and Lords. I used some of it for my SOM analysis. Dig down to
`https://www.publicwhip.org.uk/project/data.php`
and `https://www.publicwhip.org.uk/data/` to get at the raw data.

Mushrooms

Fancy predicting whether a mushroom is edible or poisonous? Well, pop over to
`https://www.kaggle.com/uciml/mushroom-classification`
This contains data for over 8,000 mushrooms with information about their geometry, colours, etc. and whether they are edible. Rather you than me.

Financial

Plenty of historical stock, index and exchange rate data can be found at
`https://finance.yahoo.com`

For specific stocks go to
`https://finance.yahoo.com/quote/[STOCKSYMBOLHERE]/history/`

The Bank of England has economic indicators, interest rates etc. at
`https://www.bankofengland.co.uk/statistics/`

Banknotes

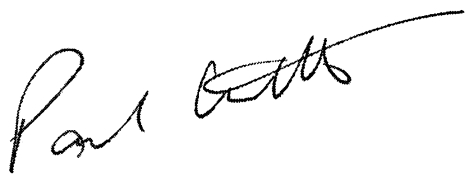
Data for characteristics of banknotes can be found here
`http://archive.ics.uci.edu/ml/datasets/banknote+authentication`

Animals

Data for characteristics of various animals can be found here
`http://archive.ics.uci.edu/ml/datasets/zoo`

Epilogue

I hope you enjoyed that as much as I did. My job is over, so I can sit down with that single malt. But for you the work is only just beginning. Please don't be a stranger, email me, `paul@wilmott.com`. Perhaps you'll find typos, or maybe even major mistakes. Or maybe you'll want clarification of some technical point or have ideas for new material. Probably you'll end up teaching me, and that's how it should be. You can also sign up at `wilmott.com`, it's free! There we can all get together to shoot the breeze, and plan the fightback for when the machines take over.



P.S. I'll leave the final words to The Machine, from which (whom?) I have learned so much.

Final Words

Oh boy! It was a very enjoyable shoot. Thanks for reading! What did you guys think of the new series? Was this a fun experience or did it take some time to get started?

What makes a good leader? A clear vision and focus. Leadership is what really matters — the vision of the future. It matters how you work it.

To truly understand how we're going to get there, we need to understand how you are going to be able to execute on your vision. Don't let a lack of leadership distract you from your true purpose. Your first task is to define where you are in this new world, and then to make those changes.

The new world is the new vision.

It is not the future, but it's the future.

Index

Numbers in bold are pages with major entries about the subject.

Made in the USA
Middletown, DE
28 February 2020

MACHINE LEARNING: AN APPLIED MATHEMATICS INTRODUCTION

Machine Learning: An Applied Mathematics Introduction covers the essential mathematics behind all of the following topics

- *K* Nearest Neighbours
- *K* Means Clustering
- Naïve Bayes Classifier
- Regression Methods
- Support Vector Machines
- Self-Organizing Maps
- Decision Trees
- Neural Networks
- Reinforcement Learning

Paul Wilmott brings three decades of experience in mathematics education, and his inimitable style, to the hottest of subjects. This book is an accessible introduction for anyone who wants to understand the foundations but also wants to "get to the meat without having to eat too many vegetables."

Paul Wilmott has been called "cult derivatives lecturer" by the *Financial Times* and "financial mathematics guru" by the BBC.

ISBN 978-1-9160816-0-4
51999
9 781916 081604

Panda Ohana
PUBLISHING