Chapman & Hall/CRC
Machine Learning & Pattern Recognition Series

MACHINE LEARNING

An Algorithmic Perspective

SECOND EDITION

STEPHEN MARSLAND

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

MACHINE LEARNING

An Algorithmic Perspective

SECOND EDITION

Chapman & Hall/CRC
Machine Learning & Pattern Recognition Series

SERIES EDITORS

Ralf Herbrich
Amazon Development Center
Berlin, Germany

Thore Graepel
Microsoft Research Ltd.
Cambridge, UK

AIMS AND SCOPE

This series reflects the latest advances and applications in machine learning and pattern recognition through the publication of a broad range of reference works, textbooks, and handbooks. The inclusion of concrete examples, applications, and methods is highly encouraged. The scope of the series includes, but is not limited to, titles in the areas of machine learning, pattern recognition, computational intelligence, robotics, computational/statistical learning theory, natural language processing, computer vision, game AI, game theory, neural networks, computational neuroscience, and other relevant topics, such as machine learning applied to bioinformatics or cognitive science, which might be proposed by potential contributors.

PUBLISHED TITLES

BAYESIAN PROGRAMMING
Pierre Bessière, Emmanuel Mazer, Juan-Manuel Ahuactzin, and Kamel Mekhnacha

UTILITY-BASED LEARNING FROM DATA
Craig Friedman and Sven Sandow

HANDBOOK OF NATURAL LANGUAGE PROCESSING, SECOND EDITION
Nitin Indurkhya and Fred J. Damerau

COST-SENSITIVE MACHINE LEARNING
Balaji Krishnapuram, Shipeng Yu, and Bharat Rao

COMPUTATIONAL TRUST MODELS AND MACHINE LEARNING
Xin Liu, Anwitaman Datta, and Ee-Peng Lim

MULTILINEAR SUBSPACE LEARNING: DIMENSIONALITY REDUCTION OF MULTIDIMENSIONAL DATA
Haiping Lu, Konstantinos N. Plataniotis, and Anastasios N. Venetsanopoulos

MACHINE LEARNING: An Algorithmic Perspective, Second Edition
Stephen Marsland

A FIRST COURSE IN MACHINE LEARNING
Simon Rogers and Mark Girolami

MULTI-LABEL DIMENSIONALITY REDUCTION
Liang Sun, Shuiwang Ji, and Jieping Ye

ENSEMBLE METHODS: FOUNDATIONS AND ALGORITHMS
Zhi-Hua Zhou

Chapman & Hall/CRC
Machine Learning & Pattern Recognition Series

MACHINE LEARNING

An Algorithmic Perspective

SECOND EDITION

STEPHEN MARSLAND

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20140826

International Standard Book Number-13: 978-1-4665-8333-7 (eBook - PDF)

Again, for Monika

Contents

Prologue to 2nd Edition

There have been some interesting developments in machine learning over the past four years, since the 1st edition of this book came out. One is the rise of Deep Belief Networks as an area of real research interest (and business interest, as large internet-based companies look to snap up every small company working in the area), while another is the continuing work on statistical interpretations of machine learning algorithms. This second one is very good for the field as an area of research, but it does mean that computer science students, whose statistical background can be rather lacking, find it hard to get started in an area that they are sure should be of interest to them. The hope is that this book, focussing on the *algorithms* of machine learning as it does, will help such students get a handle on the ideas, and that it will start them on a journey towards mastery of the relevant mathematics and statistics as well as the necessary programming and experimentation.

In addition, the libraries available for the Python language have continued to develop, so that there are now many more facilities available for the programmer. This has enabled me to provide a simple implementation of the Support Vector Machine that can be used for experiments, and to simplify the code in a few other places. All of the code that was used to create the examples in the book is available at `http://stephenmonika.net/` (in the 'Book' tab), and use and experimentation with any of this code, as part of any study on machine learning, is strongly encouraged.

Some of the changes to the book include:

- the addition of two new chapters on two of those new areas: Deep Belief Networks (Chapter 17) and Gaussian Processes (Chapter 18).
- a reordering of the chapters, and some of the material within the chapters, to make a more natural flow.
- the reworking of the Support Vector Machine material so that there is running code and the suggestions of experiments to be performed.
- the addition of Random Forests (as Section 13.3), the Perceptron convergence theorem (Section 3.4.1), a proper consideration of accuracy methods (Section 2.2.4), conjugate gradient optimisation for the MLP (Section 9.3.2), and more on the Kalman filter and particle filter in Chapter 16.
- improved code including better use of naming conventions in Python.
- various improvements in the clarity of explanation and detail throughout the book.

I would like to thank the people who have written to me about various parts of the book, and made suggestions about things that could be included or explained better. I would also like to thank the students at Massey University who have studied the material with me, either as part of their coursework, or as first steps in research, whether in the theory or the application of machine learning. Those that have contributed particularly to the content of the second edition include Nirosha Priyadarshani, James Curtis, Andy Gilman, Örjan

Ekeberg, and the Osnabrück Knowledge-Based Systems Research group, especially Joachim Hertzberg, Sven Albrecht, and Thomas Wieman.

Stephen Marsland
Ashhurst, New Zealand

Prologue to 1st Edition

One of the most interesting features of machine learning is that it lies on the boundary of several different academic disciplines, principally computer science, statistics, mathematics, and engineering. This has been a problem as well as an asset, since these groups have traditionally not talked to each other very much. To make it even worse, the areas where machine learning methods can be applied vary even more widely, from finance to biology and medicine to physics and chemistry and beyond. Over the past ten years this inherent multi-disciplinarity has been embraced and understood, with many benefits for researchers in the field. This makes writing a textbook on machine learning rather tricky, since it is potentially of interest to people from a variety of different academic backgrounds.

In universities, machine learning is usually studied as part of artificial intelligence, which puts it firmly into computer science and—given the focus on algorithms—it certainly fits there. However, understanding why these algorithms work requires a certain amount of statistical and mathematical sophistication that is often missing from computer science undergraduates. When I started to look for a textbook that was suitable for classes of undergraduate computer science and engineering students, I discovered that the level of mathematical knowledge required was (unfortunately) rather in excess of that of the majority of the students. It seemed that there was a rather crucial gap, and it resulted in me writing the first draft of the student notes that have become this book. The emphasis is on the algorithms that make up the machine learning methods, and on understanding how and why these algorithms work. It is intended to be a practical book, with lots of programming examples and is supported by a website that makes available all of the code that was used to make the figures and examples in the book. The website for the book is: `http://stephenmonika.net/MLbook.html`.

For this kind of practical approach, examples in a real programming language are preferred over some kind of pseudocode, since it enables the reader to run the programs and experiment with data without having to work out irrelevant implementation details that are specific to their chosen language. Any computer language can be used for writing machine learning code, and there are very good resources available in many different languages, but the code examples in this book are written in Python. I have chosen Python for several reasons, primarily that it is freely available, multi-platform, relatively nice to use and is becoming a default for scientific computing. If you already know how to write code in any other programming language, then you should not have many problems learning Python. If you don't know how to code at all, then it is an ideal first language as well. Chapter A provides a basic primer on using Python for numerical computing.

Machine learning is a rich area. There are lots of very good books on machine learning for those with the mathematical sophistication to follow them, and it is hoped that this book could provide an entry point to students looking to study the subject further as well as those studying it as part of a degree. In addition to books, there are many resources for machine learning available via the Internet, with more being created all the time. The Machine Learning Open Source Software website at `http://mloss.org/software/` provides links to a host of software in different languages.

There is a very useful resource for machine learning in the UCI Machine Learning Repos-

itory (`http://archive.ics.uci.edu/ml/`). This website holds lots of datasets that can be downloaded and used for experimenting with different machine learning algorithms and seeing how well they work. The repository is going to be the principal source of data for this book. By using these test datasets for experimenting with the algorithms, we do not have to worry about getting hold of suitable data and preprocessing it into a suitable form for learning. This is typically a large part of any real problem, but it gets in the way of learning about the algorithms.

I am very grateful to a lot of people who have read sections of the book and provided suggestions, spotted errors, and given encouragement when required. In particular for the first edition, thanks to Zbigniew Nowicki, Joseph Marsland, Bob Hodgson, Patrick Rynhart, Gary Allen, Linda Chua, Mark Bebbington, JP Lewis, Tom Duckett, and Monika Nowicki. Thanks especially to Jonathan Shapiro, who helped me discover machine learning and who may recognise some of his own examples.

Stephen Marsland
Ashhurst, New Zealand

Introduction

Suppose that you have a website selling software that you've written. You want to make the website more personalised to the user, so you start to collect data about visitors, such as their computer type/operating system, web browser, the country that they live in, and the time of day they visited the website. You can get this data for any visitor, and for people who actually buy something, you know what they bought, and how they paid for it (say PayPal or a credit card). So, for each person who buys something from your website, you have a list of data that looks like (computer type, web browser, country, time, software bought, how paid). For instance, the first three pieces of data you collect could be:

- Macintosh OS X, Safari, UK, morning, SuperGame1, credit card
- Windows XP, Internet Explorer, USA, afternoon, SuperGame1, PayPal
- Windows Vista, Firefox, NZ, evening, SuperGame2, PayPal

Based on this data, you would like to be able to populate a 'Things You Might Be Interested In' box within the webpage, so that it shows software that might be relevant to each visitor, based on the data that you can access while the webpage loads, i.e., computer and OS, country, and the time of day. Your hope is that as more people visit your website and you store more data, you will be able to identify trends, such as that Macintosh users from New Zealand (NZ) love your first game, while Firefox users, who are often more knowledgeable about computers, want your automatic download application and virus/internet worm detector, etc.

Once you have collected a large set of such data, you start to examine it and work out what you can do with it. The problem you have is one of prediction: given the data you have, predict what the next person will buy, and the reason that you think that it might work is that people who seem to be similar often act similarly. So how can you actually go about solving the problem? This is one of the fundamental problems that this book tries to solve. It is an example of what is called supervised learning, because we know what the right answers are for some examples (the software that was actually bought) so we can give the learner some examples where we know the right answer. We will talk about supervised learning more in Section 1.3.

1.1 IF DATA HAD MASS, THE EARTH WOULD BE A BLACK HOLE

Around the world, computers capture and store terabytes of data every day. Even leaving aside your collection of MP3s and holiday photographs, there are computers belonging to shops, banks, hospitals, scientific laboratories, and many more that are storing data incessantly. For example, banks are building up pictures of how people spend their money,

hospitals are recording what treatments patients are on for which ailments (and how they respond to them), and engine monitoring systems in cars are recording information about the engine in order to detect when it might fail. The challenge is to do something useful with this data: if the bank's computers can learn about spending patterns, can they detect credit card fraud quickly? If hospitals share data, then can treatments that don't work as well as expected be identified quickly? Can an intelligent car give you early warning of problems so that you don't end up stranded in the worst part of town? These are some of the questions that machine learning methods can be used to answer.

Science has also taken advantage of the ability of computers to store massive amounts of data. Biology has led the way, with the ability to measure gene expression in DNA microarrays producing immense datasets, along with protein transcription data and phylogenetic trees relating species to each other. However, other sciences have not been slow to follow. Astronomy now uses digital telescopes, so that each night the world's observatories are storing incredibly high-resolution images of the night sky; around a terabyte per night. Equally, medical science stores the outcomes of medical tests from measurements as diverse as magnetic resonance imaging (MRI) scans and simple blood tests. The explosion in stored data is well known; the challenge is to do something useful with that data. The Large Hadron Collider at CERN apparently produces about 25 petabytes of data per year.

The size and complexity of these datasets mean that humans are unable to extract useful information from them. Even the way that the data is stored works against us. Given a file full of numbers, our minds generally turn away from looking at them for long. Take some of the same data and plot it in a graph and we can do something. Compare the table and graph shown in Figure 1.1: the graph is rather easier to look at and deal with. Unfortunately, our three-dimensional world doesn't let us do much with data in higher dimensions, and even the simple webpage data that we collected above has four different features, so if we plotted it with one dimension for each feature we'd need four dimensions! There are two things that we can do with this: reduce the number of dimensions (until our simple brains can deal with the problem) or use computers, which don't know that high-dimensional problems are difficult, and don't get bored with looking at massive data files of numbers. The two pictures in Figure 1.2 demonstrate one problem with reducing the number of dimensions (more technically, projecting it into fewer dimensions), which is that it can hide useful information and make things look rather strange. This is one reason why machine learning is becoming so popular — the problems of our human limitations go away if we can make computers do the dirty work for us. There is one other thing that can help if the number of dimensions is not too much larger than three, which is to use glyphs that use other representations, such as size or colour of the datapoints to represent information about some other dimension, but this does not help if the dataset has 100 dimensions in it.

In fact, you have probably interacted with machine learning algorithms at some time. They are used in many of the software programs that we use, such as Microsoft's infamous paperclip in Office (maybe not the most positive example), spam filters, voice recognition software, and lots of computer games. They are also part of automatic number-plate recognition systems for petrol station security cameras and toll roads, are used in some anti-skid braking and vehicle stability systems, and they are even part of the set of algorithms that decide whether a bank will give you a loan.

The attention-grabbing title to this section would only be true if data was very heavy. It is very hard to work out how much data there actually is in all of the world's computers, but it was estimated in 2012 that was about 2.8 zettabytes (2.8×10^{21} bytes), up from about 160 exabytes (160×10^{18} bytes) of data that were created and stored in 2006, and projected to reach 40 zettabytes by 2020. However, to make a black hole the size of the earth would

x_1	x_2	Class
0.1	1	1
0.15	0.2	2
0.48	0.6	3
0.1	0.6	1
0.2	0.15	2
0.5	0.55	3
0.2	1	1
0.3	0.25	2
0.52	0.6	3
0.3	0.6	1
0.4	0.2	2
0.52	0.5	3

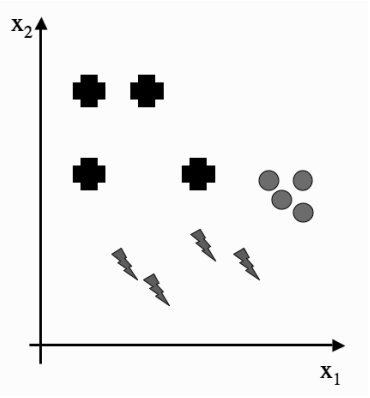

FIGURE 1.1 A set of datapoints as numerical values and as points plotted on a graph. It is easier for us to visualise data than to see it in a table, but if the data has more than three dimensions, we can't view it all at once.

FIGURE 1.2 Two views of the same two wind turbines (Te Apiti wind farm, Ashhurst, New Zealand) taken at an angle of about 30° to each other. The two-dimensional projections of three-dimensional objects hides information.

take a mass of about 40×10^{35} grams. So data would have to be so heavy that you couldn't possibly lift a data pen, let alone a computer before the section title were true! However, and more interestingly for machine learning, the same report that estimated the figure of 2.8 zettabytes ('*Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East*' by John Gantz and David Reinsel and sponsored by EMC Corporation) also reported that while a quarter of this data could produce useful information, only around 3% of it was tagged, and less that 0.5% of it was actually used for analysis!

1.2 LEARNING

Before we delve too much further into the topic, let's step back and think about what learning actually is. The key concept that we will need to think about for our machines is learning from data, since data is what we have; terabytes of it, in some cases. However, it isn't too large a step to put that into human behavioural terms, and talk about learning from experience. Hopefully, we all agree that humans and other animals can display behaviours that we label as intelligent by learning from experience. Learning is what gives us flexibility in our life; the fact that we can adjust and adapt to new circumstances, and learn new tricks, no matter how old a dog we are! The important parts of animal learning for this book are remembering, adapting, and generalising: recognising that last time we were in this situation (saw this data) we tried out some particular action (gave this output) and it worked (was correct), so we'll try it again, or it didn't work, so we'll try something different. The last word, generalising, is about recognising similarity between different situations, so that things that applied in one place can be used in another. This is what makes learning useful, because we can use our knowledge in lots of different places.

Of course, there are plenty of other bits to intelligence, such as reasoning, and logical deduction, but we won't worry too much about those. We are interested in the most fundamental parts of intelligence—learning and adapting—and how we can model them in a computer. There has also been a lot of interest in making computers reason and deduce facts. This was the basis of most early Artificial Intelligence, and is sometimes known as symbolic processing because the computer manipulates symbols that reflect the environment. In contrast, machine learning methods are sometimes called subsymbolic because no symbols or symbolic manipulation are involved.

1.2.1 Machine Learning

Machine learning, then, is about making computers modify or adapt their actions (whether these actions are making predictions, or controlling a robot) so that these actions get more accurate, where accuracy is measured by how well the chosen actions reflect the correct ones. Imagine that you are playing Scrabble (or some other game) against a computer. You might beat it every time in the beginning, but after lots of games it starts beating you, until finally you never win. Either you are getting worse, or the computer is learning how to win at Scrabble. Having learnt to beat you, it can go on and use the same strategies against other players, so that it doesn't start from scratch with each new player; this is a form of generalisation.

It is only over the past decade or so that the inherent multi-disciplinarity of machine learning has been recognised. It merges ideas from neuroscience and biology, statistics, mathematics, and physics, to make computers learn. There is a fantastic existence proof that learning is possible, which is the bag of water and electricity (together with a few trace chemicals) sitting between your ears. In Section 3.1 we will have a brief peek inside and see

if there is anything we can borrow/steal in order to make machine learning algorithms. It turns out that there is, and neural networks have grown from exactly this, although even their own father wouldn't recognise them now, after the developments that have seen them reinterpreted as statistical learners. Another thing that has driven the change in direction of machine learning research is data mining, which looks at the extraction of useful information from massive datasets (by men with computers and pocket protectors rather than pickaxes and hard hats), and which requires efficient algorithms, putting more of the emphasis back onto computer science.

The computational complexity of the machine learning methods will also be of interest to us since what we are producing is algorithms. It is particularly important because we might want to use some of the methods on very large datasets, so algorithms that have high-degree polynomial complexity in the size of the dataset (or worse) will be a problem. The complexity is often broken into two parts: the complexity of training, and the complexity of applying the trained algorithm. Training does not happen very often, and is not usually time critical, so it can take longer. However, we often want a decision about a test point quickly, and there are potentially lots of test points when an algorithm is in use, so this needs to have low computational cost.

1.3 TYPES OF MACHINE LEARNING

In the example that started the chapter, your webpage, the aim was to predict what software a visitor to the website might buy based on information that you can collect. There are a couple of interesting things in there. The first is the data. It might be useful to know what software visitors have bought before, and how old they are. However, it is not possible to get that information from their web browser (even cookies can't tell you how old somebody is), so you can't use that information. Picking the variables that you want to use (which are called features in the jargon) is a very important part of finding good solutions to problems, and something that we will talk about in several places in the book. Equally, choosing how to process the data can be important. This can be seen in the example in the time of access. Your computer can store this down to the nearest millisecond, but that isn't very useful, since you would like to spot similar patterns between users. For this reason, in the example above I chose to quantise it down to one of the set `morning, afternoon, evening, night`; obviously I need to ensure that these times are correct for their time zones, too.

We are going to loosely define learning as meaning getting better at some task through practice. This leads to a couple of vital questions: how does the computer know whether it is getting better or not, and how does it know how to improve? There are several different possible answers to these questions, and they produce different types of machine learning. For now we will consider the question of knowing whether or not the machine is learning. We can tell the algorithm the correct answer for a problem so that it gets it right next time (which is what would happen in the webpage example, since we know what software the person bought). We hope that we only have to tell it a few right answers and then it can 'work out' how to get the correct answers for other problems (generalise). Alternatively, we can tell it whether or not the answer was correct, but not how to find the correct answer, so that it has to search for the right answer. A variant of this is that we give a score for the answer, according to how correct it is, rather than just a 'right or wrong' response. Finally, we might not have any correct answers; we just want the algorithm to find inputs that have something in common.

These different answers to the question provide a useful way to classify the different algorithms that we will be talking about:

Supervised learning A training set of examples with the correct responses (targets) is provided and, based on this training set, the algorithm generalises to respond correctly to all possible inputs. This is also called learning from exemplars.

Unsupervised learning Correct responses are not provided, but instead the algorithm tries to identify similarities between the inputs so that inputs that have something in common are categorised together. The statistical approach to unsupervised learning is known as density estimation.

Reinforcement learning This is somewhere between supervised and unsupervised learning. The algorithm gets told when the answer is wrong, but does not get told how to correct it. It has to explore and try out different possibilities until it works out how to get the answer right. Reinforcement learning is sometime called learning with a critic because of this monitor that scores the answer, but does not suggest improvements.

Evolutionary learning Biological evolution can be seen as a learning process: biological organisms adapt to improve their survival rates and chance of having offspring in their environment. We'll look at how we can model this in a computer, using an idea of fitness, which corresponds to a score for how good the current solution is.

The most common type of learning is supervised learning, and it is going to be the focus of the next few chapters. So, before we get started, we'll have a look at what it is, and the kinds of problems that can be solved using it.

1.4 SUPERVISED LEARNING

As has already been suggested, the webpage example is a typical problem for supervised learning. There is a set of data (the training data) that consists of a set of input data that has target data, which is the answer that the algorithm should produce, attached. This is usually written as a set of data $(\mathbf{x}_i, \mathbf{t}_i)$, where the inputs are $\mathbf{x}_i$, the targets are $\mathbf{t}_i$, and the i index suggests that we have lots of pieces of data, indexed by i running from 1 to some upper limit N. Note that the inputs and targets are written in boldface font to signify vectors, since each piece of data has values for several different features; the notation used in the book is described in more detail in Section 2.1. If we had examples of every possible piece of input data, then we could put them together into a big look-up table, and there would be no need for machine learning at all. The thing that makes machine learning better than that is generalisation: the algorithm should produce sensible outputs for inputs that weren't encountered during learning. This also has the result that the algorithm can deal with noise, which is small inaccuracies in the data that are inherent in measuring any real world process. It is hard to specify rigorously what generalisation means, but let's see if an example helps.

1.4.1 Regression

Suppose that I gave you the following datapoints and asked you to tell me the value of the output (which we will call y since it is not a target datapoint) when $x = 0.44$ (here, x, t, and y are not written in boldface font since they are scalars, as opposed to vectors).

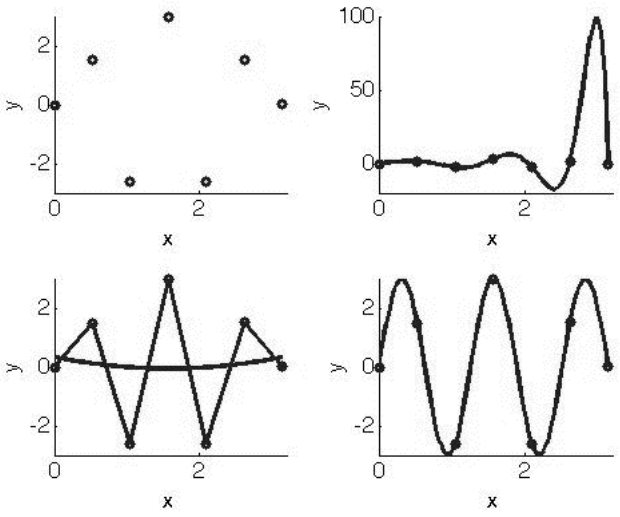

FIGURE 1.3 *Top left:* A few datapoints from a sample problem. *Bottom left:* Two possible ways to predict the values between the known datapoints: connecting the points with straight lines, or using a cubic approximation (which in this case misses all of the points). *Top and bottom right:* Two more complex approximators (see the text for details) that pass through the points, although the lower one is rather better than the top.

x	t
0	0
0.5236	1.5
1.0472	-2.5981
1.5708	3.0
2.0944	-2.5981
2.6180	1.5
3.1416	0

Since the value $x = 0.44$ isn't in the examples given, you need to find some way to predict what value it has. You assume that the values come from some sort of function, and try to find out what the function is. Then you'll be able to give the output value y for any given value of x. This is known as a regression problem in statistics: fit a mathematical function describing a curve, so that the curve passes as close as possible to all of the datapoints. It is generally a problem of function approximation or interpolation, working out the value between values that we know.

The problem is how to work out what function to choose. Have a look at Figure 1.3. The top-left plot shows a plot of the 7 values of x and y in the table, while the other plots show different attempts to fit a curve through the datapoints. The bottom-left plot shows two possible answers found by using straight lines to connect up the points, and also what happens if we try to use a cubic function (something that can be written as $ax^3 + bx^2 + cx + d = 0$). The top-right plot shows what happens when we try to match the function using a different polynomial, this time of the form $ax^{10} + bx^9 + \ldots + jx + k = 0$,

and finally the bottom-right plot shows the function $y = 3\sin(5x)$. Which of these functions would you choose?

The straight-line approximation probably isn't what we want, since it doesn't tell us much about the data. However, the cubic plot on the same set of axes is terrible: it doesn't get anywhere near the datapoints. What about the plot on the top-right? It looks like it goes through all of the datapoints exactly, but it is very wiggly (look at the value on the y-axis, which goes up to 100 instead of around three, as in the other figures). In fact, the data were made with the sine function plotted on the bottom-right, so that is the correct answer in this case, but the algorithm doesn't know that, and to it the two solutions on the right both look equally good. The only way we can tell which solution is better is to test how well they generalise. We pick a value that is between our datapoints, use our curves to predict its value, and see which is better. This will tell us that the bottom-right curve is better in the example.

So one thing that our machine learning algorithms can do is interpolate between datapoints. This might not seem to be intelligent behaviour, or even very difficult in two dimensions, but it is rather harder in higher dimensional spaces. The same thing is true of the other thing that our algorithms will do, which is classification—grouping examples into different classes—which is discussed next. However, the algorithms are learning by our definition if they adapt so that their performance improves, and it is surprising how often real problems that we want to solve can be reduced to classification or regression problems.

1.4.2 Classification

The classification problem consists of taking input vectors and deciding which of N classes they belong to, based on training from exemplars of each class. The most important point about the classification problem is that it is discrete — each example belongs to precisely one class, and the set of classes covers the whole possible output space. These two constraints are not necessarily realistic; sometimes examples might belong partially to two different classes. There are fuzzy classifiers that try to solve this problem, but we won't be talking about them in this book. In addition, there are many places where we might not be able to categorise every possible input. For example, consider a vending machine, where we use a neural network to learn to recognise all the different coins. We train the classifier to recognise all New Zealand coins, but what if a British coin is put into the machine? In that case, the classifier will identify it as the New Zealand coin that is closest to it in appearance, but this is not really what is wanted: rather, the classifier should identify that it is not one of the coins it was trained on. This is called novelty detection. For now we'll assume that we will not receive inputs that we cannot classify accurately.

Let's consider how to set up a coin classifier. When the coin is pushed into the slot, the machine takes a few measurements of it. These could include the diameter, the weight, and possibly the shape, and are the features that will generate our input vector. In this case, our input vector will have three elements, each of which will be a number showing the measurement of that feature (choosing a number to represent the shape would involve an encoding, for example that 1=circle, 2=hexagon, etc.). Of course, there are many other features that we could measure. If our vending machine included an atomic absorption spectroscope, then we could estimate the density of the material and its composition, or if it had a camera, we could take a photograph of the coin and feed that image into the classifier. The question of which features to choose is not always an easy one. We don't want to use too many inputs, because that will make the training of the classifier take longer (and also, as the number of input dimensions grows, the number of datapoints required increases

FIGURE 1.4 The New Zealand coins.

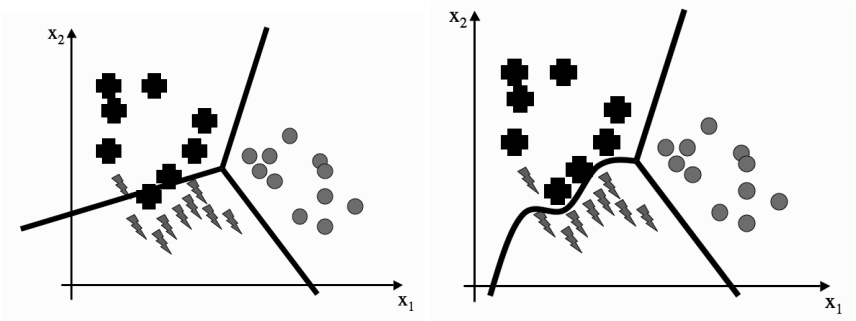

FIGURE 1.5 *Left:* A set of straight line decision boundaries for a classification problem. *Right:* An alternative set of decision boundaries that separate the plusses from the lightening strikes better, but requires a line that isn't straight.

faster; this is known as the curse of dimensionality and will be discussed in Section 2.1.2), but we need to make sure that we can reliably separate the classes based on those features. For example, if we tried to separate coins based only on colour, we wouldn't get very far, because the 20¢ and 50¢ coins are both silver and the \$1 and \$2 coins both bronze. However, if we use colour and diameter, we can do a pretty good job of the coin classification problem for NZ coins. There are some features that are entirely useless. For example, knowing that the coin is circular doesn't tell us anything about NZ coins, which are all circular (see Figure 1.4). In other countries, though, it could be very useful.

The methods of performing classification that we will see during this book are very different in the ways that they learn about the solution; in essence they aim to do the same thing: find decision boundaries that can be used to separate out the different classes. Given the features that are used as inputs to the classifier, we need to identify some values of those features that will enable us to decide which class the current input is in. Figure 1.5 shows a set of 2D inputs with three different classes shown, and two different decision boundaries; on the left they are straight lines, and are therefore simple, but don't categorise as well as the non-linear curve on the right.

Now that we have seen these two types of problem, let's take a look at the whole process of machine learning from the practitioner's viewpoint.

1.5 THE MACHINE LEARNING PROCESS

This section assumes that you have some problem that you are interested in using machine learning on, such as the coin classification that was described previously. It briefly examines the process by which machine learning algorithms can be selected, applied, and evaluated for the problem.

Data Collection and Preparation Throughout this book we will be in the fortunate position of having datasets readily available for downloading and using to test the algorithms. This is, of course, less commonly the case when the desire is to learn about some new problem, when either the data has to be collected from scratch, or at the very least, assembled and prepared. In fact, if the problem is completely new, so that appropriate data can be chosen, then this process should be merged with the next step of feature selection, so that only the required data is collected. This can typically be done by assembling a reasonably small dataset with all of the features that you believe might be useful, and experimenting with it before choosing the best features and collecting and analysing the full dataset.

Often the difficulty is that there is a large amount of data that *might* be relevant, but it is hard to collect, either because it requires many measurements to be taken, or because they are in a variety of places and formats, and merging it appropriately is difficult, as is ensuring that it is clean; that is, it does not have significant errors, missing data, etc.

For supervised learning, target data is also needed, which can require the involvement of experts in the relevant field and significant investments of time.

Finally, the quantity of data needs to be considered. Machine learning algorithms need significant amounts of data, preferably without too much noise, but with increased dataset size comes increased computational costs, and the sweet spot at which there is enough data without excessive computational overhead is generally impossible to predict.

Feature Selection An example of this part of the process was given in Section 1.4.2 when we looked at possible features that might be useful for coin recognition. It consists of identifying the features that are most useful for the problem under examination. This invariably requires prior knowledge of the problem and the data; our common sense was used in the coins example above to identify some potentially useful features and to exclude others.

As well as the identification of features that are useful for the learner, it is also necessary that the features can be collected without significant expense or time, and that they are robust to noise and other corruption of the data that may arise in the collection process.

Algorithm Choice Given the dataset, the choice of an appropriate algorithm (or algorithms) is what this book should be able to prepare you for, in that the knowledge of the underlying principles of each algorithm and examples of their use is precisely what is required for this.

Parameter and Model Selection For many of the algorithms there are parameters that have to be set manually, or that require experimentation to identify appropriate values. These requirements are discussed at the appropriate points of the book.

Training Given the dataset, algorithm, and parameters, training should be simply the use of computational resources in order to build a model of the data in order to predict the outputs on new data.

Evaluation Before a system can be deployed it needs to be tested and evaluated for accuracy on data that it was not trained on. This can often include a comparison with human experts in the field, and the selection of appropriate metrics for this comparison.

1.6 A NOTE ON PROGRAMMING

This book is aimed at helping you understand and use machine learning algorithms, and that means writing computer programs. The book contains algorithms in both pseudocode, and as fragments of Python programs based on NumPy (Appendix A provides an introduction to both Python and NumPy for the beginner), and the website provides complete working code for all of the algorithms.

Understanding how to use machine learning algorithms is fine in theory, but without testing the programs on data, and seeing what the parameters do, you won't get the complete picture. In general, writing the code for yourself is always the best way to check that you understand what the algorithm is doing, and finding the unexpected details.

Unfortunately, debugging machine learning code is even harder than general debugging – it is quite easy to make a program that compiles and runs, but just doesn't seem to actually learn. In that case, you need to start testing the program carefully. However, you can quickly get frustrated with the fact that, because so many of the algorithms are stochastic, the results are not repeatable anyway. This can be temporarily avoided by setting the random number seed, which has the effect of making the random number generator follow the same pattern each time, as can be seen in the following example of running code at the Python command line (marked as >>>), where the 10 numbers that appear after the seed is set are the same in both cases, and would carry on the same forever (there is more about the pseudo-random numbers that computers generate in Section 15.1.1):

```
>>> import numpy as np
>>> np.random.seed(4)
>>> np.random.rand(10)
array([ 0.96702984,  0.54723225,  0.97268436,  0.71481599,  0.69772882,
        0.2160895 ,  0.97627445,  0.00623026,  0.25298236,  0.43479153])
>>> np.random.rand(10)
array([ 0.77938292,  0.19768507,  0.86299324,  0.98340068,  0.16384224,
        0.59733394,  0.0089861 ,  0.38657128,  0.04416006,  0.95665297])
>>> np.random.seed(4)
>>> np.random.rand(10)
array([ 0.96702984,  0.54723225,  0.97268436,  0.71481599,  0.69772882,
        0.2160895 ,  0.97627445,  0.00623026,  0.25298236,  0.43479153])
```

This way, on each run the randomness will be avoided, and the parameters will all be the same.

Another thing that is useful is the use of 2D toy datasets, where you can plot things, since you can see whether or not something unexpected is going on. In addition, these

datasets can be made very simple, such as separable by a straight line (we'll see more of this in Chapter 3) so that you can see whether it deals with simple cases, at least.

Another way to 'cheat' temporarily is to include the target as one of the inputs, so that the algorithm really has no excuse for getting the wrong answer.

Finally, having a reference program that works and that you can compare is also useful, and I hope that the code on the book website will help people get out of unexpected traps and strange errors.

1.7 A ROADMAP TO THE BOOK

As far as possible, this book works from general to specific and simple to complex, while keeping related concepts in nearby chapters. Given the focus on algorithms and encouraging the use of experimentation rather than starting from the underlying statistical concepts, the book starts with some older, and reasonably simple algorithms, which are examples of supervised learning.

Chapter 2 follows up many of the concepts in this introductory chapter in order to highlight some of the overarching ideas of machine learning and thus the data requirements of it, as well as providing some material on basic probability and statistics that will not be required by all readers, but is included for completeness.

Chapters 3, 4, and 5 follow the main historical sweep of supervised learning using neural networks, as well as introducing concepts such as interpolation. They are followed by chapters on dimensionality reduction (Chapter 6) and the use of probabilistic methods like the EM algorithm and nearest neighbour methods (Chapter 7). The idea of optimal decision boundaries and kernel methods are introduced in Chapter 8, which focuses on the Support Vector Machine and related algorithms.

One of the underlying methods for many of the preceding algorithms, optimisation, is surveyed briefly in Chapter 9, which then returns to some of the material in Chapter 4 to consider the Multi-layer Perceptron purely from the point of view of optimisation. The chapter then continues by considering search as the discrete analogue of optimisation. This leads naturally into evolutionary learning including genetic algorithms (Chapter 10), reinforcement learning (Chapter 11), and tree-based learners (Chapter 12) which are search-based methods. Methods to combine the predictions of many learners, which are often trees, are described in Chapter 13.

The important topic of unsupervised learning is considered in Chapter 14, which focuses on the Self-Organising Feature Map; many unsupervised learning algorithms are also presented in Chapter 6.

The remaining four chapters primarily describe more modern, and statistically based, approaches to machine learning, although not all of the algorithms are completely new: following an introduction to Markov Chain Monte Carlo techniques in Chapter 15 the area of Graphical Models is surveyed, with comparatively old algorithms such as the Hidden Markov Model and Kalman Filter being included along with particle filters and Bayesian networks. The ideas behind Deep Belief Networks are given in Chapter 17, starting from the historical idea of symmetric networks with the Hopfield network. An introduction to Gaussian Processes is given in Chapter 18.

Finally, an introduction to Python and NumPy is given in Appendix A, which should be sufficient to enable readers to follow the code descriptions provided in the book and use the code supplied on the book website, assuming that they have some programming experience in any programming language.

I would suggest that Chapters 2 to 4 contain enough introductory material to be essential

for anybody looking for an introduction to machine learning ideas. For an introductory one semester course I would follow them with Chapters 6 to 8, and then use the second half of Chapter 9 to introduce Chapters 10 and 11, and then Chapter 14.

A more advanced course would certainly take in Chapters 13 and 15 to 18 along with the optimisation material in Chapter 9.

I have attempted to make the material reasonably self-contained, with the relevant mathematical ideas either included in the text at the appropriate point, or with a reference to where that material is covered. This means that the reader with some prior knowledge will certainly find some parts can be safely ignored or skimmed without loss.

FURTHER READING

For a different (more statistical and example-based) take on machine learning, look at:

- Chapter 1 of T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd edition, Springer, Berlin, Germany, 2008.

Other texts that provide alternative views of similar material include:

- Chapter 1 of R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*, 2nd edition, Wiley-Interscience, New York, USA, 2001.
- Chapter 1 of S. Haykin. *Neural Networks: A Comprehensive Foundation*, 2nd edition, Prentice-Hall, New Jersey, USA, 1999.

Preliminaries

This chapter has two purposes: to present some of the overarching important concepts of machine learning, and to see how some of the basic ideas of data processing and statistics arise in machine learning. One of the most useful ways to break down the effects of learning, which is to put it in terms of the statistical concepts of bias and variance, is given in Section 2.5, following on from a section where those concepts are introduced for the beginner.

2.1 SOME TERMINOLOGY

We start by considering some of the terminology that we will use throughout the book; we've already seen a bit of it in the Introduction. We will talk about inputs and input vectors for our learning algorithms. Likewise, we will talk about the outputs of the algorithm. The inputs are the data that is fed into the algorithm. In general, machine learning algorithms all work by taking a set of input values, producing an output (answer) for that input vector, and then moving on to the next input. The input vector will typically be several real numbers, which is why it is described as a vector: it is written down as a series of numbers, e.g., $(0.2, 0.45, 0.75, -0.3)$. The size of this vector, i.e., the number of elements in the vector, is called the dimensionality of the input. This is because if we were to plot the vector as a point, we would need one dimension of space for each of the different elements of the vector, so that the example above has 4 dimensions. We will talk about this more in Section 2.1.1.

We will often write equations in vector and matrix notation, with lowercase boldface letters being used for vectors and uppercase boldface letters for matrices. A vector $\mathbf{x}$ has elements $(x_1, x_2, \ldots, x_m)$. We will use the following notation in the book:

Inputs An input vector is the data given as one input to the algorithm. Written as $\mathbf{x}$, with elements x_i, where i runs from 1 to the number of input dimensions, m.

Weights w_{ij}, are the weighted connections between nodes i and j. For neural networks these weights are analogous to the synapses in the brain. They are arranged into a matrix $\mathbf{W}$.

Outputs The output vector is $\mathbf{y}$, with elements y_j, where j runs from 1 to the number of output dimensions, n. We can write $\mathbf{y}(\mathbf{x}, \mathbf{W})$ to remind ourselves that the output depends on the inputs to the algorithm and the current set of weights of the network.

Targets The target vector $\mathbf{t}$, with elements t_j, where j runs from 1 to the number of output dimensions, n, are the extra data that we need for supervised learning, since they provide the 'correct' answers that the algorithm is learning about.

Activation Function For neural networks, $g(\cdot)$ is a mathematical function that describes the firing of the neuron as a response to the weighted inputs, such as the threshold function described in Section 3.1.2.

Error E, a function that computes the inaccuracies of the network as a function of the outputs $\mathbf{y}$ and targets $\mathbf{t}$.

2.1.1 Weight Space

When working with data it is often useful to be able to plot it and look at it. If our data has only two or three input dimensions, then this is pretty easy: we use the x-axis for feature 1, the y-axis for feature 2, and the z-axis for feature 3. We then plot the positions of the input vectors on these axes. The same thing can be extended to as many dimensions as we like provided that we don't actually want to look at it in our 3D world. Even if we have 200 input dimensions (that is, 200 elements in each of our input vectors) then we can try to imagine it plotted by using 200 axes that are all mutually orthogonal (that is, at right angles to each other). One of the great things about computers is that they aren't constrained in the same way we are—ask a computer to hold a 200-dimensional array and it does it. Provided that you get the algorithm right (always the difficult bit!), then the computer doesn't know that 200 dimensions is harder than 2 for us humans.

We can look at projections of the data into our 3D world by plotting just three of the features against each other, but this is usually rather confusing: things can look very close together in your chosen three axes, but can be a very long way apart in the full set. You've experienced this in your 2D view of the 3D world; Figure 1.2 shows two different views of some wind turbines. The two turbines appear to be very close together from one angle, but are obviously separate from another.

As well as plotting datapoints, we can also plot anything else that we feel like. In particular, we can plot some of the parameters of a machine learning algorithm. This is particularly useful for neural networks (which we will start to see in the next chapter) since the parameters of a neural network are the values of a set of weights that connect the neurons to the inputs. There is a schematic of a neural network on the left of Figure 2.1, showing the inputs on the left, and the neurons on the right. If we treat the weights that get fed into one of the neurons as a set of coordinates in what is known as weight space, then we can plot them. We think about the weights that connect into a particular neuron, and plot the strengths of the weights by using one axis for each weight that comes into the neuron, and plotting the position of the neuron as the location, using the value of w_1 as the position on the 1st axis, the value of w_2 on the 2nd axis, etc. This is shown on the right of Figure 2.1.

Now that we have a space in which we can talk about how close together neurons and inputs are, since we can imagine positioning neurons and inputs in the same space by plotting the position of each neuron as the location where its weights say it should be. The two spaces will have the same dimension (providing that we don't use a bias node (see Section 3.3.2), otherwise the weight space will have one extra dimension) so we can plot the position of neurons in the input space. This gives us a different way of learning, since by changing the weights we are changing the location of the neurons in this weight space. We can measure distances between inputs and neurons by computing the Euclidean distance, which in two dimensions can be written as:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \tag{2.1}$$

So we can use the idea of neurons and inputs being 'close together' in order to decide

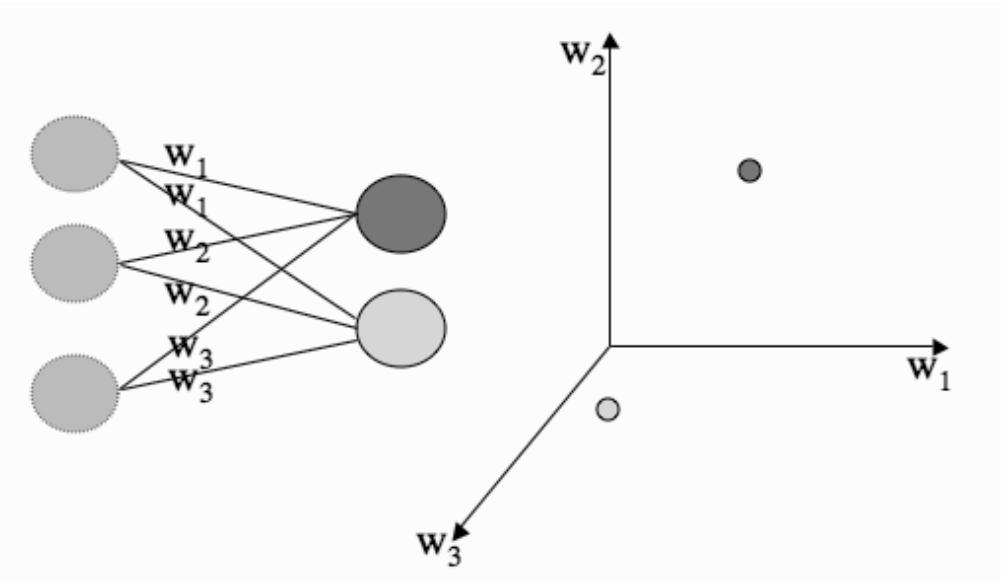

FIGURE 2.1 The position of two neurons in weight space. The labels on the network refer to the dimension in which that weight is plotted, not its value.

when a neuron should fire and when it shouldn't. If the neuron is close to the input in this sense then it should fire, and if it is not close then it shouldn't. This picture of weight space can be helpful for understanding another important concept in machine learning, which is what effect the number of input dimensions can have. The input vector is telling us everything we know about that example, and usually we don't know enough about the data to know what is useful and what is not (think back to the coin classification example in Section 1.4.2), so it might seem sensible to include all of the information that we can get, and let the algorithm sort out for itself what it needs. Unfortunately, we are about to see that doing this comes at a significant cost.

2.1.2 The Curse of Dimensionality

The curse of dimensionality is a very strong name, so you can probably guess that it is a bit of a problem. The essence of the curse is the realisation that as the number of dimensions increases, the volume of the unit hypersphere does not increase with it. The unit hypersphere is the region we get if we start at the origin (the centre of our coordinate system) and draw all the points that are distance 1 away from the origin. In 2 dimensions we get a circle of radius 1 around $(0, 0)$ (drawn in Figure 2.2), and in 3D we get a sphere around $(0, 0, 0)$ (Figure 2.3). In higher dimensions, the sphere becomes a hypersphere. The following table shows the size of the unit hypersphere for the first few dimensions, and the graph in Figure 2.4 shows the same thing, but also shows clearly that as the number of dimensions tends to infinity, so the volume of the hypersphere tends to zero.

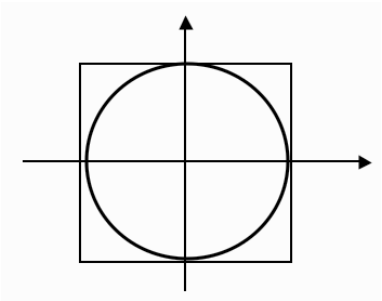

FIGURE 2.2 The unit circle in 2D with its bounding box.

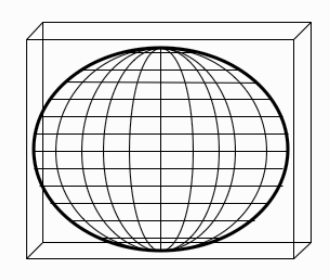

FIGURE 2.3 The unit sphere in 3D with its bounding cube. The sphere does not reach as far into the corners as the circle does, and this gets more noticeable as the number of dimensions increases.

Dimension	Volume
1	2.0000
2	3.1416
3	4.1888
4	4.9348
5	5.2636
6	5.1677
7	4.7248
8	4.0587
9	3.2985
10	2.5502

At first sight this seems completely counterintuitive. However, think about enclosing the hypersphere in a box of width 2 (between -1 and 1 along each axis), so that the box just touches the sides of the hypersphere. For the circle, almost all of the area inside the box is included in the circle, except for a little bit at each corner (see Figure 2.2) The same is true in 3D (Figure 2.3), but if we think about the 100-dimensional hypersphere (not necessarily something you want to imagine), and follow the diagonal line from the origin out to one of the corners of the box, then we intersect the boundary of the hypersphere when all the coordinates are 0.1. The remaining 90% of the line inside the box is outside the hypersphere, and so the volume of the hypersphere is obviously shrinking as the number of dimensions grows. The graph in Figure 2.4 shows that when the number of dimensions is above about 20, the volume is effectively zero. It was computed using the formula for the volume of the hypersphere of dimension n as $v_n = (2\pi/n)v_{n-2}$. So as soon as $n > 2\pi$, the volume starts to shrink.

The curse of dimensionality will apply to our machine learning algorithms because as the number of input dimensions gets larger, we will need more data to enable the algorithm to generalise sufficiently well. Our algorithms try to separate data into classes based on the features; therefore as the number of features increases, so will the number of datapoints we need. For this reason, we will often have to be careful about what information we give to the algorithm, meaning that we need to understand something about the data in advance.

Regardless of how many input dimensions there are, the point of machine learning is to

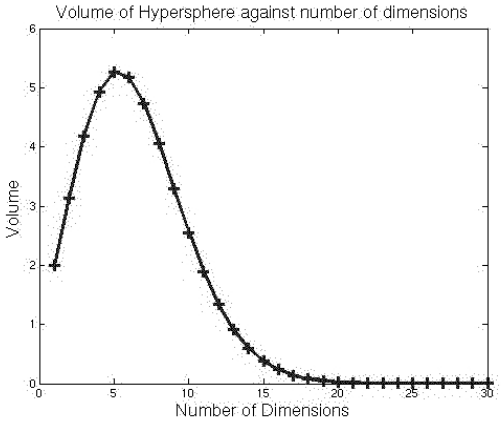

FIGURE 2.4 The volume of the unit hypersphere for different numbers of dimensions.

make predictions on data inputs. In the next section we consider how to evaluate how well an algorithm actually achieves this.

2.2 KNOWING WHAT YOU KNOW: TESTING MACHINE LEARNING ALGORITHMS

The purpose of learning is to get better at predicting the outputs, be they class labels or continuous regression values. The only real way to know how successfully the algorithm has learnt is to compare the predictions with known target labels, which is how the training is done for supervised learning. This suggests that one thing you can do is just to look at the error that the algorithm makes on the training set.

However, we want the algorithms to generalise to examples that were not seen in the training set, and we obviously can't test this by using the training set. So we need some different data, a test set, to test it on as well. We use this test set of (input, target) pairs by feeding them into the network and comparing the predicted output with the target, but we don't modify the weights or other parameters for them: we use them to decide how well the algorithm has learnt. The only problem with this is that it reduces the amount of data that we have available for training, but that is something that we will just have to live with.

2.2.1 Overfitting

Unfortunately, things are a little bit more complicated than that, since we might also want to know how well the algorithm is generalising as it learns: we need to make sure that we do enough training that the algorithm generalises well. In fact, there is at least as much danger in over-training as there is in under-training. The number of degrees of variability in most machine learning algorithms is huge — for a neural network there are lots of weights, and each of them can vary. This is undoubtedly more variation than there is in the function we are learning, so we need to be careful: if we train for too long, then we will overfit the data, which means that we have learnt about the noise and inaccuracies in the data as well as the actual function. Therefore, the model that we learn will be much too complicated, and won't be able to generalise.

Figure 2.5 shows this by plotting the predictions of some algorithm (as the curve) at

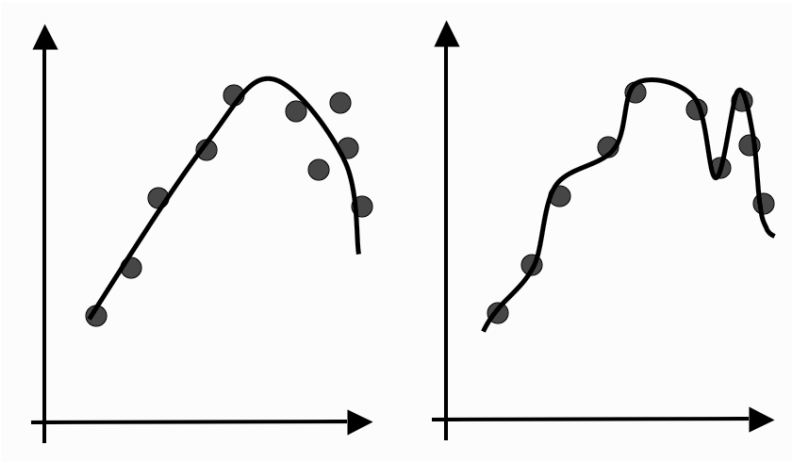

FIGURE 2.5 The effect of overfitting is that rather than finding the generating function (as shown on the left), the neural network matches the inputs perfectly, including the noise in them (on the right). This reduces the generalisation capabilities of the network.

two different points in the learning process. On the left of the figure the curve fits the overall trend of the data well (it has generalised to the underlying general function), but the training error would still not be that close to zero since it passes near, but not through, the training data. As the network continues to learn, it will eventually produce a much more complex model that has a lower training error (close to zero), meaning that it has memorised the training examples, including any noise component of them, so that is has overfitted the training data.

We want to stop the learning process before the algorithm overfits, which means that we need to know how well it is generalising at each timestep. We can't use the training data for this, because we wouldn't detect overfitting, but we can't use the testing data either, because we're saving that for the final tests. So we need a third set of data to use for this purpose, which is called the validation set because we're using it to validate the learning so far. This is known as cross-validation in statistics. It is part of model selection: choosing the right parameters for the model so that it generalises as well as possible.

2.2.2 Training, Testing, and Validation Sets

We now need three sets of data: the training set to actually train the algorithm, the validation set to keep track of how well it is doing as it learns, and the test set to produce the final results. This is becoming expensive in data, especially since for supervised learning it all has to have target values attached (and even for unsupervised learning, the validation and test sets need targets so that you have something to compare to), and it is not always easy to get accurate labels (which may well be why you want to learn about the data). The area of semi-supervised learning attempts to deal with this need for large amounts of labelled data; see the Further Reading section for some references.

Clearly, each algorithm is going to need some reasonable amount of data to learn from (precise needs vary, but the more data the algorithm sees, the more likely it is to have seen examples of each possible type of input, although more data also increases the computational time to learn). However, the same argument can be used to argue that the validation and

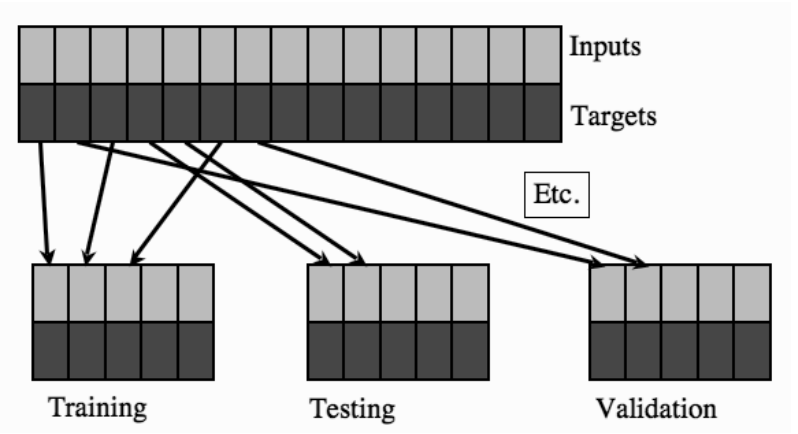

FIGURE 2.6 The dataset is split into different sets, some for training, some for validation, and some for testing.

test sets should also be reasonably large. Generally, the exact proportion of training to testing to validation data is up to you, but it is typical to do something like 50:25:25 if you have plenty of data, and 60:20:20 if you don't. How you do the splitting can also matter. Many datasets are presented with the first set of datapoints being in class 1, the next in class 2, and so on. If you pick the first few points to be the training set, the next the test set, etc., then the results are going to be pretty bad, since the training did not see all the classes. This can be dealt with by randomly reordering the data first, or by assigning each datapoint randomly to one of the sets, as is shown in Figure 2.6.

If you are really short of training data, so that if you have a separate validation set there is a worry that the algorithm won't be sufficiently trained; then it is possible to perform leave-some-out, multi-fold cross-validation. The idea is shown in Figure 2.7. The dataset is randomly partitioned into K subsets, and one subset is used as a validation set, while the algorithm is trained on all of the others. A different subset is then left out and a new model is trained on that subset, repeating the same process for all of the different subsets. Finally, the model that produced the lowest validation error is tested and used. We've traded off data for computation time, since we've had to train K different models instead of just one. In the most extreme case of this there is leave-one-out cross-validation, where the algorithm is validated on just one piece of data, training on all of the rest.

2.2.3 The Confusion Matrix

Regardless of how much data we use to test the trained algorithm, we still need to work out whether or not the result is good. We will look here at a method that is suitable for classification problems that is known as the confusion matrix. For regression problems things are more complicated because the results are continuous, and so the most common thing to use is the sum-of-squares error that we will use to drive the training in the following chapters. We will see these methods being used as we look at examples.

The confusion matrix is a nice simple idea: make a square matrix that contains all the possible classes in both the horizontal and vertical directions and list the classes along the top of a table as the predicted outputs, and then down the left-hand side as the targets. So for example, the element of the matrix at (i, j) tells us how many input patterns were put

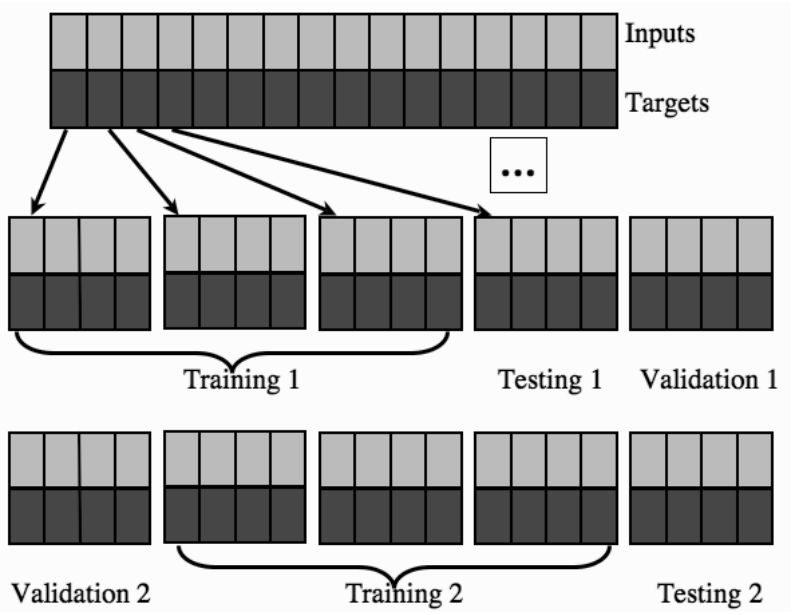

FIGURE 2.7 Leave-some-out, multi-fold cross-validation gets around the problem of data shortage by training many models. It works by splitting the data into sets, training a model on most sets and holding one out for validation (and another for testing). Different models are trained with different sets being held out.

into class i in the targets, but class j by the algorithm. Anything on the leading diagonal (the diagonal that starts at the top left of the matrix and runs down to the bottom right) is a correct answer. Suppose that we have three classes: C_1, C_2, and C_3. Now we count the number of times that the output was class C_1 when the target was C_1, then when the target was C_2, and so on until we've filled in the table:

	Outputs		
	C_1	C_2	C_3
C_1	5	1	0
C_2	1	4	1
C_3	2	0	4

This table tells us that, for the three classes, most examples were classified correctly, but two examples of class C_3 were misclassified as C_1, and so on. For a small number of classes this is a nice way to look at the outputs. If you just want one number, then it is possible to divide the sum of the elements on the leading diagonal by the sum of all of the elements in the matrix, which gives the fraction of correct responses. This is known as the accuracy, and we are about to see that it is not the last word in evaluating the results of a machine learning algorithm.

2.2.4 Accuracy Metrics

We can do more to analyse the results than just measuring the accuracy. If you consider the possible outputs of the classes, then they can be arranged in a simple chart like this

(where a true positive is an observation correctly put into class 1, while a false positive is an observation incorrectly put into class 1, while negative examples (both true and false) are those put into class 2):

True Positives	False Positives
False Negatives	True Negatives

The entries on the leading diagonal of this chart are correct and those off the diagonal are wrong, just as with the confusion matrix. Note, however, that this chart and the concepts of false positives, etc., are based on binary classification.

Accuracy is then defined as the sum of the number of true positives and true negatives divided by the total number of examples (where # means 'number of', and TP stands for True Positive, etc.):

$$\text{Accuracy} = \frac{\#TP + \#FP}{\#TP + \#FP + \#TN + \#FN}. \tag{2.2}$$

The problem with accuracy is that it doesn't tell us everything about the results, since it turns four numbers into just one. There are two complementary pairs of measurements that can help us to interpret the performance of a classifier, namely sensitivity and specificity, and precision and recall. Their definitions are shown next, followed by some explanation.

$$\text{Sensitivity} = \frac{\#TP}{\#TP + \#FN} \tag{2.3}$$

$$\text{Specificity} = \frac{\#TN}{\#TN + \#FP} \tag{2.4}$$

$$\text{Precision} = \frac{\#TP}{\#TP + \#FP} \tag{2.5}$$

$$\text{Recall} = \frac{\#TP}{\#TP + \#FN} \tag{2.6}$$

Sensitivity (also known as the true positive rate) is the ratio of the number of correct positive examples to the number classified as positive, while specificity is the same ratio for negative examples. Precision is the ratio of correct positive examples to the number of actual positive examples, while recall is the ratio of the number of correct positive examples out of those that were classified as positive, which is the same as sensitivity. If you look at the chart again you can see that sensitivity and specificity sum the columns for the denominator, while precision and recall sum the first column and the first row, and so miss out some information about how well the learner does on the negative examples.

Together, either of these pairs of measures gives more information than just the accuracy. If you consider precision and recall, then you can see that they are to some extent inversely related, in that if the number of false positives increases (meaning that the algorithm is using a broader definition of that class), then the number of false negatives often decreases, and vice versa. They can be combined to give a single measure, the F_1 measure, which can be written in terms of precision and recall as:

$$F_1 = 2\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{2.7}$$

FIGURE 2.8 An example of an ROC curve. The diagonal line represents exactly chance, so anything above the line is better than chance, and the further from the line, the better. Of the two curves shown, the one that is further away from the diagonal line would represent a more accurate method.

and in terms of the numbers of false positives, etc. (from which it can be seen that it computes the mean of the false examples) as:

$$F_1 = \frac{\#TP}{\#TP + (\#FN + \#FP)/2}. \tag{2.8}$$

2.2.5 The Receiver Operator Characteristic (ROC) Curve

Since we can use these measures to evaluate a particular classifier, we can also compare classifiers – either the same classifier with different learning parameters, or completely different classifiers. In this case, the Receiver Operator Characteristic curve (almost always known just as the ROC curve) is useful. This is a plot of the percentage of true positives on the y axis against false positives on the x axis; an example is shown in Figure 2.8. A single run of a classifier produces a single point on the ROC plot, and a perfect classifier would be a point at $(0, 1)$ (100% true positives, 0% false positives), while the anti-classifier that got everything wrong would be at (1,0); so the closer to the top-left-hand corner the result of a classifier is, the better the classifier has performed. Any classifier that sits on the diagonal line from (0,0) to (1,1) behaves exactly at the chance level (assuming that the positive and negative classes are equally common) and so presumably a lot of learning effort is wasted since a fair coin would do just as well.

In order to compare classifiers, or choices of parameters settings for the same classifier, you could just compute the point that is furthest from the 'chance' line along the diagonal. However, it is normal to compute the area under the curve (AUC) instead. If you only have one point for each classifier, the curve is the trapezoid that runs from (0,0) up to the point and then from there to (1,1). If there are more points (based on more runs of the classifier, such as trained and/or tested on different datasets), then they are just included in order along the diagonal line.

The key to getting a curve rather than a point on the ROC curve is to use cross-validation. If you use 10-fold cross-validation, then you have 10 classifiers, with 10 different

test sets, and you also have the 'ground truth' labels. The true labels can be used to produce a ranked list of the different cross-validation-trained results, which can be used to specify a curve through the 10 datapoints on the ROC curve that correspond to the results of this classifier. By producing an ROC curve for each classifier it is possible to compare their results.

2.2.6 Unbalanced Datasets

Note that for the accuracy we have implicitly assumed that there are the same number of positive and negative examples in the dataset (which is known as a balanced dataset). However, this is often not true (this can potentially cause problems for the learners as well, as we shall see later in the book). In the case where it is not, we can compute the balanced accuracy as the sum of sensitivity and specificity divided by 2. However, a more correct measure is Matthew's Correlation Coefficient, which is computed as:

$$MCC = \frac{\#TP \times \#TN - \#FP \times \#FN}{\sqrt{(\#TP + \#FP)(\#TP + \#FN)(\#TN + \#FP)(\#TN + \#FN)}} \tag{2.9}$$

If any of the brackets in the denominator are 0, then the whole of the denominator is set to 1. This provides a balanced accuracy computation.

As a final note on these methods of evaluation, if there are more than two classes and it is useful to distinguish the different types of error, then the calculations get a little more complicated, since instead of one set of false positives and one set of false negatives, you have some for each class. In this case, specificity and recall are not the same. However, it is possible to create a set of results, where you use one class as the positives and everything else as the negatives, and repeat this for each of the different classes.

2.2.7 Measurement Precision

There is a different way to evaluate the accuracy of a learning system, which unfortunately also uses the word precision, although with a different meaning. The concept here is to treat the machine learning algorithm as a measurement system. We feed in inputs and look at the outputs that we get. Even before comparing them to the target values, we can measure something about the algorithm: if we feed in a set of similar inputs, then we would expect to get similar outputs for them. This measure of the variability of the algorithm is also known as precision, and it tells us how repeatable the predictions that the algorithm makes are. It might be useful to think of precision as being something like the variance of a probability distribution: it tells you how much spread around the mean to expect.

The point is that just because an algorithm is precise it does not mean that it is accurate – it can be precisely wrong if it always gives the wrong prediction. One measure of how well the algorithm's predictions match reality is known as trueness, and it can be defined as the average distance between the correct output and the prediction. Trueness doesn't usually make much sense for classification problems unless there is some concept of certain classes being similar to each other. Figure 2.9 illustrates the idea of trueness and precision in the traditional way: as a darts game, with four examples with varying trueness and precision for the three darts thrown by a player.

This section has considered the endpoint of machine learning, looking at the outputs, and thinking about what we need to do with the input data in terms of having multiple datasets, etc. In the next section we return to the starting point and consider how we can start analysing a dataset by dealing with probabilities.