Detailed Notes for CSCU9M5, Introduction to Machine Learning

Prof. Kevin Swingler

August 16, 2024

Contents

Chapter 1

Machine Learning and Analytics

1.1 Introduction

This chapter covers a set of methods that have been variously described as **data mining**, **machine learning** and **predictive analytics**. The methods it describes are concerned with the task of using data to teach a computer to perform a chosen task. There are a great many abilities that we might want computers to have that cannot be programmed because the rules are not known. If a computer can learn from examples or by trial and error, then it can discover the rules for itself, thus removing the need for a human to program the rules directly. This is **task oriented** analytics because the goal is not to produce a report that informs a human reader; the goal is to produce a computer program that can perform a chosen task. The main advantage of this is that the task can then be performed automatically by computers without the need for human intervention. This chapter presents methods for allowing computers to learn from data.

My first job after graduating was with the Ford Motor Company, where I worked on a system that monitored driver behaviour and predicted when they might be at risk of falling asleep at the wheel. The rules that are needed to prediict driver alertness from steering wheel movement are not known, so writing a program to implement those rules was not possible. It was possible to collect data, however. Drivers in a dual control car were recorded as they drove around a test track. Their use of the car's controls were recorded as were EEG readings from their head, which give a reliable indication of state of alertness. My job was to discover the relationship between the driver's manipulation of the car's controls and their state of alertness, as indicated by the EEG signal. If a computer could learn that relationship, then the alertness state of any driver could be predicted from the driver's actions without the need for the EEG measurement equipment. This is a good example of a machine learning project.

The driver alertness monitor worked well and it had the curious property of obtaining a skill that no human has. It had learned to predict alertness from steering wheel movement. We do not know what rules it used—they are not represented explicitly in neural networks, which was the method we used. Section 1.6.5 describes neural networks in more detail. Since that early project, I have worked on a great many machine learning projects, teaching computers to spot insurance fraud, pick targets for direct marketing campaigns, spot when cows are at their most fertile, choose the right treatment for patients with renal failure, predict the pattern of side effects from chemotherapy and determine how likely it is that a customer will repay a loan. I sometimes wonder whether I have taught more students or more computers!

All the examples above, while being from a variety of different application areas follow the same pattern. There is a system that has inputs (customer profile, insurance claim details, patient blood test results, etc.) and an output (Did they repay the loan? Was the claim genuine? Was the treatment successful?). By *system* we mean any process that takes inputs and generates an output. A remarkable variety of tasks can be represented in this way.

1.2 Data Structure and Definitions

Data Before embarking on a study of the methods for machine learning, it is important to understand the raw materials involved: the data. Data used for machine learning generally consist of a set of **variables**, each of which has an associated domain from which it can take possible **values**. The variable $MaritalStatus$, for example, might take any value from the set $\{Married, Partnership, Single, Divorced, Widowed\}$ and the variable *height* might take any value from the set of positive real numbers. A collection of data, known as a **data set**, contains a set of values for a number of variables. It is often represented in tabular format in which one row is a single **data point** and is made up of a value for each of the variables in the table. A column of the table corresponds to a single variable.

Data represented in tabular form like this is known as **structured data**. Many data mining techniques require structured data. There are some new machine learning methods that work on **unstructured data** such as images and text, and we will briefly touch on those towards the end of this chapter. On the whole, however, we will discuss structured data.

Data Types Variables can be of different types. Firstly, a distinction is made between **numeric** types, which represent numbers and **nominal** types, which are words. Numeric types have an ordering ($3 < 5$, for example) and a fixed distance metric (the distance between 1 and 3 is 2, the same as the distance between 5 and 7). Nominal types do not have an ordering and the distance between two nominal values is either 0 (they are the same) or 1 (they are different). Numeric types are further subdivided into **discrete** and **continuous** types. Discrete numeric variables can take one of a finite set of values and so have something in common with nominal types. Continuous variables may take any numeric value.

Nominal types are sometimes also called **categorical** types. Another variation on the nominal type is the **ordinal** type, which takes values from a set of names that have an order, for example {*Very much*, *quite a lot*, *a little*, *not at all*}. These have more in common with discrete numeric variables. Finally, nominal types might be **binary** (sometimes called **dichotomous**), which means they may take one of only two possible values (Yes or No for example).

Sometimes, the type of a variable is determined by how you treat it, rather than by its inherent qualities. When considering the type of a variable, or the type you will treat a variable as having, it is important to understand the consequences of that choice on the machine learning process. That is more important than a simple judgement of the variable in question, and will be referred to several times through out this part of the book.

Take, for example, the variable $NumberOfChildren$, which represents the number of children a person has. Perhaps this data is being used to help target marketing messages. It looks like a number—it even has *Number* in its name, but it does not have the qualities of a number as we have defined them. First of all, it is discrete. You cannot have 2.5 children. Secondly, there is a sense in which the distances between points along its scale are not equal. The difference between having zero children and having one is far greater than the difference between having one or two (believe me). The number of children somebody has is really a class to which they belong. You will see in later sections that the way a variable is represented in a machine learning model is determined by its type and the choice can have consequences for the complexity and accuracy of a model. Choosing a data type for a variable is a lot more than a simple academic naming exercise.

Each variable will belong to either the set of **inputs** or (in the case of supervised learning) the set of **outputs**. Machine learning is often concerned with the task of discovering a function that maps the input variables to an output. These functions may be represented as mathematical formulae or sets of rules or graphs of nodes and edges, but they all have the same task: to map inputs to outputs. In statistics, input variables are often called **independent variables** and outputs are known as **dependent variables**.

1.2.1 Notation

This chapter will rely more heavily on mathematical notation than some others in the book, but if that is a little outside of your comfort zone, do not be put off. The equations are easy to read with a little practise. Many of my students progress quickly from a state of equation anxiety to being quite happy with them, and you can too. Here is the notation this chapter employs:

Variables and vectors are represented as upper case letters. Inputs to a function are denoted X and outputs are Y. In the case of a vector, individual elements within the vector are given a subscript to show their position, so if X is a vector of three variables, then they are denoted X_1, X_2, X_3. Indexes start at 1.

Variables can take one of a set of values and there are times when we want to refer to the value at a current observation, for example $X = 1$. More commonly, we want to refer to a value without being specific, and in those situations, lower case letters are used. So x_i refers to an observed value of the variable X_i. This is useful when describing the jth member of a data set as the vector x^j. The ith element of the jth member of a data set is denoted x_i^j. Sets (in particular, data sets) are represented in bold upper case. The data set $\mathbf{D}$, for example is made up of data points: sometimes single points, $\mathbf{D} = \{x^1, \dots x^n\}$ and sometimes in pairs, $\mathbf{D} = \{(x^1, y^1), \dots (x^n, y^n)\}$. I will often refer to observed values across a vector as a **pattern**. For example, the variables $Age, Height, Weight$ might take the pattern 35, 176, 75 in a single observation. It is also common to describe a single pattern as a **data point**.

The **domain** of a variable defines the values it can take. We use a decorated font (known as blackboard bold) to denote domains. The domains of variables X and Y are denoted $\mathbb{X}$ and $\mathbb{Y}$ in general but we can be more explicit using $\mathbb{R}$ to denote the set of real numbers, $\mathbb{Z}$ to denote integers and $\mathbb{B}$ to represent the binary domain. You have already seen set notation used to represent the domain of nominal variables, for example the binary output variable, Y might have a domain of $\mathbb{Y} = \{Yes, No\}$. See equation 1.38 for an example of iterating over the variables in the domains of X and Y.

We, and many other textbooks, will talk about **random variables**. You will see a statement like X *is a random variable*. A random variable is one that we can observe taking different values, but that we cannot control directly. If we watch cars drive past and note their colour, then *carcolour* is a random variable. Note that *random* does not mean entirely unpredictable, nor does it necessarily mean uniformly random. Some values can be far more likely than others.

1.2.2 Machine Learning Tasks

Approaches to machine learning can also be classified as being either **supervised** or **unsupervised**. Supervised learning involves data that describe both the inputs and the outputs to the system and requires a mapping to be learned from the inputs to the outputs. Unsupervised learning involves only the inputs and requires the algorithm to organise or characterise the data in some way. Machine learning activities can be further classified by the type of task they attempt to perform. The following sections introduce some of the main types of task that machine learning is used to perform.

Classification

Classification is a supervised learning task where an input pattern is classified as belonging to one of a number of possible classes. The output variable is nominal and the inputs can be a mix of numeric or nominal. Examples of classification tasks include spotting a fraudulent insurance claim, recognising a face from a CCTV image, diagnosing an illness from its symptoms and (my old favourite) spotting when a driver is falling asleep at the wheel.

Prediction

Prediction is a supervised learning task where a continuous output value is calculated from an input pattern. The learning task is to find the relationship between the input variables and one or more output variables. An example is predicting the next day's circulation figures for a newspaper based on the price, recent advertising spend, the type of story on the front cover and the price of the main competitor. Another example is predicting the revenue a new store might generate based on its size, the availability of parking, the demographics of the local population and its location. The inputs can be a mixture of numeric and nominal values, but the output in a prediction task is numeric.

You will also see the term prediction used to refer to classifications in the context of the output from a classifier model. This captures the sense that the output from a classifier has a level of uncertainty associated with it: it is an educated guess. This makes more sense in some applications than others. If we are classifying loan applicants with respect to the risk of them defaulting on payments, then there is a real sense that the output is a prediction of future behaviour. Other examples such as face recognition involve putting a label on an existing fact and are not predictions in the common sense of the word at all. If this is all a bit confused, just remember that predictions output numbers and classifications output names. The act of performing a numeric prediction is also known as **regression**, as you will see later on.

Clustering

Clustering is an unsupervised learning task in which data points that are close to each other (by some distance metric) are assigned to one of a number of clusters so that members of different clusters are far apart. The input variables can be numeric (in which case a common metric is Euclidean distance) or nominal (where distance is either one or zero). There is no explicit output variable in the training data, but an output variable is created to name the clusters. An example of clustering is organising the customers of a company into groups that share characteristics so that they may receive more relevant marketing materials. This is sometimes called customer profiling and the groups are sometimes known as tribes. Other examples involve clustering of geographical locations in applications as diverse as helping a fast food chain locate new outlets or working out where earthquakes are likely to hit. Clustering is a little like classification except that the class labels are not given by the training data. They are inferred from the distribution of points in the input data.

Novelty Detection

Novelty detection is an unsupervised learning task that requires the system to spot patterns of data that have not been seen before. There is no output variable in the training data but the resulting system will have a binary output that classifies each input pattern as novel or not. Novelty detection is used in cases where only one class of data can be collected but it is important to know when the process generating that data has changed. A common example is machine health monitoring, in which data from a functioning machine is easy to collect, but data from a machine that is about the break down is impossible or very expensive to collect. In such cases, it is enough to raise an alarm when the readings from the machine are abnormal. Novelty detection is also used in conjunction with other machine learning tasks to look for data that indicates that the real world system generating the data has changed and that the existing algorithms may need to be updated. It can act as a early warning system in a great many situations. Novelty detection has features in common with clustering. It attempts to characterise a data set in such a way that allows a judgement to be made about new data.

Probability Distribution Estimation

Clustering methods attempt to partition a data set into discrete subsets so that a new data point may be allocated to one of them. Novelty detection attempts to recognise whether a new data point is similar to the training data. Both answer versions of a more general question about the relative density of the training data at different parts of the input space. Building a model that takes a single data point as input and produces an estimate of the density of the population data at that point is known as **Probability Distribution Estimation**. It involves building a model from the data in the form of a function from the inputs, X to a probability estimate, $p(X)$. Note that the process is unsupervised as the probabilities are not known and must be inferred from the data.

There are two types of input variable, as we have already discussed. Those that are continuous and those that are discrete. The difference is very important when modelling probabilities. Discrete variables are relatively simple to model, you simply count the number of times each possible value occurs and divide each count by the total. That way, each probability is between zero and one and they all add up to one. The set of discrete probabilities over the values a variable can take are known as its **probability mass function** (PMF).

Things are not as simple for continuous variables. If a variable can take any one of an infinite number of values, then the probability of any particular value being observed is zero (pretty much) so we do not talk about the probability at single points, but about the **density** of values across the range a variable covers. A function known as the **probability density function** (PDF) is used to calculate the density at a given point, but this number is not a probability. You can only use a PDF to calculate the probability of a value falling within a chosen range. For those of you who remember your calculus, this is done by integrating the PDF over the chosen range.

Probability functions (both PMFs and PDFs) are used to estimate the probability of a variable (or set of variable) taking a given value (or values). PDFs are smooth functions, reflecting the continuous nature of the idea of a density. Think about the normal (bell curve) distribution. It is highest at the mean of the variable, so this is where the values are most dense. In one sense, density measures how many values in a dataset are near a given point.

A probability or density function can be used for novelty detection because observations that are given a low density by the model are in some sense novel, or rare. If the density function is parametrised in the right way, it can also act as a soft clustering method, assigning points to each of the clusters with a certain probability. This is discussed in more detail in section 1.8.

1.3 Some Theory

This section discusses some important theoretical foundations on which many of the machine learning algorithms and much machine learning practice are based. It is important to understand these concepts as they have real consequences for practical data mining projects.

1.3.1 Statistical Models

A statistical model attempts to represent some real world system in mathematical form so that some aspects of the model are shaped by observed data. Statistical models capture the stochastic nature of the real world process they model, making use of distributions over the values each variable may take. They are generally quite stylised representations of a real world process that generates data. When statisticians talk about data generating processes, they mean anything that generates data that might be measured, so it could be a machine whose temperature and oil pressure constantly change, or it could be the heights and ages of the population of school teachers. The data do not fully define the model, however. It is almost always the case that assumptions are required that constrain the model in some way and that the data shapes the model within the constraints of those assumptions.

In all but very rare cases, the data used to build the model do not represent the full **population**, that is every single data point that exists for these particular variables. Rather, it represents a **sample** of a smaller number of data points from which qualities of the population must be inferred.

Here is a simple example. Imagine you supply a chain of sandwich shops and you need to know how many sandwiches to make each day. The total sold each day represents a random variable, X. Having collected data from your customers for a while, you have a dataset in which observations of X have been made each day. As X is a random variable, it has a probability distribution associated with each of the possible values it might take. If you knew what this distribution was, you could answer questions such as "*What is the most probable number of sandwiches I will sell?*" and "*What is the probability of me selling more than 1000 sandwiches tomorrow?*".

Armed with the data describing previous daily sales, you can attempt to build your distribution model. First of all, you need to make an assumption about the shape of the distribution. For the sake of simplicity, let us assume that X is normally distributed in this case. That means the distribution can be modelled completely using two parameters: the mean and the variance. Those parameters can be calculated easily from the data, plugged into the equation for the normal distribution, and the model is complete. You could now answer the questions we posed above. This neatly illustrates the two parts of statistical model building: the assumptions made by a human designer and the parameter estimates gained from the data. The values we calculated for the mean and variance are estimates of the true (but unknown) population mean and variance. A different sample would probably have given different (but similar) estimates and a lot of statistical theory is dedicated to dealing with that fact.

Perhaps the assumption that the data are normally distributed is good, but what if it is not? This can be tested and if the assumption proves poor, another model should be built. The normal distribution is simple—it only requires two parameters to describe it, so perhaps a more complex model would be better. We could split the sale figures into bands and count how often sales fell into each band. Perhaps on 55 days we sold 1000-1100 sandwiches, on 50 days we sold 1101-1200, and so on. Building a histogram from this data (or a look up table that gives the number of times sales fell into each band) provides a more detailed model. The assumption we have made is less restrictive—just that the size of the bands is correct. The more bands we use, the more closely we can model the sampled data but the more data we will need to get meaningful counts for each band (I hope you can see why). What might be less obvious, but is just as true, is that the more bands we use, the more specifically we will model the sample data, increasing the risk of actually making our model less true to the population and the real world we are trying to model. If it really is true that sandwich sales are normally distributed, then the right model can be built from a much smaller sample and can actually be more accurate than a model that assumes more complexity than there really is. Figure 1.1 illustrates this point, which is sometimes known as **Occam's Razor** after William of Occam, who stated that among competing useful hypotheses, the one with the fewest assumptions should be selected. This is also often referred to as a preference for **parsimony**.

The process of making assumptions and estimating parameters using a sample of data is common to almost every approach to machine learning. The trade-off between the level of complexity of the model and its ability to capture the detail of the sample and the population is also ubiquitous. The next section addresses these ideas in more formal detail.

The Bias-Variance Tradeoff

In the previous section, we introduced the role that assumptions make in the process of statistical modelling. They can be used to impose simplifying constraints on a model in an attempt to mitigate the risks associated with estimating population parameters from a sample of data. As usual, we need to name these ideas before we can use them. In machine learning, the data used to train a model (i.e. to estimate its parameters) is called the **training data** and the data used to test how well that model does on data that was not involved in the training process is known as the **test data**. We will meet a third set, the **validation data** later on. The process of estimating the parameters is known as

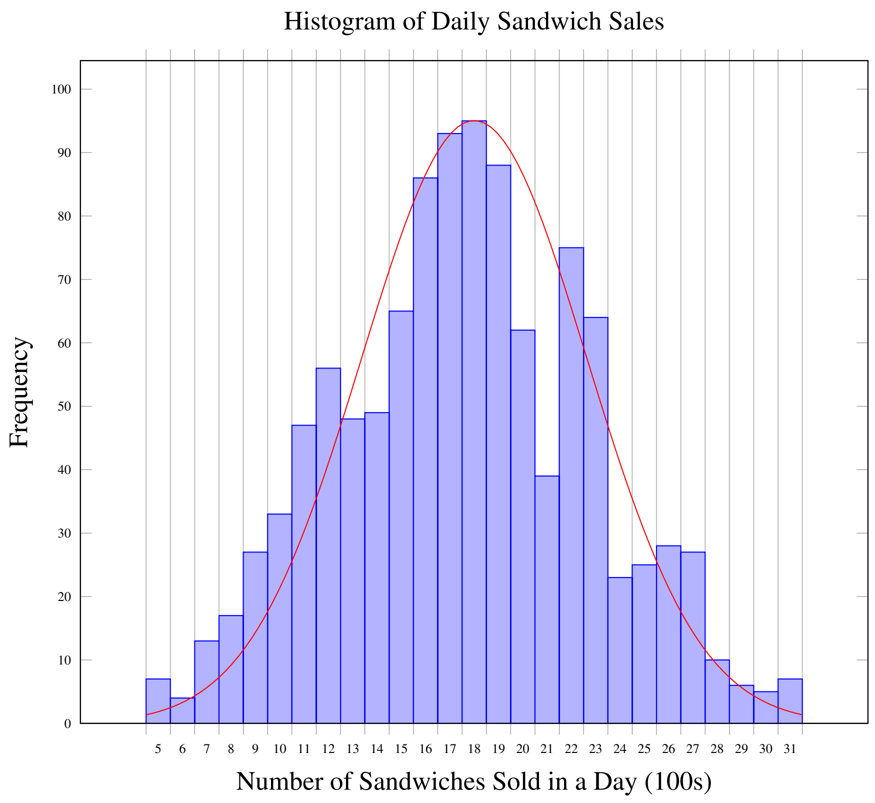


Figure 1.1: Two alternative models of sandwich sales by frequency. The red line assumes a normal distribution with a mean of 1800 sandwiches sold and a standard deviation of 600. The blue histogram is a more complex model that splits the data into many bins, meaning the counts vary greatly from one band to the next. The overall shape is similar to the normal model but the histogram look up table is unlikely to generalise as well to new data.

model fitting and a model that does a poor job of representing the data is said to be **under fit**. Similarly, a model that is more complex than is required and represents the training data so well that it actually represents a poor model of the real system that generated the data is said to be **over fit**. Clearly, both of these situations should be avoided.

Simplifying a model by imposing constraints on its complexity is known as **regularisation**. It may be performed because the designer has good reason to suspect a given model structure or level of complexity is right, for example when a normal distribution was assumed for the sandwich example above. Alternatively, it may be performed as part of a process designed to reduce over fitting. Any deviation from the model that best fits the training data is called **bias**. Most assumptions about model structure introduce **model bias**, which limits the ability of the model to represent the full structure of the training data regardless of how the parameters are set. You can also introduce **estimation bias** when learning the parameter values, further simplifying a model by preventing it from learning the parameters as well as it might. As Mark Twain put it, "*We should be careful to get out of an experience only the wisdom that is in it, and stop there …*".

Section 1.3.1 mentions that data used for building statistical models is a sample from the larger population and that a different sample may lead to different estimates for model parameters. The simpler the model, the less room there is for these differences between models. Recall the example in section 1.3.1 where the distribution of sandwiches sold each day is being modelled. A simple model with bias in the form of an assumption of normality will result in similar mean values across a range of different samples. That is to say that there will be little **variance** among the estimates of the mean across many different samples. The different samples do not exist, of course—there is only the one data set—but if they did, if the experiment could somehow be repeated many times, the resulting means would have low variance. If those same repeated experiments, generating many alternative samples, were each modelled using a much more complex model, then the differences among those models would be greater. Look at the histogram in figure 1.1 again and see how some bars are much lower than the pattern suggests. In another sample, those bars might be as high as expected and others may be unexpectedly low. The means and variances in both cases would be very similar, meaning the red line would not change much, so the complex models display more variance than the simple ones.

The examples above discuss what is known as the **bias-variance tradeoff**. It tends to be the case that models with a lot of bias have low variance and models with little bias have more variance. High variance is associated with overfitting and high bias can be associated with underfitting of the training data.

In reality, you only have the one data set, so you cannot measure the variance. You can control the bias, however and you can use data that was not used for training to see how well different levels of bias compare. You cannot use the test data for this—its job is purely to allow you to report how well your chosen model performs on data it has not seen. The data to use for exploring different levels of bias is the **validation set**. Sections 1.5.1 and 1.10.1 describe more advanced methods for validation, but for now you should understand that the validation data is used to allow you to explore different levels of bias with the same training data. The right level of bias is that which generates a model that performs best on the validation data.

In practical terms, you have one data set and only the validation methods to guide your search for the correct model. The goal of the machine learning process is to use the available data to generate a model that is able to **generalise** to new data. That means that its accuracy does not diminish when it is deployed and used with real data.

Learning and Cost Functions

So far we have talked about estimating parameters, but not discussed how this is done. This section addresses that question. Take the sandwich shop example again. We want to estimate the mean of the distribution, μ. You probably know how to calculate the mean: you sum the values and divide by the number of values there are (n): $\mu = \sum_i x_i/n$. How is that formula arrived at, though? In fact, it is the result of attempting to find a value for μ that has a particular property: it minimises the sum of the squared differences between μ and each of the values in the data set. We will prove that it does so in a little while, but let us first ask why it is done.

The principle behind many parameter estimation (or learning) algorithms is known as **cost function minimisation**. A **cost function**, C, defines the quality of an estimate based on the training data. In the case of the mean, the cost function is the sum of squared differences, i.e. $C = \sum_i (x_i - \mu)^2$. The cost function takes a set of data and an estimate of a parameter value and returns a value that reflects the quality of the estimate, given the data. If the data are $\{1, 2, 3\}$, then a choice of $\mu = 1$ will result in a cost of $(1-1)^2 + (2-1)^2 + (3-1)^2 = 0 + 1 + 4 = 5$. Choosing $\mu = 2$ results in a cost of $1 + 0 + 1 = 2$, so according to this cost function, $\mu = 2$ is a better estimate than $\mu = 1$. The process of finding the parameter value that minimises the output from the cost function is known as cost function minimisation. This can sometimes be done analytically, as you will see below, and otherwise needs to be done using a search procedure such as gradient descent (see section 1.3.1).

There are many different cost functions and we will introduce them as they are needed, so do not get the idea that cost function means squared difference or that cost functions are for calculating the mean only. They are used widely in many different machine learning methods. If you are curious and happy to follow the maths, here is the derivation of the formula for calculating a mean based on the sum of squared differences cost function. If you are not happy to follow the maths, feel free to skip the next bit.

First, let us state the problem. Given a set of n data points, $\{x_1 \dots x_n\}$, a parameter μ, and a cost function, C, which is the sum of the squared differences between μ and each x_i, find value of μ that minimises C. First, we write C:

$$C = \sum_{i=1}^{n} (x_i - \mu)^2 \tag{1.1}$$

We know that C is a convex function (so it only has one minimum). The minimum is the point where the derivative of C with respect to μ is zero:

$$\frac{dc}{d\mu} = 2\sum_{i=1}^{n} (x_i - \mu) = 0 \tag{1.2}$$

Solving that gives you the point where C is minimised by μ.

$$\begin{aligned}\sum_{i=1}^{n} x_i - \sum_{i=1}^{n} \mu = 0 \\ \sum_{i=1}^{n} x_i - n\mu = 0 \\ \sum_{i=1}^{n} x_i = n\mu\end{aligned} \tag{1.3}$$

which finally gives us the familiar formula for calculating a mean.

$$\mu = \frac{\sum_{i=1}^{n} x_i}{n} \tag{1.4}$$

Log-Likelihood Cost Functions For classification models and other distribution estimation techniques, squared error is not a good cost function as you cannot compare the probability value output by a model to the true probability, which is not known. Given a sample of observations of a random variable X, $\{x^1 \dots x^n\}$, the task is to build a parametrised function, $P(X)$ that outputs the probability (mass or density) associated with an observed value. The data set does not contain values for $P(X)$, only the samples, x. The question of how good the parameter values are can be framed as the question of what is the probability of the parameters taking those values given the current data. In statistics, the probability of something observed (like the parameters) being explained by a possible cause (the data) is known as its **likelihood**.

A likelihood function is the product of the probability estimates given by a model in response to a set of data (that is to say, we multiply all the probability estimates from a dataset together). The better the probability estimates match the distribution of the labels in the data, the higher the likelihood becomes. The likelihood of a set of parameter values in a model given a set of data points is calculated by evaluating the probability of each data point according to the model, and multiplying those probabilities together, giving the equation

$$\mathcal{L} = \prod_{j=1}^{n} \hat{P}(x^j) \tag{1.5}$$

where $\hat{P}(x^j)$ is the estimate from the model of the probability associated with sample x^j from the data. As the product of many fractional numbers soon becomes very small, it is more usual to sum the logs of the probabilities, calculating the **log likelihood**, which is

$$\ell = \sum_{j=1}^{n} \ln \hat{P}(x^j) \tag{1.6}$$

Consider the simple example of calculating the single parameter associated with a model of an unfair coin toss. The parameter to be estimated is $P(H)$, which estimates the probability of seeing a head. Obviously, the probability of seeing a tail is $1 - P(H)$. Given a data set with 75 heads and 25 tails, you might think it is obvious that $P(H) = 0.75$ but there are many cases where that simple calculation cannot be made and a cost function is needed. What's more, you arrived at 0.75 by dividing the number of heads by the total number of coin flips, which is the maximum likelihood estimate, as shown below. If the initial estimate for the parameter is 0.5 (a fair coin) then the log likelihood is $75 \times \ln 0.5 + 25 \times \ln 0.5 = -69.3$. A better choice might be 0.6, in which case the likelihood is $75 \times \ln 0.6 + 25 \times \ln 0.4 = -51.1$, which is an improvement. You can verify for yourself with a calculator or a spreadsheet that $P(H) = 0.75$ maximises the likelihood with $75 \times \ln 0.75 + 25 \times \ln 0.25 = -28.8$. Figure 1.2 shows a plot of $P(H)$ against log likelihood for a coin that is biased so that it falls heads 75% of the time, and highlights its maximum at $P(H) = 0.75$

Just as we showed how the calculation for the mean is derived analytically from a cost function, we can also show how the maximum likelihood estimate of a probability is calculated as the frequency of the event divided by the number of events in total. As before, you can skip this example if you are not comfortable with the maths. In this

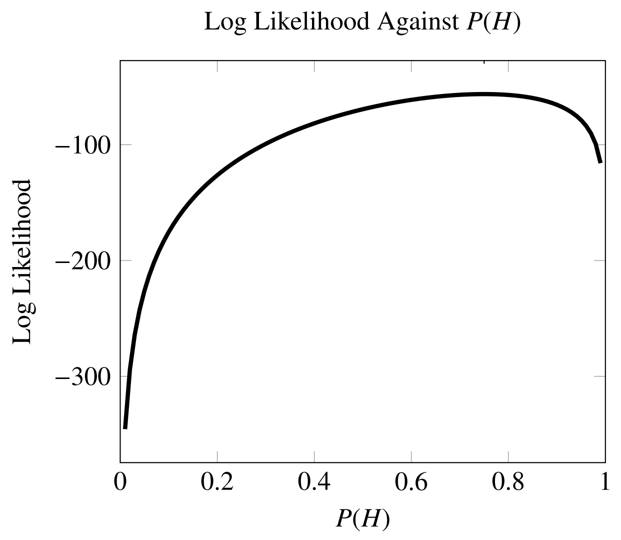


Figure 1.2: The log likelihood associated with different estimates of the probability of a biased coin coming up heads when the true bias means that probability is $P(H) = 0.75$. Note that the maximum point is where $P(H) = 0.75$.

example, we will use the biased coin from the discussion above. The model can be specified by a single parameter, θ, which is the probability of flipping a head, so the model is defined as $P(X = H) = \theta$ and $P(X = T) = 1 - \theta$. As before, we state the problem: Given a set of n observations, $\{x_1 \dots x_n\}$ with $x_i \in \{H, T\}$, find $\theta = P(X = H)$ by maximising the log likelihood function, ℓ. First, we define ℓ, which is the sum of the logs of the probability estimates made for each observed coin flip (i.e each x_i). If there are h heads and t tails observed then that sum is h times the log of the current model's estimate of the probability of flipping a head plus t times its estimate of the probability of flipping a tail:

$$\sum h \ln(\theta) + t \ln(1 - \theta) \tag{1.7}$$

Next, we differentiate this using the fact that the derivative of $\ln(x)$ is $1/x$ and set the derivative to equal zero

$$\frac{dl}{d\theta} = \frac{h}{\theta} + \frac{t}{\theta - 1} = 0 \tag{1.8}$$

Multiply the numerator of each term by the denominator of the other to produce a common denominator:

$$\frac{h(\theta - 1) + t\theta}{\theta(\theta - 1)} = 0 \tag{1.9}$$

We can now ignore the denominator because making the numerator equal zero will make the whole fraction equal zero:

$$h(\theta - 1) + t\theta = 0 \tag{1.10}$$
$$h + t = \frac{h}{\theta} \tag{1.11}$$

so

$$\theta = \frac{h}{h + t} \tag{1.12}$$

which is the proportion of all observations that came up heads, and the maximium likelihood estimate for the probability of flipping a head, derived from those data.

Gradient Descent

In many cases it is not possible to find the parameter value (or values where a model has many parameters) analytically (by solving the equation where the derivative is zero) and a search is required. This might be because a solution that produces a low cost is preferred over one that completely minimises the cost with reference to the training data. This, as described in section 1.3.1, is known as introducing **estimation bias**. Calculating the parameter value that locates the global minimum in the cost function produces an **unbiased estimate** of the parameter value, so equation 1.4 represents an unbiased estimate of the mean. Biased estimates can be used to attempt to avoid over fitting.

Alternatively the cost function may not be in a form that allows the derivative to be set to equal zero and solved. Non-convex functions (those with more than one local minimum) cannot be solved in this way, for example. In such cases, the minimum must be searched for. Searching involves trying a parameter value, evaluating the choice with the cost function, and then making a decision about how to change the parameter value to reduce the cost. A common method is known as **gradient descent** and involves making small changes to the parameter being estimated so that each change reduces the cost function. The direction of the move is dictated by the derivative of the cost function at the point of the current estimate. A negative derivative indicates that increasing the parameter estimate slightly would reduce the cost slightly, for example.

One problem with gradient descent occurs when it is applied to non-convex functions. A function that is not convex has multiple points where a small change in the input in any direction results in the cost increasing. These are known as local minima because there are other turning points elsewhere in the function where the cost is lower, but local iterative searches like gradient descent cannot reach them from the local minimum as that would require a (sometimes long) journey uphill. There are many variations on the basic gradient descent algorithm and many methods for dealing with local minima. Some of these will be addressed as part of the treatment of specific algorithms later in the chapter.

Most statistical models used in modern machine learning tasks have many parameters, which makes the cost minimisation process more complicated. The size of the space of all possible combinations of parameter values grows exponentially with the number of parameters so the model building process can only explore a tiny subset of possible models.

The basic approach to fitting parameter values from a data set using gradient descent involves picking a set of parameter values and calculating the cost associated with them. The cost function is defined over the entire data set so each parameter change involves calculating the total cost over the whole sample. This approach is known as **batch** gradient descent and can be quite inefficient with a large data set. An alternative is to approximate the change in parameter values that the full cost function would produce by changing the parameter values based on each data point alone, one at a time. This is known as **stochastic** gradient descent, as the path the search takes depends on the order in which the data are seen. Stochastic gradient descent makes many more steps (one per data point rather than one per pass of the full data set) so it can be faster than a batch approach, but each change is based on only a single example, so carries a lot less information and leads to a less smooth descent. A compromise between these two approaches is known as **mini batch** gradient descent and involves updating each parameter based on the average cost of a small subset of data points. This attempts to smooth the descent without the inefficiency of a full batch approach.

Algorithm 1 describes the basic principle of gradient descent applied to fitting parameter values by attempting to minimise a cost function with reference to a data set and figure 1.3 summarises the discussion about cost functions and gradient descent.

Algorithm 1 Parameter Estimation by Basic Gradient Descent of a Cost Function

```
Let $\mathbf{D}$ be the training data set
Let $\beta = \beta_1 \dots \beta_p$ represent a vector of parameters to be estimated
Initialise the parameter values to an initial state
repeat
    Select $(\mathbf{X}, \mathbf{Y}) \subset \mathbf{D}$
    Calculate the derivative $\frac{\partial C(\mathbf{X},\mathbf{Y},\beta)}{\partial \beta}$
    Update $\beta$ in the direction of the cost derivative
until Stopping Criteria are met
```

A final note on terminology: we have used **cost** and **cost function** throughout, but you will also see the phrases **loss** and **loss function**. Strictly, a loss function measures a single data point and a cost function is an average of losses over a batch or dataset.

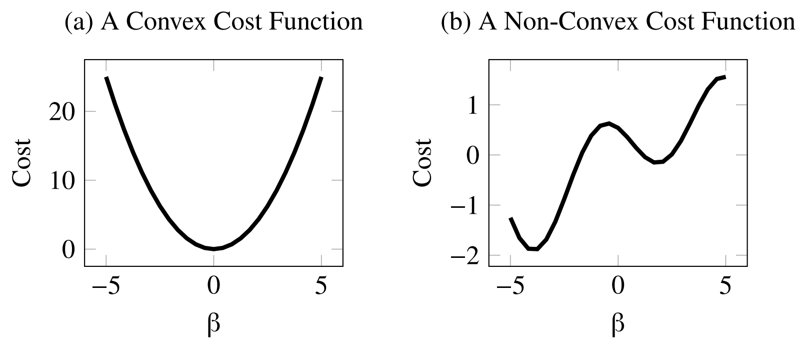


Figure 1.3: A convex cost function (a) and a non-convex cost function (b). An algorithm performing gradient descent would find the global minumum of function (a) but would fall into the local minimum in (b) if starting at $\beta > -0.5$.

1.4 Running a Machine Learning Project

Data driven projects are very different from traditional software engineering projects. The goal in both cases is the same: to develop a system where a computer program has a defined functionality. The methodologies used are very different however. In fact, there are a number of technical, commercial and cultural differences between data driven projects and standard software engineering or IT projects. These will be discussed further in section 1.16. This section concentrates on the data mining (or machine learning) methodologies. Most data mining projects have the same set of components to be considered:

- The process that generates the data and the data read from it
- The context in which the data are generated and used
- A goal or requirement from the system being developed
- The tools, technologies and methodologies of the machine learning process
- The deployment of the resulting predictive system

1.4.1 The CRISP-DM Process

The cross industry standard process for data mining (CRISP-DM) [**?**] was developed in 2000 and is still a good approach today. It defines a cycle of stages that may require more than one iteration before a project is successfully completed. Figure 1.4 illustrates the stages, which are described next.

Business Understanding

The business (and the term is used in the broadest sense, much as it is when we talk about business rules when modelling a database, to mean any organisation) that generates the data has systems, rules, goals, people and a relationship with the rest of the world. The data used in a data mining project are a product of these things and must be understood in their context. The process of understanding the business can be formal or informal. It involves, among other things, deciding on the task to be performed by the machine learning solution. That requires an understanding of what is possible. Section 1.2.2 discusses the main types of machine learning task in broad terms: prediction, classification, clustering and novelty detection. Matching business goals to these tasks is an important exercise at this stage in the project. The job of the data scientist may be to help others in the company choose the right tasks, which means helping them understand what might be done. I have found a useful process during early sessions is the ask clients to state sentences beginning with "*It would be great if we could…*" without regard for whether they think it might be possible or not.

Whether or not a task is likely to be successfully carried out using machine learning depends partly on the art of the possible and partly on access to the right data. As computers grow continually faster, machine learning more powerful and data more plentiful, the list of things that computers can learn to do is growing all the time. Without the

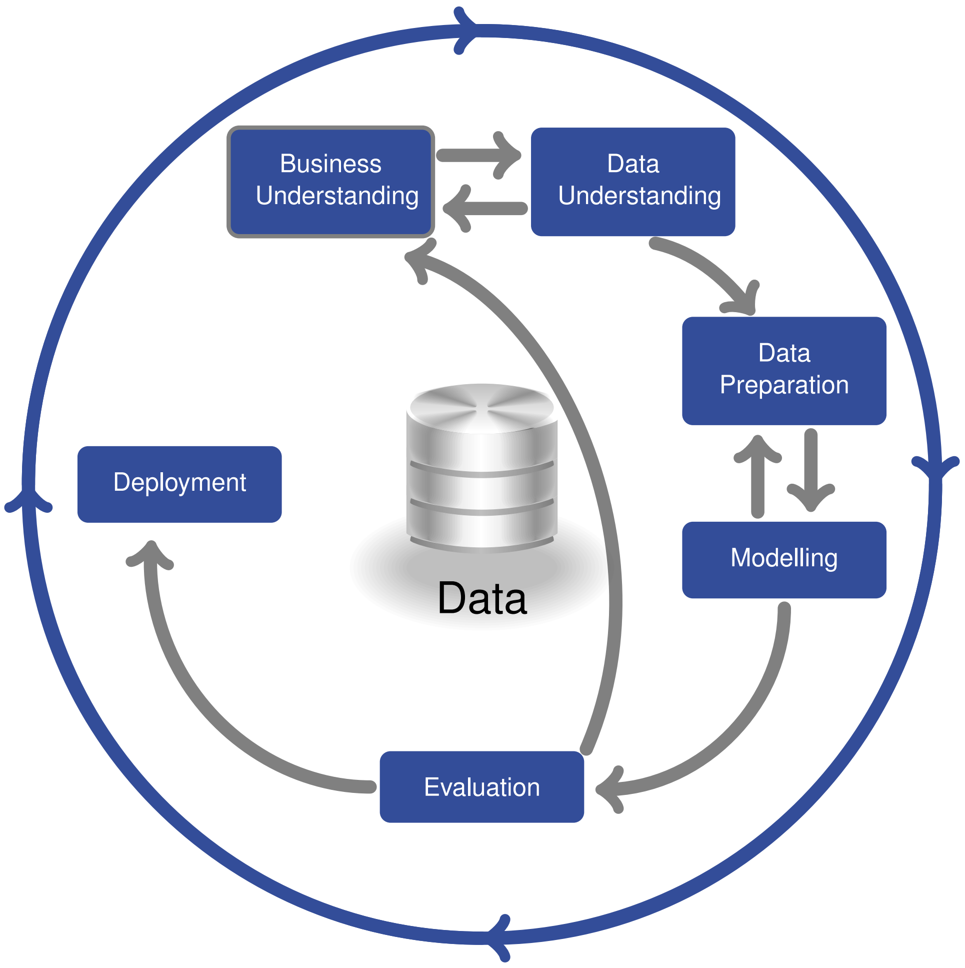


Figure 1.4: The CRISP-DM process for running a data mining project.

right data though, the most advanced machine learning methods can do nothing, which brings us to the second phase, data understanding.

Data Understanding

The main questions to address in the data understanding phase are "*What data do we have?*", "*What data can we collect?*", and "*What might the data allow us to do?*". This is an extension of the business understanding phase as it attempts to match available data to desired tasks. Other considerations concerning the data are about its cost and the timing of its availability. If a machine learning system is to be deployed to make real time predictions, it needs access to the right data in real time too. That almost always means automated access to digital data. Increasingly, that is not a problem, but it is still something that needs to be established.

The outcome of the first two phases should be a decision about what task the machine learning will attempt to perform and what data will be used to teach it. The next three phases are all part of the machine learning methodology designed to ensure that the machine learns to carry out the intended task. This involves preparing the data, choosing the right methods and techniques, finding the right level of bias and attempting to build a model that will be able to accurately generalise to new data.

Data Preparation

Data are rarely clean, correct and in the exact format required for machine learning. Data preparation is the process of manipulating the data into a shape that will give the machine learning algorithms the best chance of success. This involves **data cleaning**, where missing values, data entry errors and other problem data are removed or fixed. Data cleansing can be done automatically or manually with the help of data visualisations. The most useful visualisation at this stage is the histogram (or frequency bar chart, see section 1.8 for a full discussion). The histogram of a single variable (input or output) can alert you to a number of potential problems that will have a negative impact on the quality of the model it contributes to building.

The two main concerns at this stage are **data quality** and **data quantity**. Both concerns are related, as the quantity of data required to perform a task depends to some extent on the quality of the data. In the era of big data, the amount of data available for a machine learning project is not always an issue, but there are still many cases where data sets are small or (in the case of unbalanced data) examples of classes of interest are rare. A number of data quality issues are discussed below, after which we will return to the question of data quantity.

Data Transformations

There are a number of simple transformations that are commonly applied to variables before machine learning takes place. Some are applied automatically by machine learning software, but that varies from package to package so you should understand what transformations should be performed and when. It is also important to note that the test data should be separated off **before** any data transformations take place. You should carefully record everything you do to the data before training a model because you will need to do the same things to the test data and to all future data that are processed by the model.

Many techniques do not work well if the numeric variables being used vary over different ranges from each other. Methods based on distances between data points, such as k-nearest neighbours (section 1.6.2) and k-means (section 1.7.1), are more effective if the variables all share the same range and optimisation methods such as gradient descent also perform better under such circumstances. Two common ways to transform numeric variables to a standard range are **rescaling** and **standardisation**. You will also see the term **normalisation** used to refer to either of these transformations, so check the definitions wherever you meet that word. Rescaling chooses a target range (usually [0,1] or [-1,1]) and rescales a data set to fall into that range:

$$x_i \leftarrow (x_i - min(X))/(max(X) - min(X)) \tag{1.13}$$

and has the advantage of fixing an exact range. If new data falls outside the range used for scaling, however, then the scaled range will exceed its defined limits, which may not be desirable, depending on the circumstances. Standardisation transforms the data so that it has a mean of zero and a standard deviation of one. This allows different variables to be compared in terms of the distance of their values from the mean in standard units. The range of the scaled data depends on the range of the original data, but with a mean of zero and a standard deviation of 1, it can be expected to lie mostly in the range [-4,4]. Standardisation assumes that the data are normally distributed and is performed by rescaling like this:

$$x_i \leftarrow (x_i - mean(X))/sd(X) \tag{1.14}$$

Numeric data may also be transformed into discrete values if required. This might be because there are a small number of possible values for a variable and each has a different meaning in terms of the output, making the relationship non-linear. For example, a variable that reflects the number of children a customer has could be coded as $\{0, 1, 2, 3, more\}$ because having no children is very different from having one and we want the model to reflect that. Continuous variables may be split into subranges, called **bins** for similar reasons.

One-of-*k* Encoding Some machine learning techniques (particularly those that perform regression) require the variables to be numeric so nominal values and discrete numeric values must be recoded into numeric form. It makes no sense to arbitrarily label each nominal value with a numeric value (you cannot recode marital status to $1 = married, 2 = single, 3 = divorced$ etc.) as the distances between values make no sense. The recoding is done by creating **dummy variables**, which take values in {0,1}. One dummy variable is created for each possible value the variable can take and a 1-of-*k* encoding used. That means that all but one of the dummy variables has the value zero and one of them (the one that represents the value to be expressed) has a value of one. The exception to the 1-of-*k* rule is that binary variables are encoded as a single output that takes a value of zero to indicate one class and one to represent the other. When the variable in question is an output, many techniques represent the probability of the

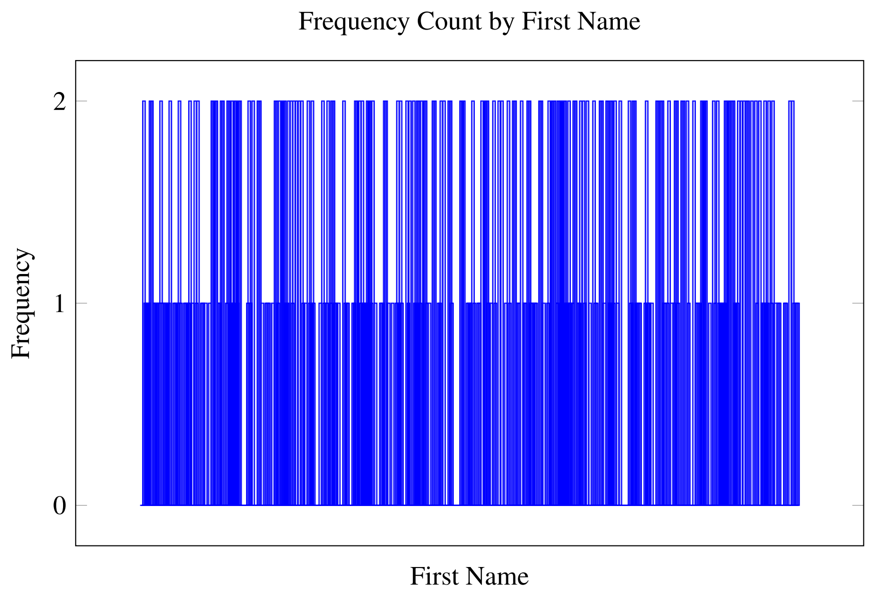


Figure 1.5: The frequency bar chart of a discrete variable with many different values. Such variables are generally unsuitable for use in machine learning unless the values are grouped to reduce their number. Large numbers of values can lead to overly complex models and overfitting.

input pattern belonging to each class as a value between zero and one in each dummy variable. Not all techniques require this transformation—decision trees are a good example where nominal data can be left as they are and discrete variables must be tagged as discrete but do not need to be recoded.

Data Quality

Variables that are either discrete or nominal should have a reasonably small number of different values and each value should appear many times. A bar chart that is very wide and low indicates that these qualities are not present in the variable. Figure 1.5 shows an example. It is the bar chart generated from a variable called *FirstName* and you should be able to see that there are a great many different values and they are all quite rare. First name is usually a poor choice of variable for a machine learning model because there is rarely a good relationship between it and any outcome you might like to predict. The meaning of first name is enough to tell us that, but other variables that have the wide and low quality are equally poor choices as there is little scope for generalisation if each value is rare and the resulting model is likely to be very complex as it has to take into account every possible value. Such variables contribute to low bias and high variance.

Some variables that have a large number of nominal values can be re-coded into a smaller set of named groups. The group names are then used as input values rather than the original values. For example, specific job titles (*Teacher*, *Bricklayer*, etc.) might be recoded as *WhiteCollar* and *BlueCollar*. Not only does this reduce the complexity of the model, introducing bias and encouraging better generalisation, it also forces you to encode the meaning behind the variable. This can sometimes add information and sometimes introduce assumptions that might not be right, so care is needed.

Histograms and bar charts can also reveal data entry errors and other anomalies such as outliers and rare values. Figure 1.6 shows the bar chart of a discrete variable that reports home ownership status. You can see from the chart that there are millions of data points represented and that the possible values with any real data behind them are *Own*, *Mortgage* and *Rent* but there are a very small number of values called *Morgage*. Clearly, they are data entry errors. In other cases, they might be genuine rare values (*Hotel* for example), in which case a decision would be needed about whether or not to include that data in the model. Including it makes the model more complete as there are no cases it cannot handle, but it also makes the model more complex, as discussed above.

Outliers are values from continuous variables that are far from the bulk of the other values. In normally distributed data, this might mean values that are more than 3 standard deviations from the mean. In more complex distributions

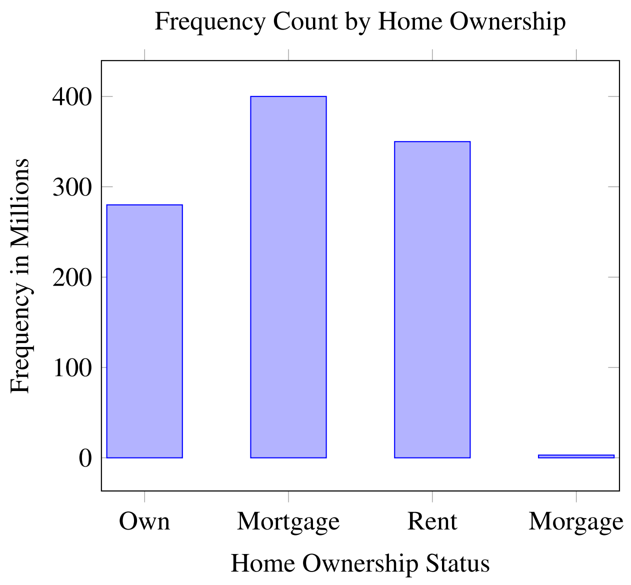


Figure 1.6: An example bar chart with a minority value, in this case due to a typing error when entering *Mortgage*. The solution to this is obviously to correct the errors, but in other cases, the minority class may be of interest and other methods are needed to address the imbalance.

(with multiple modes, for example) it means a value that has few near neighbours. Figure 1.7 shows an example histogram that contains outliers, which are much higher than the rest of the data. Outliers may be data entry errors, in which case the row they are on should be corrected (if possible) or deleted otherwise. Outliers can make the task of learning more difficult for many machine learning techniques and skew the size of the errors they report.

Variables that have potential problems, but which may still be useful should be flagged so that the problems can be taken into consideration later when variable selection decisions are made (see section 1.5.2). In some cases where the data problems are with the output variable, there is not the option to exclude the variable and neither is there the option to delete the problem cases. For example, if the output variable is categorical and one category is in a profound minority, that minority value cannot always be ignored. In fact, it is often the value of interest. Minority output classes like customers who responded to a direct marketing campaign or patients with a rare disease are the very classes the model is being built to spot, so they must be included in the model. Such variables are called **unbalanced classes**. Figure 1.8 shows a bar chart of an output variable with a minority class and section 1.12 discusses methods for handling such minority values.

A final point to make about data quality concerns whether or not the data contain the information required to perform the desired task. If there is no relationship between the inputs and the output, then no accurate model can be built. A more dangerous situation occurs, particularly in the case of classification tasks when the data contain the information required to split the examples, but not for the right reasons. An old story of this type involves a project designed to teach a computer to discriminate between American and Russian tanks. The algorithm learned the data well and performed to a high level of accuracy on test data. In the field, however, the performance was terrible. It turned out that the images of American tanks that were used during training were high quality images against plain backgrounds and the images of the Russian tanks were poor quality surveillance photographs. You can no doubt guess what the algorithm really learned to discriminate.

Data Quantity

"*Do I have enough data to build a model?*" is a common question in some applied fields of machine learning. Applications in which data are rare or expensive to collect may require careful consideration before a model is built. The following aspects need to be considered when data sets are small:

- the number of input variables being modelled
- the number of different values nominal variables can take

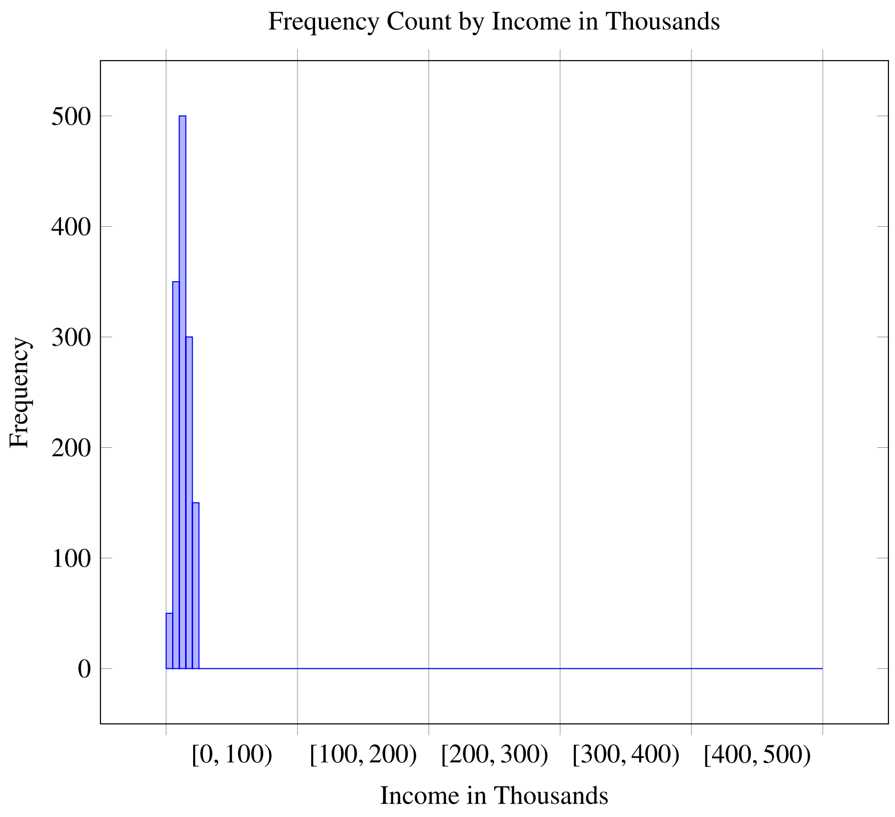


Figure 1.7: A histogram that is the result of an outlier in the data. The bar representing the outlier cannot be seen as it is so small, but the fact that the *X* axis goes up to 500 suggests that the outlier is close to that number.

- how close to uniform the distribution of values in each variable is
- the number of different classes in the output (in cases of classification)
- how much noise there is in the data
- the intrinsic dimensionality of the data (or the degree of correlation among inputs)

The more variables and classes you have, the more data you need to record a sufficient number of combinations of values across them. The number of possible combinations of values grows exponentially with the number of variables, but it is likely that the full space is not covered uniformly so it is often the case that the number of data points required to build a model does not grow exponentially with the number of variables. Similarly, the more different values a variable can take, the more samples you need to experience its full domain in combination with other variables.

The distribution of values in each of the variables also plays a role in determining the required data quantity. Unbalanced variables need more samples to be sure of collecting sufficient examples of the rare values. This is particularly true when the output variable is imbalanced and the rare values represent the cases of interest. Some data sets contain more noise or more variance around the output than others. A data set that contains customers' responses to a survey paired with their buying behaviour is low quality because people are not always honest on a survey and those data are not great predictors of buying behaviour. On the other hand, measurements taken from a machine that is producing a product of variable quality (a paper mill or a distillation process, for example) can be measured accurately and there may be a deterministic relationship between the measurements and the machine output. The customer survey example will need more data to average out the variation than the machine modelling example.

It is often the case that input variables are correlated with each other to some extent, meaning that the input data form clusters or areas of high or low density. If much of the potential input space is never actually represented in the real world measurements, then far less data is needed to cover the relatively small areas that are dense with data.

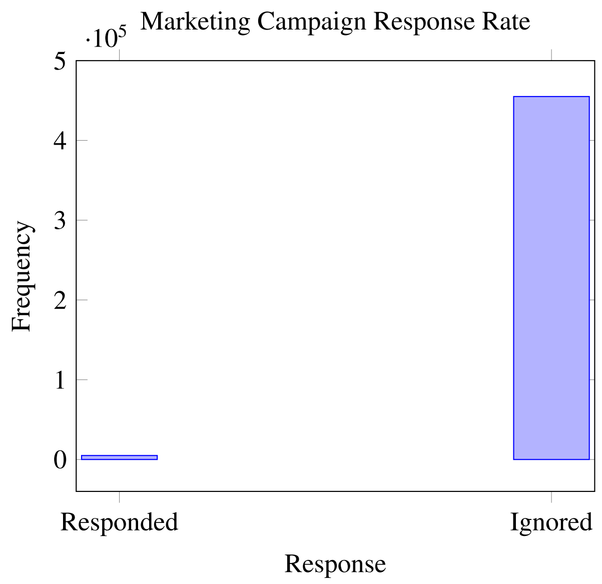


Figure 1.8: An example of an imbalanced output variable plotted in a bar chart. The data are the response counts from a direct marketing campaign and show that only 5000 people out of a total of 500,000 responded. The imbalance will need to be addressed when the data are modelled using a machine learning technique.

Consider a data set that measures age, marital status and number of children a person has. The area that represents people under 16 will have no examples with a marital status of anything other than single and will be very sparse at the high end of the number of children scale. This reduces the dimensionality of the data to some extent. This new smaller space is referred to as the **intrinsic dimensionality** of the data.

Finally, as described in section 1.3.1, the available data needs to be split into training, validation and testing subsets, further reducing the quantity of data available for building the model.

Modelling

The modelling process is where the textbook machine learning methodology is applied to attempt to find the model with the right level of bias and the best ability to generalise. This process, described in more detail in section 1.5, builds many different models and validates their ability on data that were not used to train them. There are methods for splitting and sampling the data used for training and validation, methods for searching for a good model, and methods for dealing with some of the problems in the data described above.

Evaluation

There are two stages of evaluation during a machine learning process. One is part of the search for a good model that takes place during the modelling process, and is known as **validation**. The other is a final evaluation of the quality of the outcome of the modelling process and is called **testing**. A subset of the data is set aside for testing before any decisions are made about data preparation or modelling. This is known as the **test data** and is used to provide a final evaluation of the modelling process but not to influence it in any way. It tells you how well you might expect your model to work on new data once it is deployed if nothing changes in the system that is generating the data.

The model evaluation considers the overall accuracy of the model and the ways in which the output of the model differs from the observed data. Such differences are known as **errors** but when statisticians talk about error, they do not mean mistake. A statistical error is not like a bug in a computer program, it is a (usually) unavoidable consequence of modelling uncertain systems with statistical models. The error when making a prediction is the difference between the true outcome and the prediction. When making a classification, the answer is usually either correct (no error) or incorrect (an error). Taken over a whole test data set, the percentage of correct classifications is known as the model accuracy.

Section 1.3.1 discusses cost functions and explains how learning rules attempt to minimise a cost function when estimating parameter values. Sometimes the cost function is the same as the reported error rate, but there are many other error or accuracy measures available. They are discussed further in the relevant sections below. Finally, note that not all errors are equal. In many classification tasks, the cost of one type of misclassification is higher than that of another. In medical screening, it is worse to miss a disease that is present than to send a patient for further tests that turn out negative. In insurance fraud detection it is more costly to pay out on a fraudulent claim than it is to run a few further checks.

Deployment

If the model's test performance is satisfactory then it is deployed. Recall that we pointed out that the result of machine learning analytics is usually a system capable of performing a task rather than a written analysis report. This is why the process is sometimes called predictive analytics. While the requirements for training such models can be high in terms of data, time, human expertise and computer power, running a model once it is built is fast, cheap and requires no further access to the original data. To make live predictions (or classifications) a model needs a small amount of code to interpret its parameters and a file that represents the parameters and the structure of the model. These can run in any programming language you choose, so are easy to embed in websites, call centres, machine monitoring systems, medical equipment, car engines, microwave ovens, mobile phones, home security systems and, well, you get the idea.

At any stage during the CRISP-DM life cycle, it may be necessary to retreat to an earlier stage to collect more data, better understand the business or the system behind the data, try a different modelling technique or simply update an existing model. While some systems offer real time online learning, it is often the case that a human expert is needed in the loop if something fundamental has changed in the data being processed.

1.5 Machine Learning Methodology

This section describes in detail the process of building a reliable statistical model from data. The goal is to use a sample of data to best approximate the system that generates a larger (sometimes ever expanding) population of data. The measure of success is the model's ability to generalise to data that was not used during the training and validation phase. The task during the training and validation stage is to find a technique and a specific model with a particular level of bias that is best suited to the data at hand. The training data are used to estimate the model parameters and the validation data are used to compare the many different models that will be built. Once a method and its associated settings have been chosen, a final model can be built using all of the training and validation data, ready to be tested with the test set.

1.5.1 The Model Building Search Space

We have said that modelling a data set is a process of choosing the right technique, the right settings, and the right level of bias. It is time to be more specific about those things. When we talk about a machine learning **technique**, we are referring in the broadest sense to the way the model is represented. For example, we might talk about techniques such as multilayer perceptrons or decision trees. For each technique, there are usually more than one **learning algorithm** and more than one choice of **cost function**. These two things combine to dictate the approach to parameter estimation that a technique uses. There are also a set of options concerning how the learning algorithm works. These are generally called **hyper parameters** and they can be thought of as dials that control different aspects of the algorithm's attempt to minimise the cost function. Hyper parameters will be discussed in more detail in the sections on specific techniques as they are often specific to a technique.

Finally, in addition to the choice of technique and the parameter fitting, there is the question of **structure discovery**. Each statistical model represents the mapping from input to output as a parametrised structure. Some models are simple linear equations, such as the standard $Y = aX + b$ of a linear regression where a and b are the parameters to be estimated and the structure is defined by the formula. Other structures include trees of branching decisions, graphs of nodes connected by edges, and series of connected non-linear feedforward functions. Some models do not require structure discovery; a standard linear regression model has a single parameter for every input and that is that. Others do, for example a decision tree is built by choosing which variable to place at each decision node. Structure discovery will be discussed further in the relevant sections for those techniques that need it later in this chapter.

In addition to all of the technique related decisions, there are choices to be made about the data. Which variables should be included is known as the problem of **variable selection** and how variables are combined or manipulated to help the algorithm learn the correct task is known as **feature selection**. There are methods for over-sampling the data to change its distribution and tricks such as normalising the data before training that help to speed the learning process or avoid local minima. On top of that, there are the so called **ensemble** methods that use a mixture of models, all built in different ways, to offer a panel of predictions rather than a single one. Using an ensemble introduces hyper parameters of its own such as how to combine the set of predictions into a single answer.

So many options mean that the space of possible models that might be built from a single set of data is huge. To explore that space efficiently, moving towards a better solution requires a good methodology, and that is what this section describes. When talking about this search, we will use the term **approach** as a catch all for the combination of all the options described above. One approach, for example, might involve a multilayer perceptron (the technique) trained with gradient descent (the learning algorithm) on a squared error cost function with given settings for learning rate, early stopping, hidden units and activation functions (the hyper parameters). Taken all together, these constitute the learning approach.

Training, Test and Validation

As described in section 1.3.1, the available data is used in three different ways, for training, validation and testing. The test data is separated first and not used again until a final model has been built and is ready to be evaluated. It is not used to guide any design or data preparation decisions. However, the pre-processing carried out on the training and validation data, once finalised, must also be applied to the test data before it is used to evaluate the model. Again, this is a final, once only process and the test data should not be used as part of an exploration of pre-processing decisions. The size of the test set varies but 30% of the available data is a common proportion to set aside. It is important that the test data is chosen uniformly at random to avoid any similarities that might be a result of the order in which the data is stored in a file.

The training and validation data have a more fluid relationship, with data sometimes being used for training and sometimes for validation. The simplest approach is to split the remaining data (after removing the test portion) into two sets with around 70% used for training and 30% for validation. This is convenient as the split can be done manually and only once. It also speeds up the process of trying many different approaches as each can be trained and validated on a single set of data. The disadvantage of the single validation set is that it reflects the different approaches' abilities on a single subset of the data. This does not provide a sufficient level of confidence, as the variation in error between two identical approaches trained on two different validation sets can often be greater than the variation in error between two different approaches trained on the same data.

This problem is addressed using an approach called **k-fold cross validation**, in which the available data is split into k different non-intersecting subsets. Then, k different models are built, each using $k - 1$ of the subsets and validated on a different remaining one. This means that every data point features in a validation set once. It is common for the value of k to be 10 and you will see the process referred to as 10 fold cross validation. In this case, each of the ten models is trained on 90% of the data and tested on 10%. If you have limited time, 5 or even 3 fold cross validation can be performed.

Having trained ten different models using cross validation, you will have ten results. In their basic form, this will be ten cost values (or error measures). You can report the mean of these values as a measure of the quality of the modelling approach and the variance among them as a measure of how robust the approach is to different data samples. If you want to train a different model, then you perform cross validation again, and record the mean and variance of the measures that produces. You can now compare the two (or more models, if you build them) to see which is likely to give the best generalisation performance. Once you have chosen the winning approach, you can train one final model, using all the training data, and test it once and for all on the test data. Note that you do not keep any of the models built during cross validation - they are for comparison only. The final model, trained on all the training data, is the one you keep.

Cross validation provides a better estimate of the validity of a given approach but takes longer to perform as k different models need to be trained. There are also other sampling methods, such as boosting, which is described in section 1.11.2.

You can take the cross validation idea further and apply it to the whole process of searching for the best hyperparameter setting - repeating that process k times too. This is called **nested cross validation** and involves an outer loop that splits the data into k different training sets and then each of those training sets has k fold cross validation applied to it. This further reduces the bias in the performance measures estimates as they are an average of several repetitions of the process. It is also very time consuming, so cannot be used in all cases.

1.5.2 Feature and Variable Selection

Imagine you have bought some data about your customers from a consumer data provider. They have supplied data for 120 different variables about each of your customers and you intend to use that data to build a predictive model that can differentiate high spending customers from the rest. Which of the 120 variables should you use? Surely the more information you have the better, so is it best to just use them all? Unfortunately, no it is not. For any set of data and task there is an optimal subset of variables that lead to the best model (i.e. the one that generalised most accurately to new data). That subset is rarely just one of the variables and rarely all of the variables you happen to have collected. Using too few variables leads to a model that is missing important information and is not as accurate as it might be on the training and validation data.

Using too many variables can lead to a number of problems. It adds to the complexity of a model and increases the risk of overfitting, so the training error might reduce when more variables are added to a model but the validation error might increase. Adding more variables to a model also increases the number of data points required to build the model. Take a single variable from the demographic data as an example: household income. To get a reasonable spread of the different income bands and their effect on expenditure at your company, you might need 100 data points. No consider age as the next variable. If we need 100 data points to span the various age bands effectivelty, then we need $100 \times 100 = 10,000$ data points to cover the joint space of age and income. Add a third variable and the requirement is multiplied again. Every time you add a variable, the quantity of the data required to provide a useful sample across the possible combinations grows exponentially. This is known as the **curse of dimensionality** and it means that if your data set is small and you use too many variables, the data will be very sparsely spread out, or concentrated in a very narrow space. Both situations can lead to over fitting and poor generalisation. As we mentioned before, correlation among variables can reduce the effective dimension, but correlated variables can introduce modelling problems too, so the problem remains. For this reason, variable selection tries to find a subset of variables that are related to the output but uncorrelated with each other.

There are two ways in which you can reduce the number of variables used by a model. The first is to not use all of the available variables (obviously) and the other is to combine several of the variables into a new single variable. The first approach is known as variable selection and the second as feature selection.

Variable Selection

Variable selection is also known as attribute selection or variable subset selection and involves picking a subset of the available input variables for a model. It is a way of introducing model bias by simplifying a model that would otherwise have more variables and more complexity. Given a set of candidate input variables, $\mathbf{C}$ and an output variable, Y, variable selection attempts to find $\mathbf{X} \subset \mathbf{C}$, where $\mathbf{X}$ may be used to predict Y. It is desirable that $\mathbf{X}$ is minimal (contains no more variables than it needs) and accurate (allows an accurate predictive model to be built).

The important point to understand concerns how the dependence among members of $\mathbf{C}$ affects their ability to predict Y. Take two variables, X_1 and X_2, which are thought to have an effect on Y. If the effect on Y of changing X_1 does not depend on the value of X_2, then X_1 is **independent** of X_2. If we cannot know how changing X_1 will affect Y unless we know the value of X_2, then the effect of X_1 on Y is said to be dependent on X_2. It is also said, in such cases, that the relationship between (X_1, X_2) and Y is **non-linear**. For example, it might be that *age* or *income* alone predict spend poorly, but predict it accurately when combined (perhaps young rich people and older poor people are the two groups that spend most). The consequence of such dependencies is that any variable that seems unrelated to the output in isolation may actually play a crucial role when paired with one or more other variables. Just because X_1 and Y are uncorrelated and X_2 and Y are uncorrelated does not mean that (X_1, X_2) might not perfectly predict Y. Consider the logic function, XOR if you want a simple example.

An initial phase of variable selection is performed in the data preparation stage where problematic variables are removed. For example, the customer profiling data may have a customer ID or customer name fields. These contribute nothing to the learning (they have a wide and low distribution) and should be removed. This phase may also involve human domain knowledge. Variables may be removed if they are known not to contribute to the predicted outcome. Care is needed not to make incorrect assumptions at this point and as there are good analytical methods for discovering a good subset of variables, the default assumption should be to include variables if there is any doubt.

Most variable selection takes place at the second phase of the process, the automated phase. This involves a search for good variable subsets guided by a cost function. In some approaches, the cost function is the same as that used to estimate the parameters of a statistical model, as described in section 1.3.1 and in others a different cost function is used. This distinction will be discussed in more detail in the following sections. Most machine learning software provides a set of automated variable selection methods and they fall into three broad categories: **filter methods**, **wrapper methods** and **embedded methods**. The categories are characterised by how well the two cost minimisation approaches (variable selection and parameter estimation) are integrated.

Filter Methods

Filter methods are a pre-modelling step and can be run independently of the machine learning techniques that are subsequently applied. They generate a subset of variables that are then fixed and used without much further alteration in the model learning phase. This has the advantage of being time efficient. You choose the variables once and then move on to the challenges of picking the right model and the right hyper parameters without needing to repeat the variable selection process for every model you build. Its independence from the model building process is also a disadvantage as the choice of variables cannot be guided by the abilities or structure of the models being built. It also isolates the choice of variables (an important consideration when optimising bias) from all the other choices that need to be made to that same end. If the effectiveness of a variable subset depends on the choice of model and hyper parameters, then a pre-modelling filter selection cannot account for that dependency. The filtering cost function is a fast approximation to the real cost function, which is how well a machine learning technique might use the chosen variables to perform the given task.

Wrapper Methods

Wrapper methods [**?**] use the error from the chosen machine learning technique as the cost function to be minimised by variable selection. They choose a variable subset that is best suited to precisely the task at hand—building a model that minimises a chosen cost function. No changes to the machine learning technique are needed and the approaches to picking variables described in section 1.5.2 below can all be applied. The process is to pick a variable subset, use it to train and validate a model (including hyper parameter optimisation), use the resulting model's accuracy as the measure of subset quality, change the subset to effect an improvement, and repeat. One advantage of wrapper methods is that they embed the variable selection process into the model building process, allowing dependencies between the chosen variable subset and other hyperparameters to be accounted for. The cost function is also more accurate as it is the result of a full model training process rather than the approximation used by filtering methods. The disadvantage is that the process takes a lot longer as each step of the variable selection process requires a full model build and hyperparameter search (see section 1.5.3).

Embedded Methods

Some statistical models represent the relationship between inputs and outputs in a way that is human readable and some use a more opaque representation. What is more, some modelling techniques include feature selection as part of the modelling process, and some do not. Models that are human readable and perform variable selection as part of the modelling process offer an attractive combination of qualities when variable selection is required. The model can be built using all of the candidate variables and then examined to see which variables it used and which of those play an important role in the model.

Two very good machine learning techniques that embed variable selection are decision trees (see section 1.6.4) and a regression method called LASSO (see section 1.6.1). Both are able to discard variables that do not contribute to minimising the cost function and both produce a model that may be interpreted by a human so that useful information about which variables are important can be extracted.

The advantage of embedded methods is that no pre-filtering or complex search is needed in addition to the model building. The processes of variable selection, structure discovery and parameter estimation are all carried out at the same time with the same cost function. This is obviously quite efficient. One disadvantage is that you need to build the first model from all of the variables, which can increase the size of the required training set and increase the risk of over fitting unless the right level of regularisation is found.

Selecting a good variable subset, guided by a cost function, is an example of a search problem and the search space can be very large. If you have p candidate variables, then there are 2^p possible subsets that you might select and you cannot try them all. The many ways in which the search may be carried out all follow the same basic pattern: choose a subset of variables, evaluate its quality with the cost function, make a change to the subset, and repeat until either a satisfactory subset has been found or so further improvement looks possible in the remaining available time. The next sections describe each of the ingredients in such an algorithm in a little more detail.

The Initial Subset

There are three approaches to selecting the initial subset of variables. One is to use human domain knowledge in cases where some variables are known to be important (and are included) or some variables are known to be expensive to collect (and are excluded). The other two are a choice between starting with a minimal or empty set and adding during the search or starting with a full set and removing variables during the search. These last two approaches are known as **forward** and **backward** selection methods respectively.

The Cost Function

Embedded and wrapper methods make use of a chosen machine learning technique, so the cost function they use is the same as that used to estimate the model parameters. Filter methods are independent of any machine learning technique, so the choice of cost function is not so constrained. In filter methods, there are three aspects to the cost function that define a good subset of variables. One measures how well the chosen input variables predict or classify the output variable. The second measures redundancy among the chosen inputs. If two inputs are highly correlated, then one adds little to a model that already contains the other. Finally, cost functions should attempt to keep the number of variables used low, so given two competing candidate variable subsets with very similar scores for predictive power and redundancy, the smaller should score higher.

The Subset Update

We have already mentioned that there are two choices of direction for a variable subset search: forwards and backwards. In the simplest cases, updating the subset involves adding (in the forwards case) or removing (backwards) the variable that improves the cost function the most. The process stops when no improvement is possible. Cost functions that take the number of variables into account should be improved as variables are removed during backwards elimination. Cost functions that do not account for the number of variables must remove the one that causes the least increase in cost. These methods have the advantage of being reasonably fast as they make very few iterations of the search algorithm, but they only search a small proportion of all possible combinations, so will rarely find the optimal subset.

More complex methods of adding and removing variables from the subset at each iteration are available. Some add variables that make a contribution and remove any that are rendered redundant by the recent additions. Examples of this approach include stepwise regression [**?**], which adds variables one at a time based on their correlation with the output and removes any that are rendered insignificant in an F-test as a result of recently added variables. Hall [**?**] proposed a similar correlation based feature selection method, which includes variables that are correlated with the output but removes variables that are correlated with those already selected for inclusion in the model. The max-dependency, max-relevance, and min-redundancy (mRMR) approach [**?**] to feature selection uses a measure of mutual information between the input variables and the output class alongside measures of mutual information between different input variables to attempt to maximise the dependency of the output on the input while minimising the shared information (i.e. redundancy) between inputs. More complex approaches such as the use of genetic algorithms have also been proposed for variable subset selection. For example, Bala et al. [**?**] use a hybrid GA and wrapper approach to feature selection. Cantú-Paz [**?**] compared GAs with three other evolutionary computing methods, namely Estimation of Distribution algorithms (EDAs), Compact GAs, and Bayesian Optimisation Algorithms (BOAs) in terms of their ability to perform feature selection (this last point is outside the scope of this book, but might interest some readers).

1.5.3 Hyper-Parameter Search

Finding the right set of variables for your model is one search task to complete, and another is to find the right model. There are a great many choices to be made at this point, from the type of model to build to the details of how the model is built. This section introduces the concept of the **hyperparameter**. Let us first remind ourselves of the difference between a parameter and a hyperparameter in statistical modelling. A parameter influences the shape of the function a model represents. For example, in $Y = aX + b$ the parameter a determines the size of the change in Y that a small change in X will produce. A hyperparameter influences the process of estimating the parameters. If the a parameter was estimated using gradient descent, then the learning rate and the number of steps taken would both be hyperparameters and different settings for them might produce different values for a. It would not be possible to say which of the values for a was the correct one, but they could be compared by comparing each model output to the known output values in the validation data. In fact, the primary job of the validation data set is to provide feedback on the hyperparameter settings. In other words, in the search for good hyperparameter settings, the cost function being minimised is defined in terms of the validation data. Of course, the training cost function used to estimate the parameters (such as a) is defined in terms of the training data.

So: the training data are used to minimise the cost associated with the model parameters and the validation data are used to minimise the cost associated with the hyperparameters.

The hyperparameters are a set of controls that influence the model's ability to generalise, but in hard to predict ways. The search process, like many we meet in this book, involves generating a solution, evaluating it, and then changing it slightly in light of its recent performance. There are heuristics to guide the choice of some hyperparameters, but not all so a strategy is needed for the search. This section describes four different approaches: the **grid search**, the **random search**, **coordinate descent** and **manual search**. The first three are automated approaches but manual search, as the name suggests, relies on human decision making as the search progresses.

Hyper-parameters are usually specific to a learning algorithm, for example gradient descent methods often use a learning rate, neural networks require a choice about the number of parameters they contain and decision trees offer control over the number of examples required to add a decision node. Technique specific hyperparameters and the heuristics associated with their choice are discussed in section 1.6. For the overview of search methods in general, let us assume a set of k hyperparameters in a vector $\eta = \eta_1 \ldots \eta_k$ where each hyperparameter may be numeric or nominal (some take a setting from a numeric range and some take a choice among different named options).

Grid Search

A grid search is a method for exhaustively covering the hyperparameter search space in discrete steps. It splits the numeric hyperparameters into discrete steps which, when combined, form a grid. Nominal valued hyperparameters are already discrete, of course. Models are built using combinations of hyperparameter values at the intersections of the grid. For example, if η_1 is numeric in the range 0 to 1 and η_2 is nominal with possible values of a, b, or c, then we might choose to split η_1 into eleven steps: 0, 0.1, 0.2 . . . 1. This would give grid points at $a, 0$, $a, 0.1 \ldots b, 0, \ldots c, 1$, a total of $11 \times 3 = 33$ combinations to try. Each new hyperparameter that is added to the search multiplies the total by its cardinality (3 and 11 in this example) so care is needed when deciding how many splits to make. You can see why cross validation and a grid search can be time consuming.

A slightly more sophisticated approach is to use variable grid sizes, picking promising areas with a wide grid and then using a finer grained search to explore close to points that have a lower validation error. Another improvement can be made by choosing specific points along numeric ranges rather than using a fixed interval. You might first establish the order of magnitude (try 0.1, 0.01 and 0.001, for example) and then use a finer search when that decision is made.

Random Search

You might be surprised to learn that randomly picking points instead of methodically working through combinations can be much more efficient. This is because a limited time budget can be better spent exploring widely. A methodical approach may run out of time while concentrating in a small corner of the search space. Random search can also be extended to allow an algorithm to search more methodically around promising points but ignore those with a low score. The distribution from which points are picked may be uniformly random or be biased towards values that are known to be promising based on human expertise. There are also methods for updating the distribution as the search progresses to force it to pick from promising areas with a higher probability.

Coordinate Descent

Coordinate search attempts to concentrate on more promising areas of the search space by optimising one hyperparameter at a time, while holding the others constant. Hyper-parameters are usually not independent and the optimal value of a chosen hyperparameter will depend on the current values across the others, so the process is iterated several times. The process starts at a chosen (usually random) point and picks one hyperparameter to search. The best value for that hyperparameter is found and fixed to be constant. Then, another single hyperparameter is chosen and the process is repeated. The order of choosing each hyperparameter may be cyclical (η_1, then η_2, then η_3 etc.) or random or guided by knowledge of which parameters should be searched more carefully. The cycle is repeated several times in an attempt to continuously reduce the validation error. The process terminates either after a specified number of cycles or when no improvement can be made from the current point in any direction. This final point is a local optimum but may not be the global optimum.

Manual Methods

When you have built up some experience modelling data using machine learning, you start to spot certain patterns. This is especially true if you work in a specific field (data scientist in the marketing department, for example). You will know what combination of hyperparameters often works best and you will known what the most promising lines of search would be from those points. This knowledge can speed up the hyperparameter search considerably. In a commercial setting where requests for a working model come with a deadline, a manual search can be an efficient way of finding a satisfactory solution fast.

The advantage of manual search is that the human expert, with experience of similar problems and an ability to spot patterns that would be beyond the best automated search algorithm, may be able to find a better solution faster. In practical machine learning tasks this is an important advantage. In academic circles, when using hyperparameter search to test a newly proposed learning algorithm, there is a preference for automated methods because they can be properly specified and reproduced by other researchers. The apparent success of a new learning method is less impressive if it required the skills of its originator to wring the best performance out of it.

There are many other approaches that might be taken, including Bayesian methods, which are becoming popular [**?**], but the simpler approaches listed above should generally be considered first. It should also be noted that different data sets (and consequently different error functions) have varying degrees of sensitivity to hyperparameter settings. Some problems are easy to solve regardless of the settings (within sensible bounds) and others produce very different results depending on the training regime. This rather abstract treatment of hyperparameter search will be made concrete in section 1.6 as the hyperparameters for various different approaches are introduced.

1.5.4 Error Analysis

The cost function chosen for the machine learning process is one measure of the accuracy of the model, but there are many others that can be reported once the model is built. Remember that an error in machine learning terms means the difference between a predicted outcome and the outcome in the data. Error, then, is defined in terms of a set of data, not a known ground truth. This is why we talk about training error, validation error and test error—they are all related to a given data set.

Classification Accuracy

When reporting error for classification tasks, there are a number of measures that can be used but they are mostly based around the number of correct or incorrect classifications made. Take a data set with a single categorical output variable, Y that can take values in $\{a, b, c, d\}$. Once a classifier is built, it is tested by presenting example input patterns where the output class is known and comparing the model output with the class label in the data. If they match, the count for *Correct* is incremented and if they do not match, *Incorrect* is incremented. The simplest measure, known as **accuracy** calculates the proportion of classifications that are correct. It is often written as ACC and is calculated using equation 1.15

$$ACC = \frac{Correct}{Correct + Incorrect} \tag{1.15}$$

It is more useful to know if there are any patterns to the errors, for example which classes the model gets right, which it gets wrong, and when it does make an error, what form that error takes. This information is summarised in a **confusion matrix**, which counts the number of times the model classified an example that is known to be in one class as belonging to each of the other classes. The confusion matrix shows how many times data that were labelled as belonging to class a were classified by the model as being in a, b, c, or d respectively, and then the same for classes b, c, and d. Each row in the matrix represents a labelled class (known as the **actual** values) and each column indicates the

Predictions

Data Labels	*a*	*b*	*c*	*d*
a	76	4	8	0
b	3	68	7	7
c	4	5	71	6
d	3	7	6	80

Figure 1.9: An example confusion matrix with possible output classes a, b, c, or d. Rows represent the labels in the data and columns are the model output. Each cell contains a count of the number of times the model predicted the column's value and the data contained that of the row. The diagonal from top left to bottom right represents correct answers. For example, the model predicted d when the data contained d 80 times.

model outputs (sometimes labelled **predictions**—see section 1.2.2). The diagonal of cells from top left to bottom right are those where the model agrees with the data and the rest show errors. The higher the values on that diagonal are, compared to the rest, the better the model performance. Figure 1.9 shows an example confusion matrix and describes how to interpret it.

In cases of binary classification, such as fraud detection, it is useful to label one of the two classes as the *Positive* case (Fraudulent) and the other as the *Negative* case (Genuine). *Positive* usually means class of interest rather than reflecting a quality of the class (fraud is not a positive thing!) There are two ways in which an error might be made. Classifying in the positive when the label in the data is negative is a *False Positive* (e.g. classifying a transaction as fraudulent when it is not). Classifying in the negative when the label in the data is positive is a *False Negative* (e.g. classifying a transaction as genuine when it is fraudulent). There are also two ways to get the answer correct: *True Positives* and *True Negatives*. These are counted for a data set by processing each data point in turn and incrementing the appropriate count. The resulting counts are written as TP, TN, FP and FN.

The counts are used to calculate **rates**, so we talk about the true positive rate for example, which is the number of true positives divided by the number of labelled positives in the data. These are usually written as TPR, TNR, FPR and FNR. Let the number of positive cases in the data set be P and the number of negatives be N, then we calculate the rates as given in equation 1.16

$$\begin{aligned}
TPR &= \frac{TP}{P} \\
TNR &= \frac{TN}{N} \\
FPR &= \frac{FP}{N} = 1 - TNR \\
FNR &= \frac{TN}{TP + FN} = 1 - TPR
\end{aligned} \tag{1.16}$$

The true positive rate is also known as the **sensitivity**, **hit rate** or **recall** of the model. It measures the proportion of the true classes that are correctly identified, for example what proportion of fraudulent cases are spotted. The true negative rate is also known as the **specificity** and you will also see **precision** reported, which is the number of true positives divided by the number of times the model reports a positive ($TP/(TP + FP)$). Precision measures the fraction of positive predictions that are really positive (when the classifier says "Genuine", how often is the data labelled as genuine?).

If the class labels in the data used to assess the model accuracy are imbalanced (some are more common than others) then the level of accuracy you could achieve by chance increases. If there are 100 examples of fraud in the data used in the current example and 900 genuine examples, then a trivial way of achieving a 90% correct classification rate is to always predict a genuine case. That is quite accurate, but entirely useless. In such cases it is good practise to rebalance the data before training (see section 1.12) but in cases where that has not been done the error measure should take into account the accuracy you would achieve if you made guesses with the same distribution of class labels that your model does, but without using the input variables at all.

The **kappa statistic** measures agreement between the class labels in the data and the labels given by the classifier, taking the distributions of both the output class and the model's predictions into account. It is calculated as $(ModelAccuracy - ChanceAccuracy)/(1 - ChanceAccuracy)$ where $ChanceAccuracy$ is calculated from a mix of the distributions of the data and the model output when it classifies that data. It calculates the level of accuracy you would expect if you classified the data by drawing classes at random from a distribution the same as that displayed by your model's predictions on that data. So if our model always predicts class *a* and never *b* then the chance level of accuracy is equal to the proportion of the data that is labelled as class *a*. If your model predicts class *a* half the time and class *b* half the time, then the chance accuracy level is half the proportion of the data labelled *a* plus half the proportion of the data labelled *b*. Formally, the chance accuracy for two classes *a* and *b* is calculated as

$$ChanceAccuracy = PD(a)PM(a) + PD(b)PM(b) \tag{1.17}$$

where $PD(a)$ is the proportion of the data that is labelled as class a, $PD(b)$ is the the proportion of the data that is labelled as class b, and $PM(a)$ and $PM(b)$ are the same proportions in the model output set having classified all of the data. The sum is simply expanded for more than two classes.

One other measure is the **F1 score**, which is $(2 * Precision * Recall)/(Precision + Recall)$ and measures a mixture of precision and recall. The F1 score ranges between 0 and 1, where 1 means perfect recall and precision. Sometimes, the cost of a false positive is different from that of a false negative. In our fraud detection example, missing a case of fraud is more expensive than investigating what turns out to be a genuine case. In such cases, the accuracy measure may be weighted so that one type of error costs more than the other.

Prediction Accuracy Measures

Prediction models have continuous numeric outputs so the measures of accuracy can take the distance between the data and the associated predictions into account. As mean squared error (MSE) is a common cost function to minimise when training a predictor, it is also a common accuracy measure to report. It measures the average squared distance between data outputs and associated model outputs. Given a data set **D** of N paired input and output values (x, y) and a model, $f(X)$, MSE is calculated as

$$MSE = \frac{1}{N} \sum_{(x,y)\in\mathbf{D}} (f(x) - y)^2 \tag{1.18}$$

As the range of Y can differ from model to model, the MSE is not always easy to interpret. The correlation between the model output and the values in the data can be easier to interpret because regardless of the range that the data covers, the correlation coefficient (cc) can only ever be between -1 and 1. A cc of one means the model is perfect (subject to the assumptions described next) and a cc of zero indicates no relationship between the predicted and actual outputs. The cc should never be negative as that would suggest your model actively contradicts the actual data.

When we use the cc to measure accuracy, we are making some assumptions. We assume that the predicted and actual outputs are, on average, the same with some variation, which is what the cc measures. To ensure that the relationship between the predicted and actual data is suitable for measurement with cc, you should make a **scatter plot** of predicted against actual values from the data. Most software packages will do this for you, so we will concentrate on how to interpret the plot, rather than creating it. Scatter plots are discussed in general terms in section 1.13.1. Figure 1.10 shows two typical model output scatter plots with values from the true outputs from the data along the X axis and predicted values from the model up the Y axis. See how the cloud of data follows a straight line at 45 degrees from the origin but how one model's predictions are closer to that line than the other. The scatter plot is a little like the confusion matrix. An accurate model will produce a scatter plot with little variation about the 45 degree line that runs from bottom left to top right on the plot. The wider the data cloud, the larger the errors made by the model.

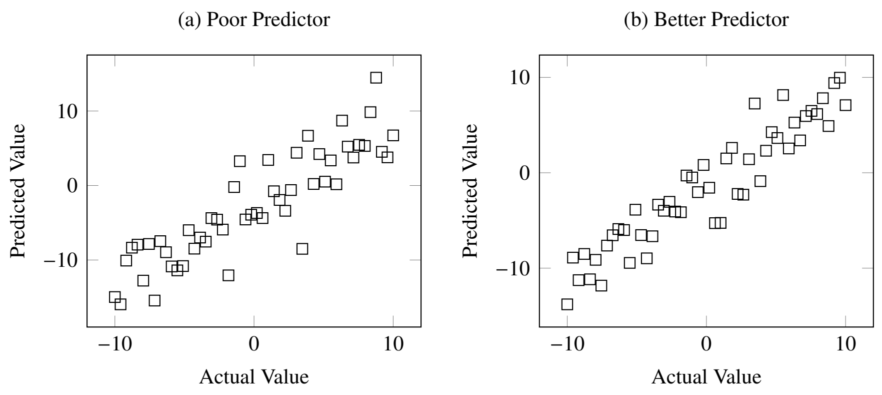


Figure 1.10: Scatter plots of predicted value against actual value for two models. The model that produced plot (a) has a higher error than the model that produced plot (b).

1.5.5 Model Interpretation and Explanation

An advantage of machine learning is that it allows computers to learn to perform tasks that cannot be programmed. Consider the example given at the start of this chapter, where a computer learned to spot when a driver is at risk of falling asleep at the wheel. We did not know the rules that mapped driver behaviour to alertness, but the computer was able to learn that mapping from data. The difficulty was that we still do not know what the rules are. The computer has an ability that we can test, but not fully understand. Some clues can be extracted, such as which variables were used in the final model, but it is not easy to take a neural network and express the rules it follows in a human accessible way. This is a problem when the machine has learned to make potentially life saving judgements concerning how safe you are to drive. It is even more important in modern applications where computers are actually learning to drive!

It is important to understand what algorithms are doing for other reasons too. Data protection laws are increasingly strict on protecting people from being the subject of automated decision making, for example banks need to be able to tell you why you were turned down for a loan. There are two types of knowledge extraction from statistical models that may be required. The first involves interpreting the model and includes activities such as **rule extraction**. It also forms part of the process of variable selection using embedded methods (see section 1.5.2). The second is concerned with **explanation** of a single prediction, which is the ability to complete the sentance that starts "*This answer was given because ...*". When the rules governing a model are many and complex, it is easier to explain each single prediction than it is to summarise the whole model.

Take a moment to consider what it might mean to explain why a particular prediction is made. One way to phrase the question concerns which input variables need the smallest change to make a significant change at the output. Explanation, then, can be considered as sensitivity analysis. For models that can be differentiated, that means calculating the partial derivatives for all the inputs at the current point. Those with the largest gradients have the largest effect on the output at this point in the input space. In non-linear models, these partial derivatives vary across the input space so the output might be very sensitive to one input for one pattern over all the inputs but not sensitive at all to that same variable when the rest of the variables have different values. Looking for an explanation becomes looking for the one (or few) variables that need the smallest change before the output changes.

Consider the fraud detection example we are currently using. Imagine a classifier labels an insurance claim as fraudulent and the insurance company is required to justify the fact that it will investigate the claim further. It cannot just say "*The computer says so...*", but it can look at the sensitivities at the point in input space represented by the claim in question. This requires an algorithm to systematically search for nearby points that change the prediction from $Fraud$ to *Genuine*. Let us say that the accident being reported happened at 11am and there were no witnesses. A local search may reveal that simply changing the number of witnesses to one is sufficient to change the output to *Genuine*, but changing the time of day makes no difference to the prediction. The conclusion must be that the number of witnesses is a sensitive variable at this point and can be offered as part of the explanation.

Later, you will see that some techniques, such as decision trees, have the advantage of being easy to interpret and that others, such as neural networks, do not. So on top of decisions about model accuracy, the data scientist also has to consider whether or not the model needs to explain its answers or reveal a human readable set of rules. Another reason for needing to understand why machines make the decisions they do is to avoid unintentional bias. This time, we are using the word bias in the more human sense - treating one set of people unfairly due to something they have in common. An oft-quoted example is the automated filtering of job applicants based on a model of who has previously been successfully recruited to the company. If a company historically employed more men, then the algorithm will reflect that bias and continue to exclude women. That, of course, is most undesirable.

1.6 Common Machine Learning Techniques

The sections above describe some theory and practice around machine learning methodology. This section describes some of the machine learning techniques themselves. There are a number of properties that any machine learning technique will have, and it is important to understand what they mean. They include:

- **A Representation**, which is the form the model takes
- **Parameters**, which are tuned to shape the function the model represents
- **A Learning Algorithm**, which is used to tune the parameters
- **A Cost Function**, which is minimised by the learning algorithm
- **Hyper parameters**, which determine the details of how the learning algorithm attempts to minimise the cost function
- **Model Bias**, which defines the limits of the representational power of the model
- **Estimation Bias**, which controls how well the parameter tuning fits the training data
- **Regularisation Methods**, which control the level of model and estimation bias
- **Data Preprocessing Requirements**, which improve the performance of the technique
- **Assumptions** about the data being modelled
- **Suitabilities** for performing different kinds of task
- **Interpretability**, which measures how easily human understandable knowledge can be extracted from a trained model or explanations for predictions can be generated.

The following sections describe a number of machine learning techniques in terms of these points.

1.6.1 Multiple Linear Regression

Multiple linear regression assumes a linear relationship between a vector, X and a scalar, Y. It assumes that each variable, X_i in X has an influence on Y that is independent of any other variable in X. The linear model predicts the expected value of Y at each point in X. For example, the variables in X might be $Income$, $NumberOfChildren$, $HouseValue$ and $MaritalStatus$ and the output Y, might be $AnnualSpend$. A multiple linear regression model assumes that each of the inputs has an effect on annual spend that is independent of the other inputs and could help a company predict the average annual expected spend for individual customers with a given profile. Note the last input, $MaritalStatus$ is nominal and would be encoded using the 1-of-k method described on page 19.

Representation A weighted linear combination of X is used to predict the expected value of Y

$$\hat{f}(X) = \beta_0 + \sum_{i=1}^{n} X_i\beta_i \tag{1.19}$$

where the β parameters define the independent contribution of each variable in X. $\hat{f}(X)$ is an estimate of the average value of Y at the input point defined by X. For simplicity of notation, let $X_0 = 1$ and use vector notation so

$$\hat{f}(X) = X.\beta \tag{1.20}$$

The vector X can be the values of the input variables themselves or a new set of feature variables derived from the inputs. For example, if the input variables to be modelled are $V_1 \ldots V_p$, then a coefficient could be calculated for every product $V_i V_j$ to allow the model to take pairwise interactions into account. In theory, every interaction among variable subsets of all sizes up to n could be modelled but there are 2^n such interactions, so in practise the way that variables are combined to create the input features needs to be managed. Small feature sets are desirable for reasons of parsimony, efficiency and due to limitations imposed by small data sets. Most text book examples and many real world applications of multiple regression do not involve derived features, they simply map the single inputs onto the output.

Learning Algorithm There are a number of learning algorithms for multiple linear regression models. The most common is ordinary least squares (OLS), which is defined by its cost function

$$C = \frac{1}{2}\sum_{j=1}^{m}(y_j - \hat{f}(x_j))^2 \tag{1.21}$$

Let $\mathbf{X}$ be the $m \times (n+1)$ matrix of training data inputs and $\mathbf{Y}$ be the m-vector of target outputs, then from equation 1.20,

$$C = \frac{1}{2}(\mathbf{Y} - \mathbf{X}\beta)^T(\mathbf{Y} - \mathbf{X}\beta) \tag{1.22}$$

and C is minimised where the derivative, $\frac{\partial C}{\partial \beta_i} = 0 \ \forall i$. Solving this gives a least squares estimate for β of

$$\beta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y} \tag{1.23}$$

Stochastic Gradient Descent (SGD) [**?**] may also be used to estimate the parameters in a linear regression model. It is an iterative approach to regression learning, which descends the error function in small steps in response to one training sample at a time, rather than building a matrix like OLS. Each step is guided by the derivative of the error function, a learning rate that restricts the size of the step and an optional regularisation term. Regularisation terms are considered in section 1.6.1 and this section considers the simpler case with no regularisation.

At each step, a single observation (x, y) is taken from the data and a predicted output, $\hat{f}(x)$ is calculated using the current set of parameter values, β. The change an individual parameter, β_i, is calculated from the derivative of the cost function, $C(\hat{f}(x), y)$:

$$\beta_i \leftarrow \beta_i - \eta_t \frac{d}{d\beta_i} C(\hat{f}(x), y) \tag{1.24}$$

where $0 < \eta_t < 1$ is a learning rate that can either be fixed to a constant value or reduced over time. When the cost function is the least squares, $C(\hat{f}(x), y) = \frac{1}{2}(\hat{f}(x) - y)^2$, the weight update becomes

$$\beta_i \leftarrow \beta_i - \eta_t(\hat{f}(x) - y)x_i \tag{1.25}$$

SGD with the least squares cost function asymptotically approaches the same result as OLS, but can be regularised by early stopping. SGD has a complexity of $O(mnp)$ where m is number of passes through the data, n is the number of training points in the data and p is the number of input variables. As n grows, m may be made smaller. SGD has the advantage when data sets are large that a small number of passes through the data are required and, in extremely large data sets, it may be possible to stop early before all of the data has been processed once (assuming the ordering of the data is not important). This can make SGD more efficient than OLS for large data sets in terms of time and memory [**?**].

Other error gradient descent variations such as batch and mini batch approaches, as discussed in section 1.3.1, can also be applied for learning the parameters of multiple linear regression models.

Bias The assumption that inputs are independent in their impact on the output is reflected in the model structure. None of the inputs interact, they simply add a weighted value to the output. This assumption introduces strong model bias as only linear models can be built. If the relationship being learned is linear too, then this is the right level of bias. If the independence assumption is wrong and variables do need to interact, then the linear model has too much bias and will not be accurate. It is noted above that estimation bias can be introduced by early stopping, but OLS is said to be an unbiased estimator because it introduces no estimation bias. A more controlled form of regularisation is available by changing the cost function from squared error to one of those described next.

Shrinkage Methods

Estimation bias can be introduced into linear models using shrinkage methods, which impose a penalty on the size of the weights (thus shrinking them). This penalty is expressed as part of the cost function. For example, ridge regression [**?**] minimises

$$C = \sum_{k=1}^{m}(y_k - \hat{f}(x_k, \beta))^2 + \lambda \sum_{i=1}^{n} \beta_i^2 \tag{1.26}$$

where $\lambda \geq 0$ controls the amount of shrinkage. The ridge regression solution can be found at

$$\beta = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{Y} \tag{1.27}$$

where $\mathbf{I}$ is the $n \times n$ identity matrix.

Another popular regularised learning rule is the lasso (least absolute shrinkage and selection operator [**?**]), which aims to minimise the cost function

$$\frac{1}{2}\sum_{j=1}^{m}(y_j - \hat{f}(X_j, \beta))^2 + \lambda \sum_{i=1}^{n} |\beta_i| \tag{1.28}$$

where λ controls the degree of regularisation. When $\lambda = 0$, the lasso solution becomes the OLS solution. With $\lambda > 0$ the regularisation causes the sum of the absolute weight values to shrink such that weights with the least contribution to error reduction can take a value of zero.

The lasso weights cannot be found analytically as equation 1.28 cannot be differentiated, but a method called least angle regression (LARS) [**?**] can be used to efficiently calculate the lasso coefficients across the range of λ values. LARS takes a similar approach to forward stepwise regression, but adds variables in a way that is not as ”all or nothing“. As each new variable is added, the model is moved towards the least squares fit of the selected variables and the model residual. At the point where an unused variable is as correlated with the residual as the current model, that new variable is added and the process continues.

Bias and Variance The degree of bias introduced by ridge regression and lasso can be controlled by the λ parameter in equations 1.26 and 1.28, which restricts the size of the coefficients and, in the case of the lasso, has the effect of causing some coefficients to go to zero. Model bias can be controlled by the choice of which variables are combined to form the features, X that form the inputs to the model.

Suitability As the output from linear regression models is numeric, they are best suited to prediction type tasks. The inputs can be numeric, nominal or binary and the main assumptions they make are that the input variables do not interact to effect the output and that the distribution of errors around the output is normal.

Data Pre-processing The results of ordinary least squares learning in a linear regression model are invariant under scaling of the inputs, which means the error would be minimmised to the same point and the function would be the same shape regardless of whether the inputs were scaled or not. Any scaling that is required is performed by the parameters of the model as part of the process of fitting the data.

You may still want to scale or normalise the inputs for other reasons, however. If the input variables have very different ranges, then the parameter values of the model will reflect that in the degree to which they scale. If you have a variable called *Distance* the resulting parameter will be of a different order of magnitude if the measurement is in millimetres or kilometres. It can affect readability if parameter values start with 0.00000. While care is needed when interpreting the meaning of parameter values from their magnitude, it is fair to say that variables with parameter values that are close to zero have less impact on the model output than those with larger parameters. A variable with a parameter that is close to zero needs to be changed by a larger amount than a variable with a large parameter to effect a certain change in the output. Scaling all of the variables so that they fall into the same range means that it makes more sense to compare parameters by their magnitude as for unscaled data a very small parameter value might reflect a very large input range rather than a lack of impact.

When regularisation is introduced, the need for scaling becomes more important. Shrinkage methods such as lasso are not invariant to input scaling so you might get different results from two models, one of which has the inputs scaled and the other of which does not. Variables are affected by shrinkage differently depending on the size of their range so it is best to standardise the input before training. If gradient descent is being used to minimise the cost, the size of the gradient in different directions will be affected by the range of the variables so, again, it is worth scaling.

Assumptions Multiple linear regression assumes that the relationship between the inputs and the mean of the output is linear and that the output data vary about that mean in a normal distribution that has the same variance along its range. Another way to say that is that the errors have a normal distribution with a mean of zero and a fixed variance. We can write all of these assumptions in a single expression:

$$\hat{y} = \beta_0 + \sum_{i=1}^{n} X_i\beta_i + N(0, \sigma) \tag{1.29}$$

which means that the average of the output has a linear relationship with the inputs (that is the $X_i\beta$ part) with a normally distributed error with a mean of zero and a variance that is a constant, called σ.

Interpretability Linear models are easy to understand but care is needed when drawing conclusions about the real world from the parameter values of the model. The parameter values tell you how much, and in what direction, each input variable contributes to the calculated value for the output. The first question to ask is "*Were the inputs scaled?*" because if they were, the parameters refer to the scaled inputs, not their original range. A model where two inputs are *income* and *age* and the output is annual *spend* in a shop will almost certainly have a smaller coefficient for *income* than for *age* if no scaling was done as the *spend* value is likely to be a small proportion of income but could easily be a multiple of *age*. If scaling has been done, the coefficients better represent the relative importance of the variables.

How much can we take from the parameter values? The sign tells us the direction of change that would be seen in the output in response to a change in the input variable. If a coefficient is negative then a change in the input value will lead to a change in the output in the opposite direction. Larger (in magnitude, which means further from zero in either the positive or negative direction) coefficients indicate a larger effect on the output. If X_1 has a coefficient of 0.5 and X_2 has a coefficient of 3, we can say that a change in X_2 will have more effect on Y that a change of the same size in X_1 would. If X_2 represents income and X_1 represents age, then that might suggest that the most important focus for marketing is people on a high income. Whether that is a sensible conclusion or not depends on aspects of the real world outside the model. For example, are there sufficient numbers of people with incomes that are higher than our current customers? Factors such as correlation between inputs and the effects of regularisation mean that interpretations of the coefficients of a regression model need to be taken with a pinch of salt.

1.6.2 K-Nearest Neighbours

Armed with a data set of labelled examples from a number of different classes and the task of assigning a new, unlabelled example to one of those classes, what is the simplest approach you might take? One simple method is to look through the labelled data and find the class of the example that is closest to the example to be classified (we will call this the query point). That would work to a degree, but one problem would occur at points where there are a very small number of examples of one class in an area that is overwhelmingly populated by examples from another. The desired classification should be the majority class in the locality around the example to be classified, not just the single closest one. The solution is to find in the training data a number (k) of the closest points to the query point and find the most common label among them. This approach is known as k-nearest neighbours (KNN) classification.

Bias and Variance K-nearest neighbours has a single hyperparameter that can be used to control bias—the value of k. When $k = 1$, as in the first version of the method described above, the bias is low, but variance is high. Increasing k adds bias but reduces variance as it smooths over local anomalies in the class labels. Making k too large introduces too much bias and reduces the accuracy of the technique. Ultimately, if $k = N$ the classification for every pattern will be the same: the label of the majority class in the data set. In this last case, bias is too high, but variance is very low as the majority class will rarely change across different random samples.

Data Preparation The numeric variables used for KNN must be scaled or standardised so that those with a larger scale do not contribute more to the distances than others. Nominal variables can be handled without needing to recode them with dummy variables but should not have too many distinct possible values as that can lead to examples with the same value being sparse and far apart. Arranging values into a small number of similar groups can treat this problem. The data can all be pre-processed once and stored ready for comparison or processed each time a distance is calculated (which is less efficient but removes the need for a duplicate of the data in pre-processed form). The query point must also undergo the same process before it is compared with the stored data.

Representation KNN does not model the data, it just uses the data in their raw form, calculating distances for each query. A common distance measure for numeric data is the Euclidian distance between a single data point, X and a query vector, Q, which is calculated as

$$d(Q, X) = \sqrt{\sum_{i=1}^{p} (x_i - q_i)^2} \tag{1.30}$$

where p is the number of variables being compared. The algorithm can be sped up a little by ignoring the square root as taking the square root of two distances does not change which is the larger. Values from a nominal variable are compared using a distance measure of $(x_i - q_i) = 0$ if the two values being compared are the same (i.e. $x_i = q_i$) and $(x_i - q_i) = 1$ if they are different. One reason for scaling the data is to ensure that the distances that contribute to the sum in equation 1.30 are similar for numeric and nominal data. A measure of confidence can be associated with each prediction based on the distribution of labels in the nearest k data points. It is also easy to check where there is disagreement for different values of k for the same query point.

Figure 1.11 shows an example with two variables and two classes. Note that it is easy to spot the nearest neighbours by eye, but an algorithm has to calculate the distance for every data point in the set to find the k nearest. The disadvantages of KNN are that it requires all the data to be available whenever a classification needs to be made and the distance between each data point and the query point needs to be calculated to produce an answer. This can be time consuming for large data sets.

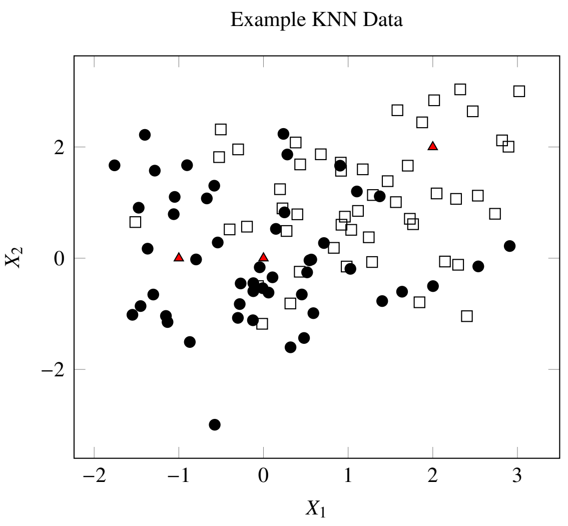


Figure 1.11: A simple two dimensional data set with inputs X_1 and X_2 and two class labels, shown as squares and circles. Three new input patterns are shown as triangles. As an exercise, classify each of them with $k = 1$ and $k = 4$.

It is unusual for a KNN solution to be put into production—it is too slow and memory intensive for that—but it can be useful to use KNN to establish a baseline of performance that other techniques are compared with. Having only one hyperparameter to explore makes the process of establishing that baseline quite simple. The following sections describe some methods that replace the need for storing the entire data set with a statistical model of the data, which is built to reflect the classification task at hand.

1.6.3 Logistic Regression

Linear regression, as described in section 1.6.1 learns a linear relationship between a vector of inputs and a continuous numeric output and assumes that the errors around the outputs are normally distributed. What happens if the output is nominal and we want to perform classification using regression techniques? Let us start with the easiest case, a binary classifier such as fraud detection where there are two output classes. You might be tempted to encode the output as a binary variable where $Fraud$ is represented as one and *Genuine* as zero. You could then perform standard regression in which the output is higher the more fraudulent the inputs look. The practical problem with this approach is that you cannot treat the output as a probability because it could go negative or higher than one. Statistically, the problem is that the errors will not be normally distributed.

The relationship between predictors and the probability of an outcome often displays diminishing returns as the probability approaches zero or one. If I offer random students the chance to help in a class in return for money, they will agree with some probability, which depends on the amount of money on offer. If I start at £10 for an hour, some students will agree and others will not. If I increase the offered amount, the probability of students agreeing will increase sharply at first but subsequent increases in pay will not produce equivalent increases in the probability of a student agreeing. If the probability is 0.5 at £10, then adding £5 to the offer might double to uptake to 0.8, but adding another £5 to the pay will not double the probability again. It might only move from 0.8 to 0.9. In other words, the relationship between pay and probability of accepting the work is non-linear. A better approach is to think in terms of odds, which double with a constant increase in pay, from 1:2 to 1:4 and then to 1:8 etc. In other words, there is a logarithmic change in odds in response to a constant change in pay. This log-odds function has a name: the **logit** function and it plays an important role in logistic regression where the goal is to build a regression model that maps input variables onto a variable representing the log-odds of belonging to a class.

Representation The notion of modelling a response to a constant change in input variables is familiar from the simple linear regression model. We assign a parameter to each input that determines the size of the impact on the output that results in a small change in the input. Rather than calculate the impact on the output variable, Y, we want to calculate the impact on the log-odds of Y being in one class over the other. In this example, we want to know the log-odds of a student agreeing to work as a function of pay. The output variable, Y can take values Yes or No, and the input X is continuous numeric. Let the probability of a student working for a pay level of X be $P(Y = Yes|X)$.

Let us define a variable, $\hat{Y}$ that represents the expected value of the probability of the target event, so

$$\hat{Y} = P(Y = Yes|X) \tag{1.31}$$

and the log odds of that event are linear in the parameters of the model

$$\log\left(\frac{\hat{Y}}{1-\hat{Y}}\right) = g(\hat{Y}) = \beta_0 + X\beta_1 \tag{1.32}$$

where g is the logit function. Estimating β_0 and β_1 from data (which we will address in the next section) provides a model that calculates the log odds of an instance with a given value for X belonging to the class $Y = Yes$. This is not the usual way of reporting class membership—probabilities are more common so we need to convert from log odds to probabilities by inverting $g(\beta_0 + X\beta_1)$ so that

$$\hat{Y} = g^{-1}(\beta_0 + X\beta_1) \tag{1.33}$$

where g^{-1} represents the inverse of the link function and can be written as

$$\hat{Y} = \frac{1}{1 + e^{-(\beta_0 + X\beta_1)}} \tag{1.34}$$

which is known as the **logistic** function and is illustrated in figure 1.12. The output of equation 1.34 varies between zero and one across the range of X and represents the probability of X belonging to the class where $Y = Yes$.

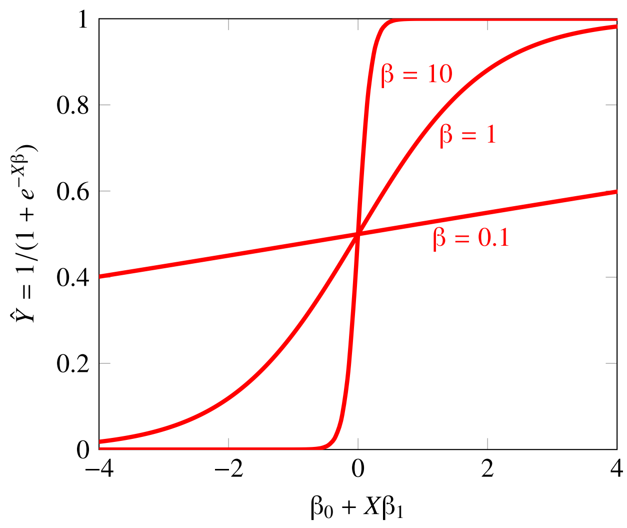


Figure 1.12: The logistic function $\hat{Y} = 1/(1 + e^{-X\beta})$ for different values of β.

Logistic regression is an example of a **generalised linear model**, where a linear combination of the input variables is combined with a **link function** to represent a chosen distribution at the output.

Parameter Estimation **Log-likelihood** is used as the cost function for logistic regression and the parameters that minimise the cost cannot be solved analytically like they can for multiple linear regression, so must be estimated using an iterative search. A common approach to binary logistic regression parameter estimation is **iteratively reweighted**

least squares (IRLS), which iteratively attempts to solves a least squares regression of $g(Y)$ on X. This is the equivalent of a maximum likelihood estimate found using Newton's method.

To run IRLS, the two output class labels are replaced with a 1 for one class (the *positive* class) and 0 for the other class. With X being a single input vector, the predicted probability that X belongs to the positive class is

$$\hat{Y} = \frac{1}{1 + e^{-(X\beta)}} \tag{1.35}$$

To understand the IRLS algorithm using matrix notation (which is common in most textbooks on the subject), you need to construct a set of matrices which are used to build a series of regression models, each weighted by the results of the previous iteration. They are:

$\mathbf{X}$ is the design matrix, which consists of the entire input data organised with one row per data point. It also contains an additional first column full of the value 1, to be used to estimate the constant term in the coefficients, β_0.

$\mathbf{Y}$ is a vector containing the target outputs for the model coded so that one class takes the value 0 and the other class is coded as 1.

β is the vector of coefficients that are being estimated by the model

$\mathbf{p}$ is the vector of predictions of the output probability, $\hat{Y}$ for each input in $\mathbf{X}$.

$\mathbf{W}$ is a square matrix with the same number of rows as the design matrix in which every entry is zero, except the diagonal from top left to bottom right, which is used to reweight the regression at each iteration. Its leading diagonal contains $p_i(1 - p_i)$ from $\mathbf{p}$, which is re-calculated at each iteration.

Having calculated the matrices for each iteration, the coefficient update step is

$$\beta \leftarrow \beta + (\mathbf{X}^T\mathbf{W}\mathbf{X})^{-1}\mathbf{X}^T(y - p) \tag{1.36}$$

1.6.4 Decision Trees

Decision trees are a visually appealing method of representing a regression or classification function by using the branches of a tree structure to split the input space into a set of contiguous sub regions, each of which has a local class or regression model.

Classification Trees

In the game *20 questions*, one player tries to guess the identity of an animal that the other player is thinking of by asking questions with yes/no answers. It pays to start with general questions that rule out large sections of possible animals whatever the answer may be. You might start with is "*Is it a mammal?*", then if it is not, move onto ruling out reptiles or fish, and so on. You do not start with "*Is it an elephant?*" because that has a very large probability of ruling out only one possible answer and a tiny probability of being correct. Whether you think about it or not, you are trying to reduce the level of uncertainty you have about the answer at each step and the possible answers to your questions form a binary tree. This is very similar to the approach taken by decision trees, which take a *divide and conquer* approach to assigning a class label to a vector of descriptive variables.

Representation A classification tree is represented as (you guessed it) a tree structure with a variable at each node, a possible value (or range of values) that the variable may take at each edge and class labels on the leaf nodes. The single root node contains the variable which removes the most uncertainty about the class label and then divides the data set into smaller subsets, one for each value that variable can take. Each of those values forms an edge out of the node and leads to the next node, which contains the variable that removes the most uncertainty about the identity of the class in the subset of the data at that node. The process repeats with the tree growing in depth and the quantity of data contributing to each decision diminishing. Figure 1.13 shows an example tree structure for classifying animals. Obviously, it is simplified and not a complete taxonomy!

It should be easy to see from figure 1.13 how a classification is made from a single example input. Start at the root node (*Mammal*) and follow the edge that matches the value in the example to be classified. Continue down following the path dictated by the example at hand until a leaf node is reached, where the classification is identified. The leaf node can also contain other data such as the number of input examples in the training data that led to this leaf and the

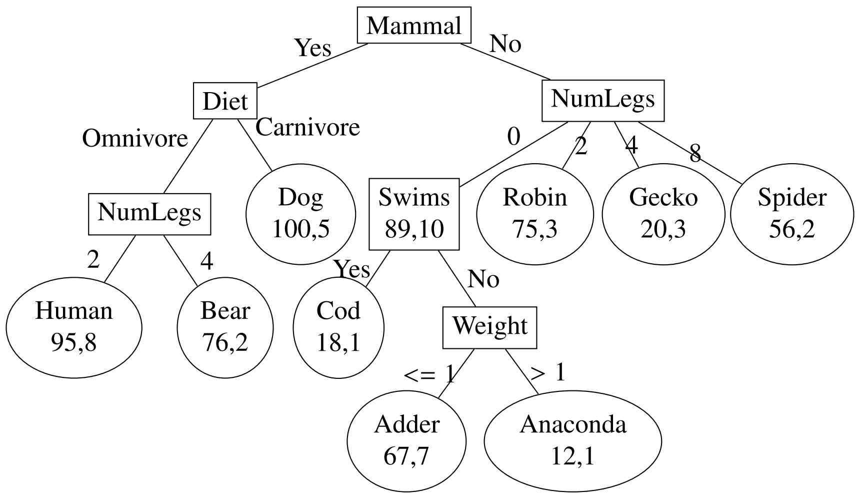


Figure 1.13: An example classification tree on a set of animal labels. Decision nodes are rectangular, class nodes are oval. Note that not all of the variables are used in every classification and that *NumLegs* is used twice. Also, *NumLegs* is a discrete variable so it has once branch per value but *Weight* is continuous so it is split in two. The leaf nodes contain two numbers: the number of times in the training data the inputs led to this node and how many of those examples were **not** labelled in agreement with the node.

number of times the classification matched that of the leaf. Note also that not all classifications need all the variables as not all the leaf nodes are at the same level.

Learning Algorithm and Cost Function The job of a decision tree is to classify input patterns correctly, so you might imagine that a good cost function would be the number of misclassified training instances. That is one possibility, but it can lead to trees failing to find a split in certain circumstances, so one of two other cost functions are generally used. They are described in this section along with the algorithms that use them. These algorithms build trees in a greedy fashion, adding a single node at a time to a tree that grows as the algorithm proceeds. That means that each new node is chosen so as to maximise the reduction in the cost function. That involves evaluating all of the variables and picking the best. To evaluate a variable, X_i the distribution of predicted output classes across the values that X_i can take is assessed.

Consider a variable, X_1 that can take values a, b, or c and another, X_2 that can take values d or e. Which should be chosen for the first node of a tree? The values each variable can take partition the training data, in this case into three (X_1) or two (X_2) different subsets. Each subset will contain a certain distribution of the output class. If the split is sufficient to build a perfect classifier, each subset will contain only examples of a single class (we say it is perfectly **homogeneous** or **pure**). The closer to a uniform distribution there is in a subset, the less pure it is and the poorer the choice of that variable for classifying the data. Imagine the classification task is binary—there are two output classes. If we split on X_1 we might find that all but one of the cases where $X_1 = a$ lead to class 1 and all but two of the rest of the outputs (those where $X_1 = b$ or $X_1 = c$) are class 2. On the other hand, X_2 splits the labels 60/40 when $X_2 = d$ and 70/30 when $X_2 = e$. Clearly, X_1 is a better choice for the split.

How is the purity measured? Two common approaches are based on information gain or Gini impurity. Both measure how homogeneous the class labels are as a result of splitting on a variable. When information gain is calculated, we talk of reducing uncertainty in the distribution and when Gini is measured, we talk of reducing impurity.

Tree tree learning algorithms such as ID3, C4.5 and C5 attempt to minimise the uncertainty about the output classes given the input variables. Uncertainty is measured using Shanon's information theory. Remember that uncertainty about a variable is measured by its **entropy**, which is calculated from its probability distribution. The entropy of the output class variable Y is the uncertainty you would face if you had to guess the label associated with a class picked at random from its distribution. If the distribution was 99 dogs and one cat, you would face less uncertainty than if the distribution was 50 dogs and 50 cats. The further from uniform a distribution is, the lower its entropy will be. The formula for calculating the entropy of Y from the set of j class labels $y \in \mathbb{Y}$ is

$$H(Y) = -\sum_{y\in\mathbb{Y}} P(Y = y)\ln P(Y = y) \tag{1.37}$$

where $y \in \mathbb{Y}$ iterates over the possible values that Y can take.

The related measure, **conditional entropy**, written $H(Y|X)$ tells us the uncertainty about Y that remains when the value of the variable X is known. If X has some ability to predict Y then the conditional entropy of Y given X will be lower than the entropy of Y alone. This difference is known as the **information gain** on Y of knowing X. Conditional entropy where Y can take values in $\mathbb{Y}$ and X can take values in $\mathbb{X}$ is calculated as

$$H(Y|X) = \sum_{y\in\mathbb{Y},x\in\mathbb{X}} P(x,y)\ln\frac{p(x)}{p(x,y)} \tag{1.38}$$

and information gain is calculated as

$$Gain = H(Y) - H(Y|X) \tag{1.39}$$

Now the tree building algorithm can be described. The first step finds the single variable from the input vector that provides the greatest information gain for Y. This is done by first calculating $H(Y)$ and then calculating $H(Y|X_i)$ for every input variable. In the tree shown in figure 1.13 we can see that the variable is *Mammal*, which can take values of Yes or No. This tells us that knowing whether or not an animal is a mammal removes the most uncertainty about its identity in the data set used to build this tree.

The next step involves splitting the data into groups—in this case, two groups: mammals and non-mammals. The same question is then asked of these two subsets of the data: which variable reduces the entropy of Y most? The answer might be different for each group, so the tree can diverge in terms of the variables it uses. In figure 1.13 it is clear that the best way to reduce entropy about the identity of mammals is to look at the variable *Diet* but the best way to continue for non-mammals is with *NumLegs*. Note that the number of legs is a discrete numeric variable with possible values of 0,2,4,8.

The process continues until either every node has zero entropy, which means that all the data at the node belongs to the same class or the number of data points at the nodes is less than a predetermined limit, set to enforce bias.

Another tree building algorithm, CART (Classification and Regression Tree) uses Gini purity to choose the variable at each node. Gini impurity for the output variable Y is calculated as

$$H(Y) = \sum_{y\in\mathbb{Y}} 1 - P(Y = y)^2 \tag{1.40}$$

Both approaches are popular, so is there a reason to choose one over the other? Entropy measures require logarithms to be calculated, which is a little more time consuming than the squaring required for Gini, but the difference will rarely matter. Raileanu and Stoffel [**?**] compared the two on a large number of empirical tests and found practically no difference in performance.

Bias The classification tree has structure, but no parameters to estimate, so the only form of bias available is model bias. You could remove model bias entirely by building a tree in which there was a route for every possible combination of values across all the variables. This would present several problems. Firstly, it would grow exponentially in size with the number of possible values the variables could take. It would also need a massive amount of data to avoid over fitting. For these reasons it is essential that the tree has a healthy dose of model bias. This means limiting the number of nodes and ensuring that each leaf node represents a split of the data of reasonable size. Quite what size of split is reasonable can be difficult to establish for a particular data set, but a mixture of experience and cross validation will help.

Numeric variables are handled in two different ways, depending on whether you flag them as being discrete or continuous. Discrete variables, like *NumLegs* in the animals example split the data into one group per value, and only have branches for values found in the data (0,2,4,8 in this example). Treating numeric variables that have many values as discrete causes many branches, each one with a smaller proportion of the data, and so encourages over fitting. The same problem occurs with nominal variables with many different values. That is one reason why they need to be identified and fixed during data pre-processing (as described in section 1.4.1). Continuous numeric variables can take any value, so they cannot have one branch per value. Continuous variables are split in two at each node and if further splits are needed, they occur at nodes further down the tree. Possible split locations are found by sorting the data by the variable to be split and identifying points where the output class changes from one point to the next. These are candidate split points and the information gain for each is calculated to find the best. Some implementations split on the lower value at the split and some at the midpoint between the two. If you have reason to believe that you know where multiple split points might be for a continuous variable, you must re-code the variable by hand and label it as discrete for the learning algorithm. For example, you might want to use age bands 0-16, 17-25, 26-35, etc. The tree would then have nodes for *Age* with one branch per sub-range.

Regression Trees and Model Trees

If the output variable, Y is continuous, then the task is prediction and a regression tree can be used. Regressions trees are very similar to classification trees, except they have numbers at their leaf nodes. Tree models partition the data into increasingly smaller blocks as the tree is descended and each leaf node represents a single block. In a regression tree, the leaf contains a constant value, the mean of the output variable in the block it represents. In a linear model tree, the leaf contains a multiple linear regression model of how the variables in that block are related to the output variable. For example, a route through the tree that takes in variables X_1, X_3, X_6 would lead to a constant $\bar{Y}$ in a regression tree and an equation $Y = \beta_0 + \beta_1 X_1 + \beta_3 X_2 + \beta_6 X_3$. Both the mean, $\bar{Y}$ and the equation parameters, β are estimated during learning from the data in the subset of the data defined by the branches from the three nodes covering X_1, X_3, and X_6.

Rather than information gain, regression and model trees are usually built using squared error as the cost function. For regression trees, where the leaf contains an estimate of the mean, the squared error is just the variance around that mean, so splits are chosen to minimise the variance of Y in the resulting splits. Apart from that, the tree building algorithm is the same for classification trees and regression trees. The next section looks at some specific tree building algorithms in a little more detail.

Decision Tree Algorithms The popular tree building algorithms such as ID3 [**?**], C4.5 [**?**] and CHAID use a greedy approach to adding nodes to the tree. In their basic form, they add the node that provides the largest single reduction in the cost function at each step. The resulting tree may not be optimal due to interactions among variables. It might be that variable X_1 is chosen as the root node when considered in isolation, but once X_2 and X_3 are added, it might turn out that X_1 is no longer required. This is an example of a greedy approach to minimising a cost function and solutions are therefore not guaranteed to be global minima.

Model bias in a decision tree can be controlled in a number of ways. The first is **early stopping**, in which the tree building algorithm stops before all the possible splits have been made. The usual criterion for stopping is when a node has less than a minimum number of data points in the split it represents. That route through the tree is not persued further, and the algorithm as a whole stops when all nodes are leaves.

An alternative to early stopping is to use **pruning** after the tree has been built. Pruning takes a trained tree and removes decision nodes from the lowest levels with reference to the validation score. If removing a node does not increase the cost on the validation data, then that node is removed. The removed node and the subtree below it are replaced by a single leaf node whose class is the most common class among the leaf nodes in the subtree that was removed.

The popular data mining software package, Weka, has an implementation of C4.5 that it calls a J48 tree. Visualising a tree in Weka uses the method of reporting the number of data points and incorrect classifications at the leaf nodes shown in figure 1.13. It also allows you to choose a minimum number of objects in the leaf nodes.

Interpretability An important advantage of a decision tree is that the entire structure (for small trees, at least) can be viewed and understood by a human. This allows general explanations to be generated concerning which variables are more important (the ones near the top of the tree) and which routes through the tree account for most of the predictions made (those with the largest numbers at their leaf nodes). As different routes through a tree often involve different variables, it is also possible to read explanations for a single decision straight from the tree by reporting which variables were used and in which order. There are many stories of companies using decision trees that are less accurate than multilayer perceptrons because of their ease of interpretation. One approach that has been informally reported involves building an MLP, then using the MLP to generate a new data set which is used to train a decision tree that mimics the MLP but is easier to interpret. We will meet this idea again in section 1.11.4 on combining ensemble models.

1.6.5 Multilayer Perceptrons

Multilayer perceptrons (MLPs) are a type of neural network [**?**] that perform a similar task to linear regression models without the restriction of the assumption of linearity. They do not assume that input variables have independent effects on the outputs. They can model the way interactions between input variables influence the output. This is achieved by chaining regression functions in a feed forward process through what are known as hidden layers. The hidden layers encode features of the data as non-linear functions of weighted sums of either the input variables or existing features from lower down the chain.

Representation Figure 1.14 shows the structure of an MLP with one hidden layer. Each node represents a function. Those at the bottom are the input nodes, which take a single input value and output that same value, unaltered. The other nodes receive a weighted sum of the outputs from connected nodes below, pass them through a function known as the **activation function**, and output a single value, which is passed as one of the inputs to every connected node in the layer above. At the output layer, this forms the output of the function. All layers, including the output can contain more than one node, allowing mappings from and to many variables. The output layer can represent continuous variables or category labels, so MLPs can be used for prediction or classification. All neurons except those on the input also receive a constant input, known as the bias, on a weighted connection. The bias plays a similar role to the constant term in a regression model.

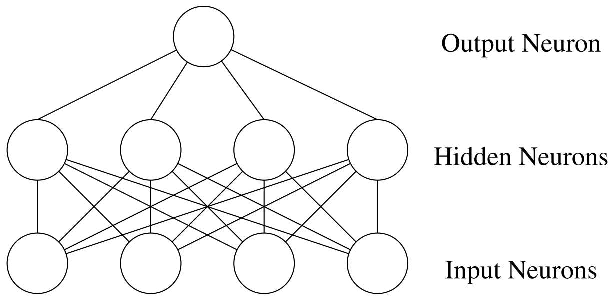


Figure 1.14: A multilayer perceptron with one hidden layer. The MLP represents a function that maps values at its input layer to values at its output. The nodes (circles) contain activation functions and the weighted connections (lines) carry the parameter values that are learned from data.

Each neuron receives an activation, a_i, which is the sum of the products of the output from neurons below and the weights with which they are connected:

$$a_i = \sum_{l \in \mathbf{L}} w_{l,i} O_l \tag{1.41}$$

where $\mathbf{L}$ is the set of nodes in the layer below the one containing node i, $O_l = act_l(a_l)$ is the output from node l and $w_{l,i}$ is the weight of the connection from node l to node i. $\mathbf{L}$ consists of neurons that are either inputs (in the case of the first hidden layer) or neurons in the previous layer plus a single bias neuron with an output set permanently to one.

The activation function associated with neuron i, $act_i(a_i)$ may be linear, where $act_i(a_i) = a_i$ or non-linear depending on which layer it occupies and other design considerations that are touched on below. Common non-linear functions include the following:

The logistic function, whose choice is inspired by the output from logistic regression, described in section 1.6.3, where the output of node i is

$$act_i(a_i) = \frac{1}{1 + e^{-a_i}} \tag{1.42}$$

The tanh, which has a very similar shape to the logistic but is symetrical around zero (which the logistic is not), where the output of node i is

$$act_i(a_i) = \tanh(a_i) \tag{1.43}$$

The rectified linear function is linear for $a_i > 0$ but returns zero when $a_i < 0$ and is calculated by

$$act_i(a_i) = \max(0, a_i) \tag{1.44}$$

Neurons with a rectified linear activation function are called rectified linear units (ReLU). They do not suffer from the vanishing gradients of the logistic and tanh functions and are simpler (and so faster) to compute. When used in a hidden layer, ReLUs create a sparse representation as units with activation below zero have an output of zero and a gradient of zero, so their weights are not changed during learning. There is also a leaky ReLU, $act_i(a_i) = \max(\epsilon, a_i)$ where ϵ is small and positive (say 0.01), which can be used when a zero gradient needs to be avoided for negative activations.

A popular activation for the output layer of classification networks is called the **softmax** and it treats the activations in the layer a little differently from the other functions we have looked at. The output of any single output unit depends on its input and the inputs to all the other units in the output layer. This achieves two things. It ensures the outputs sum to one (like well behaved probabilities) and pushes the output values apart so that differences are emphasised. To understand why, imagine a function that finds the largest output, sets its value to one and sets all the others to zero. This is a so-called winner takes all approach and simply assigns the input pattern to the class that corresponds to the highest scoring output. The softmax is a smoother version of this. It pushes the winner closer to one, but still allows the input to have a non-zero probability of belonging to other classes too. Here is the softmax function, applied to the output layer vector, Y.

$$s(Y_i) = \frac{e^{Y_i}}{\sum_{j=1}^{n} e^{Y_j}} \tag{1.45}$$

which says that the output value of unit Y_i is the exponential of the input to Y_i divided by the sum of the exponentials of all the other units in that layer.

In principle, each node could have a different activation function, but in practice, the inputs have a linear function, the hidden units all have the same non-linear function and the outputs have either a linear or non-linear activation function (linear for regression, logistic or softmax for classification). The output of an MLP when the input is x, is defined as the output of the activation function on its output neurons. In the case of a single output that is

$$\hat{y} = act_l(a_l) \tag{1.46}$$

where a_l is the activation of the single neuron at the output layer and $act_l()$ is the activation function associated with that neuron.

Learning Algorithm Most commonly, an MLP has a fixed structure of weights, the values of which are learned by a gradient descent of a cost function. A common cost function for performing regression is the quadratic:

$$C = \frac{1}{2m} \sum_{j=1}^{m} (y_j - \hat{y}_j)^2 \tag{1.47}$$

where $\hat{y}_j$ is the MLP output calculated using equations 1.46 and 1.41 in response to the input x_j. In order to minimise C, the partial derivative of the cost function with respect to each weight, $dC/dw_{l,i}$ needs to be calculated. This can be done one training example at a time. For a single training data point, the derivative of the cost function with respect to the output from the network is $dC/d\hat{y} = (y - \hat{y})$. The derivative of the error with respect to the activation, a of the output unit is dC/da, and depends on the choice of activation function. Each weight contributes $w_{l,i}O_l$ to a so the derivative $dC/dw_{l,i} = (y - \hat{y})dC/da(w_{l,i}O_l)$. Each weight is changed by $\eta dC/dw_{l,i}$ where $0 < \eta < 1$ is a learning rate that ensures individual weight changes are small. Errors are passed back through the network using the chain rule to allow earlier weights to be updated in a process known as back propagation of error [**?**].

Network and Training Design There are a number of design decisions that need to be made when building and training an MLP. Some concern the network itself and some concern the gradient descent learning algorithm. The decisions to be made are often framed as a set of hyperparameters, which form a space that needs to be searched. Different choices of hyperparemeter values can lead to different functions being learned by the MLP and different levels of error. Commonly considered hyperparameters include the following, which are largely taken from a paper describing deep neural architectures [**?**]. Those listed here apply equally to MLPs.

Number of hidden layers and units MLPs can support many hidden layers but in reality only generally contain a small number. In principle a single layer containing a finite number of nonlinear units and a linear output unit is sufficient to allow an MLP to approximate any continuous function on compact subsets of $\mathbb{R}^n$ [**?**]. However, the theory says nothing about the learnability of the weights and it has been found that adding several smaller hidden layers can be more efficient than training one large one [**?**]. The number of hidden units included in a network controls model bias. Too few units may make the network unable to represent the desired function and too many may lead to over fitting. Other forms of regularisation such as weight decay can compensate for a network with too many units, so the main cost of adding units can sometimes be the increased training time.

Some heuristics have been proposed for choosing the number of hidden units and layers in an MLP but none of them are very reliable and a better approach is to employ a search using cross validation to compare various architectures. This can be done as part of the hyperparameter search (see section 1.5.3).

Activation Function Some of the choices for activation functions are listed above. Hidden units should be non-linear. When performing regression, a linear output is used and for classification, a logistic or softmax activation function should be chosen at the output layer.

Learning Rate Gradient descent learning involves making small adjustments to model parameters (weights in the case of the MLP) to make steps down the error gradient. Making those steps too large causes the error rate to rise and making them too small causes the error to drop very slowly. The size of the changes in the weights is controlled by a learning rate. A good rule is that the learning rate should be the largest possible that does not cause the error to rise [**?**]. Variable learning rates are often used, with the learning rate diminishing according to a chosen schedule or in response to a flattening of the error rate.

Early Stopping An easy method to avoid over fitting with an iterative learning process such as SGD is to terminate the training process before the training error has flattened. The error on a separate validation data set, which is not used to learn the model parameters, can be monitored so that early stopping can take place when the validation error begins to rise. Early stopping can obscure the over fitting effects of other hyperparameter choices, and so is best left out of an initial hyperparameter search, and then used to attempt to improve a chosen configuration.

Momentum In addition to a learning rate, weight updates are often smoothed using a moving average of previous updates. The proportions of the current gradient and of the previous average gradient that contribute to a weight change produce another hyperparameter known as the **momentum** rate. It can vary between zero and one and higher values lead to more smoothing but can slow the learning process.

Training Batch Size Basic stochastic gradient descent makes one weight update per training example, approximating the change across all of the data by making small steps one at a time. Another alternative is to calculate the average gradient across all of the training data in a batch and make a single update to each weight based on a full pass through the data. The batch method makes the steps smoother but can be very slow as each weight update requires a complete pass through the training set. A compromise, known as **mini batch** training, updates the weights using the average error over small batches of training data. This smooths the error descent without slowing the process down to the same extent as a full batch approach.

Weight Initialisation The weights of an MLP cannot be all set to zero before training begins. Their values must be randomised to avoid different neurons sharing the same weight values. Weights are often picked from a uniform distribution bounded by some range, the size of the range being one hyperparameter to explore. A recommended range is between $-r$ and r where $r = c\sqrt{6/(fanin + fanout)}$ and c is 1 for tanh functions and 4 for logistic functions [**?**]. The $fanin$ and $fanout$ values are the number of weights in and out of the unit associated with the weights being set. A simpler rule of thumb is to use random uniform weights over [-0.7,0.7] and standardise the inputs [**?**], as described next.

Data Preprocessing The gradient descent algorithm used to train multilayer perceptrons is more effective if the inputs have been standardised to zero mean and unit standard deviation [**?**]. Nominal variables should be recoded as 1 of k dummy variables As each possible value adds an additional input node and associated weights, controlling the number of possible values nominal variables can take (by grouping, for example) is a way of controlling model bias and overfitting.

Regularisation Early stopping is mentioned above as one method for avoiding over fitting. Other regularisation methods applied to neural networks include the addition of noise to training data [**?**], penalties on weight size such as L1 and L2 regularisation and network structure methods such as dropout. Dropout [**?**] involves randomly ignoring a proportion of the neurons (and connected weights) in a network during training but using all of the neurons when testing. More specifically, during training each neuron is ignored with a certain probability, p, which attempts to turn a single network into the average of many sparse networks. When training is complete, weights are adjusted by a factor of p so that the output at test time is the same as the expected output during training.

Strengths and Limitations Multi layer perceptrons have the capacity to act as universal approximators. In reality, the correct architecture (the number of hidden units and the connectivity pattern among them) to represent a given function needs to be discovered and a learning algorithm must find the correct weight settings. The cost function may contain local minima in which gradient descent methods may become trapped, making the testing of a chosen architecture more difficult.

The ability to represent any function is accompanied by an increased risk (compared to simpler methods) of over fitting. Particular care is needed when training a neural network to achieve the correct trade-off between bias and variance. Another well known problem with MLPs is the fact that the cost function contains local minima or plateaux from which a gradient descent algorithm cannot escape. Training a number of MLPs from different random starting weights can (if the error function dictates it) result in a number of different solutions. This adds complexity to the search for effective hyperparameters as you cannot know whether the error difference between two models is due to their hyperparameter choices or their random starting points.

Interpretability A text book criticism of MLPs is that they are opaque to human eyes. The so called *black box* problem refers to the fact that the weights of an MLP offer very little in terms of human interpretability. It is not easy to extract information about the structure or complexity of the function a network has implemented. A lot of work has been dedicated to extracting rules or insights from multilayer perceptrons [**?**], [**?**], [**?**], [**?**]. Swingler [**?**] used a Walsh decomposition to investigate the structure of MLPs and showed that the complexity of the function being implemented by an MLP varied widely as it learned, demonstrating that simple measures such as the number of hidden units are, at best, a crude indication of function complexity. MLPs can be differentiated so the partial derivative of the output with respect to each individual input can be calculated. This allows the sensitivity of the output to the inputs at the given point to be reported. The function implemented by an MLP is non-linear so the sensitivities at one point may be different from those at another. The gradients can also be used to search for nearby points that change the output significantly, as described in section 1.5.5. This approach was put to good use by Szegedy et al. [**?**] who took images that were correctly classified by a neural network and searched the network for very similar images that changed the output class given by the network. Remarkably, the changes that were needed were imperceptible to a human observer comparing the two images.

Training, Testing and Validation Section 1.3.1 described how achieving good generalisation and avoiding over fitting are at the heart of statistical learning. An important question to be able to answer about a statistical model is "How well will it generalise to unseen data?". To answer the question, a subset of available data is kept aside for testing. This data must play no role in the selection of the model, the choice of training hyperparameters, or the model parameter setting. A single test set provides a single estimate of test error. A better estimate can be obtained by repeatedly training different models on different subsets of the data. A common method for balancing the bias-variance trade-off when training an MLP is to use cross-validation in which the data is partitioned into a number (usually ten, which is the figure used in the rest of this description) of non-overlapping test sets, each comprised of a different 10% of the data. Ten models are then trained, each on the 90% that remains for each test partition. The mean and variance of the error across these models gives a better indication of the likely error on new, unseen data.

When choosing a model and setting the training algorithm's hyperparameters, further division into training and validation data is required. For each combination of hyperparameters, a model needs to be trained and validated. The validation error is used to compare one hyperparameter set with another, but cannot be used as the estimate of model generalisation ability as it played a role in the model building. Cross validation can be used at this stage too (though this can become computationally expensive as part of a grid search) to achieve a good estimate of the differences among hyperparameter sets. Using cross validation at both levels in this way is called nested cross validation.

1.7 Unsupervised Learning

The methods discussed so far have been examples of **supervised** learning, where the variables are either input variables or output variables and a mapping is learned between the two. In unsupervised cases, the data describe a set of measurements that do not fit this input-output pattern. In these cases, we can think of the data as consisting of inputs only and the task to be performed is to organise or characterise those data in some way. Common unsupervised tasks include: **Clustering**, which attempts to divide the data into subsets whose members are similar to each other but different from the members of other subsets; **Probability Distribution Estimation**, which models the probability distribution of the data; and **Novelty Detection**, which is concerned with spotting unlikely patterns in new data from the same source. A density estimation model can be used for both clustering and novelty detection or for other purposes such as generating new synthetic data with the same distribution as the original data. Novelty detection and clustering can also be achieved with other methods, as the following sections explain.

1.7.1 K-Means Clustering

Given a data set and a requirement to describe it in a compact model, you might choose to calculate the average value for each variable. For example, given a set describing the weights and heights of a sample of people, you might find that the average person is 160cm tall and weighs 75KG. That is fine if height and weight are normally distributed and independent. Now imagine your sample contains basket ball players and their five year old sons. Height and weight will not be normally distributed in that data set. There will be two peaks in the distribution, one around the adults and one around the children. A single mean value will be meaningless as it will fall between the two sets and actually describe none of them. Figure 1.15 illustrates the point for *height*. What is needed is two separate mean points—one for the children and one for the adults. How do we separate the data though? Assume the measurements are not labelled—we do not know which measurements came from which people. More generally, there are problems with more variables and an unknown number of means. It is impossible to plot such high dimensional data and separate it by eye. An analytical method is needed. The task of finding multiple means is called **k-means clustering**.

Representation K-means clustering aims to split a data set into k clusters, each characterised by a multivariate mean. For example, with three variables and $k = 2$, the resulting model is characterised by two triples: $(\bar{X}_1^1, \bar{X}_2^1, \bar{X}_3^1)$ and $(\bar{X}_1^2, \bar{X}_2^2, \bar{X}_3^2)$ where $\bar{X}_i^m$ is the m^{th} mean of the i^{th} variable. These mean points are referred to as **centroids** and are denoted $M_1 \ldots M_k$. The set of data points that is closer to M_i than to any of the other centroids is denoted S_i.

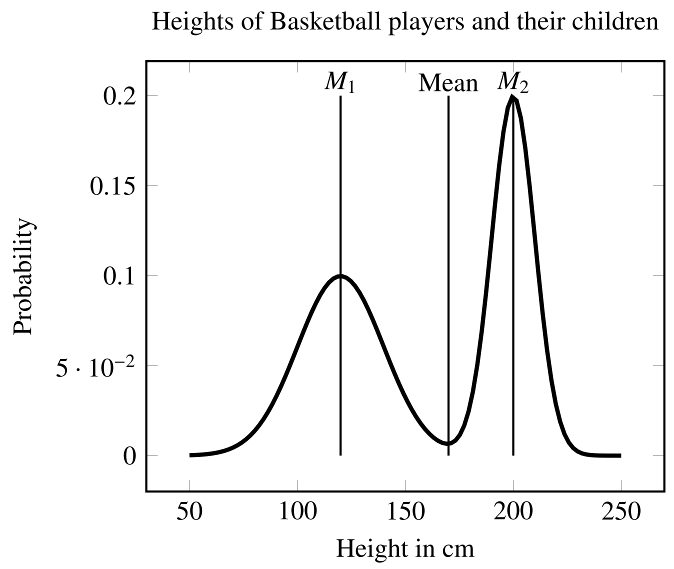


Figure 1.15: A distribution with two peaks, in this case the height of people from a group of basketball players and their five year old children. The middle vertical line shows the location of the mean, which falls far from where most of the data lie. A better characterisation of this data requires two mean values: $M_1 = 120$cm and $M_2 = 200$cm.

Cost Function and Learning Algorithm The most common method for achieving k-means clustering was proposed by MacQueen [**?**] and is named after him, though you will see many text books simply refer to the k-means clustering algorithm. Section 1.3.1 describes how the mean of a set of data minimises the total squared difference between each data point and the estimate of the mean. The cost function for k-means clustering uses the same measure: a sum of squared distances, which equates to the total Euclidean distance between each data point and the centroid closest to it.

$$\sum_{i=1}^{k} \sum_{x \in S_i} \| x - M_i \|^2 \tag{1.48}$$

where $x \in S_i$ iterates x over the values in the data that are closest to the centroid M_i. The learning algorithm finds a local minimum in equation 1.48 for a given value of k by an iterative process that involves assigning data points to the closest centroid and then moving the centroids by calculating the average of the points they have been assigned. Algorithm 2 provides the pseudo code.

Algorithm 2 MacQueen's K-Means Clustering Pseudo Code

```
Let $\mathbf{x^1} \dots \mathbf{x^N} = \mathbf{D}$ be the training data set
Choose a value for $k$
Let $M_1 \dots M_k$ be the centroids
Initialise each $M_1 \dots M_k$ to one randomly chosen data point
repeat
    Allocate every $x \in D$ to the nearest set, $S_i$
    Calculate each $M_i$ as the means of the variables in $S_i$
until No centroids moved in the last pass
```

In algorithm 2, the allocation of data points to sets is known as the **assignment step** and the calculation of the means is called the **update step**. A number of different distance measures are available for finding the nearest centroid in the assignment step. The most common is Euclidean distance, as defined in equation 1.30.

Hyperparameters and Bias and Variance There is one hyperparameter associated with the k-means process: the value of k. When $k = 1$ the algorithm simply finds the mean of each variable in X. As k grows, the complexity of the model grows and the potential for achieving smaller values from the cost function of equation 1.48 grows too. The total cost can be reduced to zero if $k = N$ because there would be a single cluster for every data point. Smaller values of k introduce more bias than large value of k. Setting k too high can produce small clusters that capture outliers or that split data that should be in a single cluster into two. Setting k too small can merge points that should be kept apart (similar to the single mean example of figure 1.15). Sometimes (as in our basketball player example) there is a correct value for k but often the choice is either made from pragmatism or an observation of the number of examples in each cluster.

Clustering algorithms are often used in marketing applications where it is desirable to divide a group of customers into a small number of different 'tribes', each of which can be characterised by their average representative. Different marketing messages can then be created for different tribes and mailshots, web pages or adverts can be targeted accordingly. In my experience, the marketing team will have a preference for the number of tribes they want, so we set k accordingly. Evidence that k is too large can be found if some of the clusters have a very small number of members (closest points), particularly if their centre is close to another centroid. As the algorithm has a random start point and makes iterative steps downhill in the cost function towards a local minimum, it is sensible to run it several times on the same data and compare the results. It may even be (as in the marketing example) that two potential solutions may be compared by a human observer and a choice made from preferences that are not part of the cost function.

Data Preparation As the k-means algorithm is distance based, variables should be scaled or standardised before learning begins. This is to avoid variables with wider ranges having a disproportionate effect on the distances from the data points to the centroids.

Limitations We have already discussed the need to choose k carefully and the fact that any single solution may be one local optimum among several. Additionally, the algorithm makes the assumption that the clusters are spherical as the distances in all directions are treated the same. In some data, the clusters may take different shapes, which k-means is not able to capture.

1.8 Density Estimation

Clustering, as described in section 1.2.2 is concerned with allocating a data point, x to one of several clusters, where the cluster boundaries are derived from a sample of data. The methods described are examples of **hard** clustering techniques, not because they are difficult, but because they define hard boundaries between the clusters. You might consider the ability to assign a point to clusters based on probabilities instead, allowing a point to belong to M_1 with a probability of 0.8 and M_2 with a probability of 0.2. This approach is known as **soft** clustering and is concerned with calculating the probability that a data point belongs to a cluster: $P(M_j|x)$. It is directly related to the task of **density estimation**, which builds a model that evaluates the probability of x being drawn at random from the distribution that generated the training data: $P(x)$. The challenge in these two tasks is to characterise the probability distribution (also known as the **density function**) that most likely generated the training data.

The task is unsupervised. Given a data set, $\{x_1 \dots x_N\}$, that was generated by some unknown distribution, $f(X)$, the task is to build a model, $\hat{f}(X)$, which takes a data point as input and generates the probability density associated with it as output. Assumptions about $f(X)$ need to be made and shaped by the data as the true from of the function is unknown. When the variables are continuous, the function $\hat{f}(X)$ generates a **density** estimate, not a probability value. This is because the density function can only calculate the probability of a value falling in a given range, which is the area under the distribution curve between the two points that delimit that range. On a continuous scale, there are an infinite number of values x could take, so each is infinitely small in its range. It does not make sense to ask for the probability that $X = 0.5$, only to ask for the probability that X falls in a small band of values around 0.5. For many applications, however, knowing the exact probability is not required. It is enough to compare densities. If we want to know the most probable of two data points, or the most likely cluster that a data point belongs too, it is enough to compare the density values. Figure 1.16 illustrates the point.

Histograms

The simplest way to characterise a distribution is to split the input range into sub-ranges (often known as **bins**) and count the number of data points in each bin. Let the k bins be $B_1 \dots B_k$ and the count of the number of data points in

Figure 1.16: A probability density function. The shaded region represents the probability of an observation picked at random having a value in the range x_1 to x_2. This is calculated by integrating the density function in the given range. Density measures themselves are not probabilities, but they can be compared so that it is true to say that the probability of picking x_1 is greater than the probability of picking x_2.

B_i be $c(B_i)$. If there are N samples in the data then the probability of a random sample being drawn from bin B_i is

$$f(x \in B_i) = \frac{c(B_i)}{N} \tag{1.49}$$

The bins are often defined to have equal width but they do not need to be. They are defined by a lower bound, l_i and an upper bound, u_i so that x is in B_i if $l_i \leq x < u_i$. As the domain of the values has been discretised, the function does actually generate a probability, so $P(X) = f(X)$ in this case. That is one advantage, but there are a number of disadvantages. Firstly, the function is not smooth. It jumps at the bin boundaries and has no discriminative power within them. If x_1 and x_2 both fall into bin C_i then $P(x_1) = P(X_2)$ according to the model. The choice of bin sizes and the starting value (the smallest value in the data sample) also affect the shape of the histogram and the apparent distribution it shows.

There is a bias-variance trade-off, even with something as simple as building a histogram. Bias is controlled by the number of bins the input range is split into. A single bin is a (useless) model with very high bias and very low variance. Adding to the number of bins reduces bias and increases variance.

When X is high dimensional, the bins need to be multidimensional too, being squares in two dimensions, cubes in three dimensions and so on. As the input space grows, the data become more sparse (unless the sample size grows exponentially with the number of variables, which is another aspect of the curse of dimensionality). This increases the risk of each bin containing one or zero data points unless the bins are very large. The alternative to using multidimensional bins is to build a dependence model like a Bayesian network (see section 1.9.3).

Kernel Density Estimation

The problems associated with a histogram can be solved by smoothing the estimate to avoid the sharp jumps and removing the need for bins. **Kernel density estimation** makes use of every point in the training data to estimate $f(X)$, making it a non-parametric technique. To calculate $f(x)$, which is the density at point x, the distance between x and every other point in the data is calculated using what is known as a kernel function. The average of these distances provides the estimate of $f(x)$. Formally, a kernel density estimator calculates:

$$f(x) = \frac{1}{Nh} \sum_{j=1}^{N} K(\frac{x - x_j}{h}) \tag{1.50}$$

where K is a kernel function, which we will define next, and h is a smoothing parameter known as the **bandwidth**. The $x - x_j$ component calculates the distance between x and each data point in the sample and summing then dividing by N calculates the average kernel output. Dividing the distance by h controls how much smoothing is performed. Larger values of h produce smoother functions but making h too large can over simplify a model. In other words, h controls the bias variance trade-off.

There are a number of choices for the kernel function, K, but the standard normal density (with a mean of zero and standard deviation of one), $\phi(Z)$ is commonly used, producing the **Gaussian kernel density** function of equation 1.51. As $\phi(x - x_j)$ is maximised when $x - x_j = 0$, points near x contribute more to the sum in equation 1.51 than more distant points. The more nearby points there are, the larger the sum becomes.

$$K(Z) = \phi(Z) = \frac{1}{\sqrt{2\pi}} exp(-\frac{Z^2}{2}) \tag{1.51}$$

where $Z = (x - x_j)/h$ as in equation 1.50.

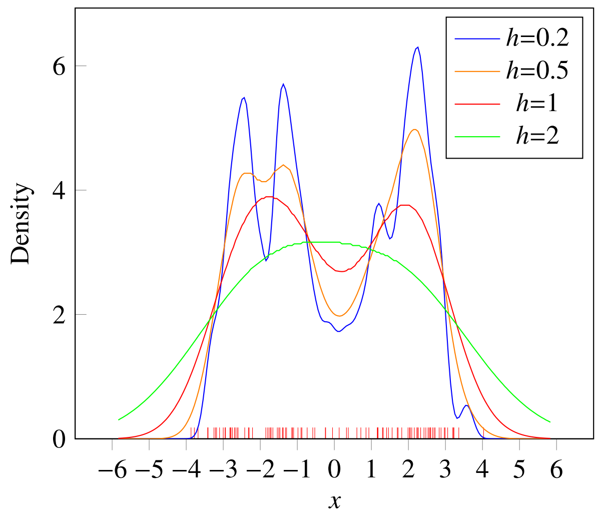


Figure 1.17: Gaussian kernel density plots fitted to the distribution shown as a rug plot along the X axis with different values of h. Note how $h = 2$ produces a model that is too general, $h = 0.2$ is over fit and $h = 1$ appears to the eye to be the best estimate.

In more than one dimension, multivariate kernel density estimation places multivariate kernels at each data point. The concept is the same, except that the smoothing parameter, h becomes a matrix, H, which can control the size, direction and orientation of the kernel. If H is a scalar times the identity matrix (zeros everywhere except the leading diagonal, which is constant) then the smoothing is equal in all directions. Allowing different positive values on the leading diagonal allows H to smooth more in some directions than others and any positive definite matrix controls the orientation and direction of the smoothing. See Sheather [**?**] for an accessible description of kernel density estimation.

Intuitively, you should be able to see that calculating the density by smoothing the distance between any given point and all the other points leads to locations that are near a lot of the observed data points having a higher density. In fact, you should be able to see that this really is a good definition of density - all the data points contribute but the closer ones contribute a lot more.

Mixture Models

One large disadvantage of kernel density estimators is that as the size of the dataset grows, so do the time and memory needed to evaluate new points. A more efficient solution is to place a small number of kernels at locations where the data are most dense. This is where density estimation and clustering coincide as the small number of kernels are placed in a way that allows the data to be separated into clusters. **Mixture models** make the assumption that the distribution observed in a set of data is the result of a number of components being mixed together—like the adult basketball players and their five year old children in the example in section 1.7.1. Mixture models are defined by the place and shape of the component functions and by weights that determine their relative contributions to the density estimate.

When the component functions are normal distributions, the model is known as a **Gaussian mixture model** (GMM). A univariate GMM is defined by a set of K Gaussian functions, each defined by a mean, μ_k and a variance, σ_k. Additionally, each function has an associated weight, α_k where $\sum_{k=1}^{K} \alpha_k = 1$, which determines the size of its contribution to the overall density estimate. The density estimate is calculated by

$$f(X) = \sum_{k=1}^{K} \alpha_k \phi(X|\mu_k, \sigma_k) \tag{1.52}$$

where $\phi(X|\mu_k, \sigma_k)$ is the normal distribution with mean μ_k and variance σ_k.

One way to think about what mixture models do is the think of there being an additional, unseen variable that labels the contribution to the overall distribution. These are known as **latent variables**. In the basketball example, the latent variable labels people as players or children.

The Expectation Maximisation Algorithm

Fitting a GMM from data involves estimating values for each μ_k, σ_k and α_k. A common approach is to use the **Expectation Maximisation** (EM) algorithm, which is a maximum likelihood estimation technique. In an approach that will remind you of the algorithm for calculating the centroids of a k-means clustering, the EM algorithm iteratively calculates the parameter values of the model in two steps, called the expectation (E) step and the maximisation (M) step. One difference between k-means clustering and GMM clustering is that the clusters generated by k-means are spherical but the covariance matrix used in GMMs allows the component functions to have different widths in different directions. Another difference is that k-means provides a hard clustering, as described at the start of section 1.8. GMMs provide a soft clustering where each point has a probability of belonging to each of the possible clusters.

Intuitively, the EM algorithm makes use of the observation that it would be easy to estimate the parameters μ_k, σ_k and α_k of each separate component if the data points were labelled by component (if we knew the values of the latent variable, in other words). That corresponds to the M step. They are not labelled like that, but it is also easy, given the parameters and the data, to calculate the probability of each point belonging to each component. That is the E step. By iterating those two steps, the algorithm moves from an initial guess to a local maximum in terms of likelihood. The calculations carried out at the two steps are as follows. Firstly, the probability of each point falling into each component is calculated. The probability of point i falling into component k is ρ_{ik} and is calculated as

$$\rho_{ik} = \frac{\alpha_k \phi(x_i|\mu_i, \sigma_i)}{\sum_{j=1}^{K} \alpha_j \phi(x_i|\mu_j, \sigma_j)} \tag{1.53}$$

where ρ_{ik} is the probability that x_i is generated by component C_k, in other words $\rho_{ik} = P(C_k|x_i, \alpha, \mu, \sigma)$. Once estimates for ρ_{ik} have been made, they are plugged into the M step as follows

$$\alpha_k = \sum_{i=1}^{N} \frac{\rho_{ik}}{N} \tag{1.54}$$

$$\mu_k = \frac{\sum_{i=1}^{N} \rho_{ik} x_i}{\sum_{i=1}^{N} \rho_{ik}} \tag{1.55}$$

$$\sigma_k = \frac{\sum_{i=1}^{N} \rho_{ik} (x_i - \mu_k)^2}{\sum_{i=1}^{N} \rho_{ik}} \tag{1.56}$$

$$\tag{1.57}$$

Note how the parameters are calculated from the probabilities and the probabilities are calcualted from the parameters. All that remains is to initialise the parameters so we have somewhere to start. The parameters are initialised so that each μ_k equals one point taken at random from the data, the variances σ_k all equal the sample variance and the weights ρ_k equal $1/K$. You could also initialise the parameters using k-means clustering.

Once the GMM has been built, a new data point, x can be evaluated using equation 1.51, which calculates the probability of x, given the components. To assign a data point to a cluster requires a calculation of the probability that each component generated x, which is achieved using Bayes rule:

$$P(C_k|x) = \frac{\alpha_k \phi(x|\mu_i, \sigma_i)}{\sum_{j=1}^{K} \alpha_j \phi(x|\mu_j, \sigma_j)} \tag{1.58}$$

where C_k is component k. Calculating equation 1.58 for each C_k gives a soft clustering and picking the C_k with the largest density produces a final hard clustering decision.

1.8.1 Discrete Probability Models

The density models described so far have been used to model continuous distributions. Where the data are nominal or discrete, a different approach is needed because you cannot fit continuous kernels (for example) to nominal variables. In the description that follows, we will use the word discrete to mean discrete, nominal or binary (in other words, not continuous).

1.9 Dependence Modelling

A simple univariate distribution on a discrete variable is easy to calculate. You simply count the number of times each discrete value occurs, divide that by the total number of observations, and produce a lookup table. You might think that the same principle can be extended to any number of variables, for example if we have variables $X_1 \in \{A, B\}$ and $X_2 \in \{C, D\}$, we would just count the occurrences of AC, AD, BC and BD. The problem is that as the number of variables grows, the number of possible combinations of values grows exponentially. Even with a modest scenario such as 20 binary variables, the number of combinations is 2^{20}, which is over a million. If some combinations have low probabilities, then the sample size required to produce a lookup table by counting every combination is many times the number of combinations.

To understand the solutions to this problem requires an understanding of the concept of independence among variables. In terms of a probability distribution, two variables are independent if the distribution of either one of them does not depend on the value of the other. That is to say, variables X_1 and X_2 are independent if the joint probability distribution across them both is equal to the product of their marginal probabilities.

$$P(X_1, X_2) = P(X_1)P(X_2) \tag{1.59}$$

from which it follows that $P(X_1) = P(X_1|X_2)$ and $P(X_2) = P(X_2|X_1)$. In other words, the marginal probability distribution of X_1 is the same as the conditional probability distribution of X_1 given X_2, and vice versa. For example, if I flip two coins, the probability of the second one coming up heads is the same, regardless of the result of the first flip. The probabilities are independent. On the other hand, if I measure two variables about the weather: $Rain \in \{Y, N\}$ and $Cloudy \in \{Y, N\}$ then $P(Rain)$ is different when $Cloudy = Y$ from when $Cloudy = N$ so $Rain$ and $Cloudy$ are dependent variables.

Variable independence is useful for building statistical models of probability distributions because we do not need to count combinations for groups of independent variables. Consider the coin flipping example. As the outcomes are independent, we can specify the full distribution with two marginal probabilities. The joint probability of a combination of values across the variables is calculated by multiplying the individual marginal probabilities together, as stated in equation 1.59. The probability of flipping a head, then another head is calculated as the probability of the first coin coming up heads multiplied by the probability of the second coin coming up heads. This extends to any number of coins (or variables). If you have p independent variables, $X_1 \dots X_p$ then the joint probability distribution is the product of the marginal probabilities:

$$P(X) = \prod_{i=1}^{p} P_i(X_i) \tag{1.60}$$

where $P_i(X_i)$ is the probability function associated with variable X_i.

Building such a model is easy, as we simply build a lookup table giving the probabilities for each individual variable. Such models are simple, but have model bias in the shape of the assumption that all variables are independent of each other. Modelling some degree of dependence among variables allows more complex models to be built and provides potential for reducing the bias. At one extreme is the simple assumption of independence among all variables, which has high bias and low variance. It requires less data than other more complex models but can lack accuracy if the assumption is incorrect for the given data. At the other extreme is the complete multivariate lookup table, which has low bias but high variance and (as we have seen) requires more data and processing time than is generally available. Somewhere between those extremes lie models with some dependencies among variables such that bias is reduced and variance controlled to an acceptable state.

It is common to define dependence among variables as a graph in which nodes represent variables and edges represent dependence. The following section summarises the main approaches to modelling probability distributions with graphical models. Such models are used for both discrete and continuous distributions, but the section illustrates the ideas with discrete distributions only.

1.9.1 Probabilistic Graphical Models

Graphical models

A probabilistic graphical model is represented by a set of vertices (or nodes) and edges, $G = (\mathbf{V}, \mathbf{E})$. The nodes represent variables and the edges represent dependence between nodes. Here we will limit the description to graphs where the nodes are connected in pairs, though higher order graphs are also possible in which an edge can connect any number of nodes. Graphs can be **directed**, in which case the edges have directions indicated by arrows or **undirected**, in which case the relationship is symmetrical. Directed graphs often represent conditional relationships whereas undirected graphs model dependence. A multigraph contains unconnected subgraphs, so in terms of probabilistic models, a multigraph represents a distribution in which some variables are completely independent of some others.

A graph can also be split into subgraphs even when it is not a multigraph. These subgraphs are known as **cliques** and are defined as a set of nodes in which every pair of nodes are connected. A graph can have cliques of different sizes, including every node being in its own clique of one. All connected pairs form cliques of two, and so on. A **maximal clique** cannot have nodes added to it from the rest of the graph and still be a clique, so it is the largest clique containing its members that can be formed.

Independent Models

The simplest model assumes all variables are independent, so its graphical representation is a multigraph of single nodes with no edges. Such models are never really represented as a graph as there is no useful analysis that can be performed on such a graph, but we include it here as the starting point. As already stated, the probability mass function for an independent model is

$$P(X) = \prod_{i=1}^{p} P_i(X_i) \tag{1.61}$$

and each $P_i(X_i)$ is calculated by building a lookup table for each discrete value X_i can take.

Maintaining the idea of a multigraph, but allowing the subgraphs to contain more than one node and assuming full dependence within each subgraph (i.e. every variable depends on every other within the subgraph but on no other variables outside it) provides a method for varying the complexity of the model while keeping the simple lookup table approach. The variables in X are split into c non-overlapping subsets, $C_1 \dots C_c$ and the full joint probability distribution of each subset is calculated as a lookup table with every combination of values the subset can take. The function that this table represents is called a **factor**. The subsets need to be kept small as the factors grow exponentially in size with the number of variables covered. The probability mass function is

$$P(X) = \prod_{i=1}^{c} P_i(C_i) \tag{1.62}$$

where C_i represents a subset of X and $P_i(C_i)$ is the joint probability mass function (the factor) over the variables in C_i. The subsets can be discovered using a greedy algorithm that starts with an independent model like that described above and iteratively combines nodes so that the cost function is reduced by the largest amount at each step. This approach was used by Harik [**?**] as a way of modelling distributions as part of an optimisation approach known as the compact genetic algorithm and is called the **marginal product model**.

Many functions cannot be represented as a product of independent variable subsets. For those functions, the convenience of calculating small independent distributions is not available so an approach is needed that allows dependence between any variables in a single graph. Two approaches are common. Bayesian networks model the conditional probabilities of connected variables, and so are directed graphs. Markov random fields model dependence among variables and are consequently undirected. The following two sections provide a little more detail on each.

1.9.2 Markov Random Fields

Markov random fields (MRFs) represent dependencies among variables with undirected connections, leading to a natural representation in cases where dependencies are not causal. The marginal product model described in section

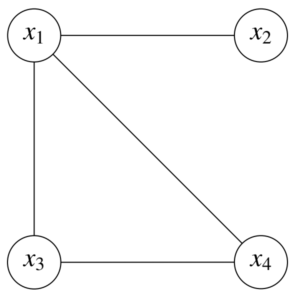


Figure 1.18: A four variable Markov Random Field. The cliques are $\{x_1\}$, $\{x_2\}$, $\{x_3\}$, $\{x_4\}$, $\{x_1, x_2\}$, $\{x_1, x_3\}$, $\{x_1, x_4\}$, $\{x_3, x_4\}$, $\{x_1, x_3, x_4\}$. Variables in the neighbourhood of x_1 are $N_1 = \{x_2, x_3, x_4\}$. Similarly, $N_2 = \{x_1\}$.

1.9.1 is an example of a Markov random field, but the idea can be extended to allow the factors to overlap. For example, a model with variables X_1, X_2, X_3, X_4 might be represented as $P(X) = \frac{1}{Z}P(X_1, X_2)P(X_2, X_3, X_4)$. Note that X_2 appears in both factors and that a normalisation, known as the partition function, Z is required to make the probabilities sum to one. Z is the sum of all the unnormalised probability estimates over every combination of input values and cannot be calculated for any but the smallest of input vectors. We will return to consider Z a little later.

The edges in a MRF represent dependence between pairs of variables and form the definition of the neighbourhood of each variable as the set of nodes to which it is directly connected. The neighbourhood of X_i is denoted N_i and represents the set of nodes with edges to X_i (excluding X_i itself). The key properties of a MRF are concerned with conditional independence among variables. Two variables, X_1 and X_2 have conditional independence given a third variable, X_3 if knowing the value of the X_3 causes X_1 and X_2 to be independent. In terms of a graphical model, conditional independence between X_1 and X_2 given X_3 means that X_1 and X_2 are not directly connected by an edge, but there is a route from X_1 to X_2 via X_3.

The **global Markov property** applies to groups of variables. It states that any two subsets of variables, A and B, are conditionally independent given a separating subset, C. If we remove all the nodes in the separating subset, there will be no route between any of the nodes in A and any in B (they will become independent). The **local Markov property** states that a variable, X_i is independent of all other variables, given its neighbours, N_i. Finally, the **pairwise Markov property** states that two nodes are conditionally independent given all the other nodes if there is no direct edge between them. A graph of a set of variables that satisfies these properties is a Markov random field.

Furthermore, if the probability distribution over the variables is strictly positive (i.e. $P(X) > 0$) then it can be represented by a MRF if it can be factorised over the cliques of the graph. This is known as the Hammersley-Clifford theorem [].

How are the factors for a MRF defined? Once the graph structure has been defined, the maximal cliques are identified and a factor (also known as a **clique potential**) is calculated over the variables in the clique. The values given by the clique potential function are not probabilities; they are better thought of as measures of the strength of agreement among variables. The network can be thought of as being in a state of tension with the clique potentials pulling the values into agreement. The level of agreement between the values and the potentials is known as the *energy* of the network. The joint probability function over all the nodes in a MRF can be written as the normalised product of its maximal clique potentials

$$P(X) = \frac{1}{Z} \prod_{c_i \in CM} \phi_i(c_i) \tag{1.63}$$

where CM is the set of maximal cliques, c_i is the set of nodes in maximal clique i (for example, $\{X_1, X_2\}$) and $\phi_i(c_i)$ is the result of evaluating c_i with the clique potential function associated with that clique. Finally, Z is the partition function

$$Z = \sum_{X} \prod_{c_i \in CM} \phi_i(c_i) \tag{1.64}$$

where the sum is over every possible combination of values in X.

a	b	$P(c = 1\|a, b)$	$P(c = 0\|a, b)$
0	0	0.2	0.8
0	1	0.3	0.7
1	0	0.75	0.25
1	1	0.1	0.9

Figure 1.19: An example conditional probability table in which the child node, c is dependent on the parents, a and b. All variables in the example are binary. Note that the probabilities in each row sum to one and there is one entry for every combination of values over a and b.

Building a MRF requires both the connectivity structure and the clique potentials to be estimated. A detailed description of the methods for doing these things is beyond the scope of this book except to say that neither are trivial. For a more indepth treatment of the topic, the reader is referred to chapter 19 in Kevin Murphy's excellent book *Machine Learning* [] and chapter 17 of the equally good *The Elements of Statistical Learning* by Hastie et al. [**?**].

1.9.3 Bayesian Belief Networks

Markov random fields are represented by non-directed graphs because two variables are either independent or they are not. In cases where there is a natural direction to the relationships among variables, a directed graph is more appropriate. Bayesian belief networks (BBNs) are one example of such a graph. The nodes in a BBN represent variables, just as they do in a MRF but the edges in a BBN represent conditional dependence. A connection from a parent node to a child node indicates that the value of the child node is conditional on the parent node. Such a graph cannot contain cycles as a variable cannot be conditionally dependent on itself, even via other variables. Bayesian networks are represented by directed acyclical graphs (DAGs) for those reasons.

As with the MRFs, a detailed treatment of BBNs is outside the scope of this book, but we will present the basic ideas and dedicate some space to a discussion of their uses and relative merits. The goal is not to give you the ability to implement one from scratch (that is not as simple as it is for MLPs and decision trees), just to allow you to use appropriate software correctly and interpret the results you get.

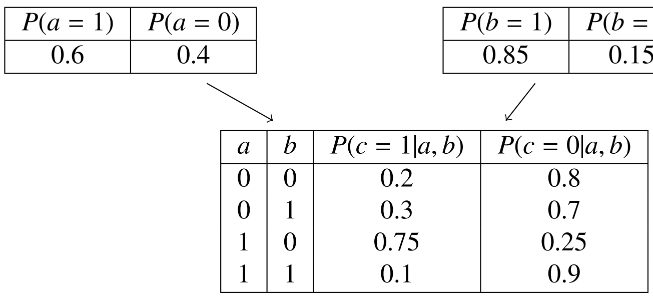
Where MRFs have clique potential functions, BBNs have a far simpler and more intuitive approach. Each child node in the graph (that is, each node with an incoming connection) contains a conditional probability distribution for the single variable represented by that node, conditioned on its parents. The joint distribution, $P(X)$ is factorised over the conditional probabilities in the graph so that

$$P(X) = \prod_{i=1}^{N} P(X_i|X_{pi}) \tag{1.65}$$

where X_{pi} represents the set of nodes that are parents of X_i.

The simplest way to represent the conditional probability functions in a discrete BBN is to list the probabilities in a table. These are known as **conditional probability tables** (CPTs) and they list the conditional probability of the variable at the child node taking each of its possible values for every possible combination of values over its parent variables. Figure 1.19 shows an example. The number of such combinations grows exponentially with the number of parents a node has. This means the quantity of data required to calculate the probabilities in a CPT grows exponentially with the number of parent nodes. A simple solution is to limit the number of parents and node can have. This introduces bias into the model. Whether such a restriction is acceptable depends on the true distribution of the data. Other methods for coping with the problem involve replacing the CPT with a more compact model such as a decision tree [].

Figure 1.20 shows a Bayesian network with the CPT tables at the nodes. The structure and CPT values either be designed by hand or learned from data. The CPT probabilities are easy to calculate from the data. You simply count the number of times each value in the child occurs for every combination of patterns over the parent variables, as shown in figure 1.19. The structure of the network is harder to discover. The challenge of structure discovery is to balance complexity with accuracy. That requires a cost function that combines measures of both. A common accuracy measure is log likelihood, coupled with a penalty for the number of parameters in the model. The number of parameters is calculated as the sum of one less than the number of columns in each CPT (the last column in each



$P(a=1)$	$P(a=0)$
0.6	0.4

$P(b=1)$	$P(b=0)$
0.85	0.15

a	b	$P(c=1\|a,b)$	$P(c=0\|a,b)$
0	0	0.2	0.8
0	1	0.3	0.7
1	0	0.75	0.25
1	1	0.1	0.9

Figure 1.20: A simple Bayesian Belief Network showing the marginal probability distributions of variables a and b and the conditional probability distribution of c given a and b.

CPT takes the total to 1, so adds no new information). One common measure is the **Akaike Information Criterion**, (AIC) which is $-2LL+K$ where LL is the log likelihood and K is the number of parameters. A lower AIC is preferred. Another is **minimum description length** (MDL), which is $-LL + K/2 \log n$, and should also be minimised.

When and How to use MRFs or BBNs

Markov random fields and Bayesian belief networks are used to model joint probability distributions. They are able to represent both discrete and continuous variables, though we only looked at the discrete case here. The main reasons for modelling a distribution are to evaluate the probability of new events (to perform novelty detection, for example); to fill in missing values in new observations; and to generate new samples of data that are very similar in their distribution to the data from which the models were built. Such models have also been used widely in the field of heuristic optimisation where they play a key role in estimation of distribution algorithms (EDAs).

BBNs model causality among variables whereas MRFs make no such assumptions. It is easier to establish independence with a data set than it is to establish causality, which generally also requires human knowledge or assumptions. Another problem with MRFs is that they cannot represent induced and non-transitive dependencies. Two variables, X_1 and X_2 may be independent, but still joined in the graph because because some other variable depends on them both or because X_3 depends on X_1 and X_2 depends on X_3. A Bayesian network does not suffer from this same limitation.

The functions used in BBNs (for example, the CPTs) are easier to interpret and design by hand than the potential functions in a MRF, which has made them popular in fields such as medical diagnosis []. There also seem to be more software packages available for building BBNs than there are for using MRFs. Useful BBN software includes Smile and Genie [] and BUGS [].

1.9.4 One More Classifier—Naïve Bayes

Now that we have an understanding of graphical network models of probability distributions, let us return to the task of classification one more time with a look at a method based on modelling the probability distribution of each class directly: the naïve Bayes classifier. To build a classifier that calculates the probability of an input vector, X belonging to one of a number of classes represented by a variable, C requires a model of the conditional probability distribution $P(C|X)$. It is much easier to model the probability distribution of a subset of the data for each class, in other words $P(X|C)$. By splitting the data by class label and modelling the distribution of each split and then applying Bayes rule, we get

$$P(C|X) = \frac{P(X|C)P(C)}{P(X)} \tag{1.66}$$

The probability distribution over the class labels, $P(C)$ is easy to calculate: it is just proportion of the data belonging to each class. The probability distribution over the data as a whole, $P(X)$ is harder to model, but does not vary with respect to the class (it models all the data over all the classes) so just acts as a normalising constant. As the task of the classifier is simply to find the class for which the probability of the data is highest, there is no need to divide all the estimates of $P(C|X)$ by $P(X)$. That turns the task of classification into the task of calculating $P(X|C)$ for each class in C and picking the one with the highest value.

What makes the approach naïve is the assumption that is made when modelling $P(X|C)$: that there is independence among all the variables in X. A naïve Bayes classifier has high bias because of this assumption, but in situations where that bias is correct (i.e. when the variables really are independent), it performs very well. By relaxing the naïve assumption of independence, more complex Bayesian classifiers can be built. In principle, with perfect knowledge or sufficient data, a full and correct model of $P(X|C)$ could be built, but as we saw in section 1.8.1, that is not generally possible. In reality then, so called semi-naïve Bayesian classifiers can be built using any of the methods discussed in section 1.9.1.

1.9.5 Novelty Detection

Novelty detection (also called anomaly detection or outlier detection) involves judging whether or not new measurements from a data source are different from those produced by that same source in the past. Common examples include machine health monitoring, where abnormal readings might indicate the need for maintenance; fraud detection where abnormal credit card use might indicate fraud; and data journalism where unusual events are of natural interest.

You can probably see how clustering algorithms or probability models can act as novelty detectors. Any event with a very low probability estimate may be considered novel and any event that is far from the centre of any cluster may also be flagged as novel.

In this section we present one method for novelty detection that is based on the multilayer perceptrons discussed in section 1.6.5. The idea is simple. Given only input values, you build an MLP in which the target outputs are the same as the inputs. The job of the MLP is to recreate the input pattern at the output layer. That would be trivial except that the number of hidden units is limited to fewer than the number of inputs, forcing the network to encode the relationships among the inputs in an efficient way. Any data point in which the relationships among the variables do not conform to the encoding discovered by the model are reproduced with poorer accuracy, and can be identified as novel.

Thompson et al. [**?**] call this the autoencoder approach. They use a multilayer perceptron that learns to reproduce the inputs at the output layer and calculate novelty as the difference between the inputs and the outputs.

Williams et al. [**?**] and Hawkins et al. [**?**] propose a replicator neural network approach with a novel stepped activation function rather than the smooth logistic. They claim that this promotes a discrete clustering of the data which improves novelty detection. Their MLP has three layers. The outer two layers use standard tanh activation function given in equation 1.43 but the middle hidden layer uses the step function, which is calculated as

$$f(act) = \frac{3}{4}\sum_{j=1}^{N-1} tanh(a(act - \frac{j}{N})) \tag{1.67}$$

where N is the number of steps required of the function and a controls the rate of transition from one step to the next (set by the authors at 100 in the work they report).

1.10 Meta-Methods

The simple machine learning methodology presented so far involves picking a single technique and a simple split of the data such as cross validation. In fact, there are many ways of improving both the choice of modelling technique and the way in which the data are sampled. These methods are aimed largely at improving the bias and variance of the resulting system and are discussed next.

1.10.1 Bootstrap Sampling

A fundamental issue when estimating models and their parameters from data is that the available data represent just one sample from the massive set of possible samples drawn from the population. Any parameter estimate based on the sample will be one of the many possible estimates that might have been made. These many possible estimates are known as the **sampling distribution**. The mean of this distribution is the best estimate (among those in the distribution) of the true parameter value and the variance of this distribution is an estimate of the variability of the estimate from sample to sample. It provides a measure of confidence in the estimates derived from the data.

In reality, a large number of samples are not available; generally there is only one. Some parametric models allow inferences about the sampling distribution to be made from a single sample based on an assumption about the distribution of the population data. For example, if the mean, $\bar{X}$ of a variable is calculated and the assumption of a normally distributed population is made, then the confidence intervals of the mean can be calculated using values from the t distribution.

For non-parametric methods, there is not a similarly straight forward method for calculating confidence intervals. One strategy for gaining insight into the variation that might be expected from different samples is to use a procedure called **bootstrapping**, which generates a large number of data sets by sampling with replacement from the original sample. Each of the bootstrap samples contain the same number of data points as the original data set (hence the need for replacement). Model building and parameter estimation is then repeated for each bootstrap sample and the models are all compared. The comparison can be across parameter values themselves or across measures of accuracy such as mean squared error.

Bootstrap sampling can be used as an alternative to k-fold cross validation. Each bootstrap sample will contain a proportion of the data from the training set as some points will be chosen more than once and some will not be chosen at all. Those that are not chosen for the sample are used as validation data to provide a measure of the variance of performance across a large number of validation samples. In a data set of N points, the probability of any single point being selected during N uniformly random samples is $1 - ((1/N)^{-1}) = 0.632$, which means that around 63% of the training data will be used for training on each bootstrap repetition.

1.11 Ensemble Methods

So far this chapter has mainly discussed single machine learning methods carried out on single training data sets. Cross validation and bootstrap methods have been introduced, which build several models, but only as a means to assess model variance. This section describes a class of modelling approach that involves building and combining multiple models from a range of re-samplings of the training data. Such approaches are known as **ensemble** methods.

An ensemble is a set of models all built to attempt the same task, but in different ways, or with different samples from the training data. Models may be built independently from each other or based on the performance of models that are already in the ensemble. Different ways of building ensembles are discussed in the following sections.

Once an ensemble has been built, the resulting models must be combined in some way to produce a single system for generating predictions from new data. Some models can be combined into a single model based on an average of their parameters. Take the simple linear model, $Y = aX + b$ for example. The parameters are a and b. Performing bootstrap sampling allows you to build M models with parameters $a_1 \ldots a_M$ and $b_1 \ldots b_M$. The average of a prediction from all of those models is

$$\bar{f}(X) = \frac{1}{M} \sum_{i=1}^{M} a_i X + b_i \tag{1.68}$$

which we can expand to become

$$\bar{f}(X) = \bar{a}X + \bar{b} \tag{1.69}$$

where $\bar{a} = (\sum_{i=1}^{M} a_i)/M$ and $\bar{b} = (\sum_{i=1}^{M} b_i)/M$.

There are many modelling techniques that do not support this type of averaging, however. Consider a decision tree where each model in the ensemble has a different structure or a MLP where any given weight in one model does not map onto the same weight in a different model. The averaging cannot be done by creating a new model, so must be done by aggregating the outputs of all the models.

Equation 1.68 can be recreated for any regression ensemble as the aggregated prediction is simply the average output across all the models. For a classification task, the simple choice for aggregation is to either pick the most common classification across the ensemble or (in models that support it) average the probability estimate associated with each class and choose the largest. It is tempting to count the number of models that agree as a proportion of all models and treat that as the probability of the input belonging to the majority class. This does not make sense, however as the models can all agree, but with a high degree of uncertainty. Imagine an ensemble where all of the models classify an input as belonging to class A with probabilities between 0.6 and 0.7. The fact that they all agree does not make the probability of the input belonging to class A suddenly equal one. Averaging the class label probabilities across the ensemble is a better way to estimate individual class probabilities.

Models may also be combined using a weighted aggregation, where different models contribute more or less strongly to the final decision. One weighting scheme makes use of the test error for each model, allowing those with better performance to contribute more. An improvement on this approach is to take model parsimony into account and weight models by both their accuracy and their complexity (for example, the number of parameters they require). Measures such as AIC and BIC can be used to produce such weightings.

Aggregating the members of an ensemble every time a prediction is needed is slow and memory intensive as all the models need to be stored and evaluated. Some ensembles can contain many different models using several different techniques (a mixture of MLPs, decision trees and SVMs, for example) so the software required to make a prediction would need to implement all of these techniques. The other problem with an ensemble is that it loses some of the interpretability that single models such as decision trees have. Considering the entire ensemble as a single, deterministic system suggests an approach that involves building a single model that mimics the ensemble. This is known as **model compression** [**?**] and involves generating a new training set in which the outputs are generated from the ensemble, not the original data labels. This new data set should be perfectly learnable as there is a noiseless mapping from inputs to output. Training a single decision tree to mimic an ensemble of many decision trees not only simplifies the prediction process but returns a degree of interpretability that is missing from the ensemble. Model compression is discussed further in section 1.11.4.

1.11.1 Bootstrap Aggregation

Section 1.10.1 describes the bootstrap method for resampling with replacement from a training data set to allow a large number of different models to be built and compared. That still leaves the question of which model to choose. A simple approach is to build a final model with all of the training data and report its performance on the test data along with the bootstrap estimate of performance variability.

A better approach involves making use of all of the models built during the bootstrap process. This approach still uses all the training data, as the probability of a data point never being picked is very small. It also makes use of the fact that combining many models can reduce bias without increasing variance. Building many models from bootstrap samples and aggregating the results is known as **bootstrap aggregation** and is often called **bagging**.

Building a set of models using bagging is straight forward. Simply generate many bootstrap samples from the training data and build a model from each. The models can then be aggregated using any of the methods discussed in section 1.11.

1.11.2 Boosting

The ensemble methods discussed above all build models independently. The results of one model do not affect the building of the next. An alternative approach is to build the first model and then build a second one that addresses the errors of the first, then a third that addresses the errors of the second, and so on. Each model can be simpler as it does not have full responsibility for learning the complete function. A method known as **boosting** combines several so called **weak** models into a strong ensemble using precisely this approach.

The most popular boosting algorithm is called **AdaBoost.M1** (meaning additive boosting) and is explained next. The standard AdaBoost algorithm is designed for classification tasks (a version for regression is described in section 1.11.3). AdaBoost works by iteratively building simple models that are trained on a weighted version of the training data. The weights are adjusted at each iteration to reflect the errors made at the last iteration. The concept of boosting is independent of the technique used to train each weak model. You could combine logistic regression models, for example, but the most common approach is to combine decision trees.

Whichever technique is used, the concept of re-weighting the training examples means the same thing. If training example (x_i, y_i) has the associated weight w_i, then it is treated as if it appears w_i times in the training data. For decision tree learning, for example, the entropy calculations that determine which variable is added at each node are based on the number of times a class label appears in the part of the training data defined by that split (see section 1.6.4). That number is altered by the weight given to each training example so that the class labels of highly weighted examples are counted proportionately more than those with lower weights. Other techniques can be weighted by changing the cost function to include the weight for each training instance. Ultimately, if you cannot change an algorithm to perform weighted learning, you can always resample the data with replacement so that each data point has a probability of being selected that is proportionate to its weight.

Algorithm 3 The AdaBoost.M1 Algorithm for Binary Classification

```
Let $\mathbf{D} = (x_i, y_i), i = 1 \ldots N, y_i \in \{-1, 1\}$ be the training dataset
Choose a number of models to build, $M$
Initialise the instance weights $w_i = 1/N, i = 1 \ldots N$
for do$m = 1 \ldots M$
    Train a classifier $G_m(X)$ on the data in $\mathbf{D}$, with each entry weighted by $w_i$
    Calculate $err_m = \frac{\sum_{i=1}^N w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^N w_i}$
    Calculate $\alpha_m = \ln((1 - err_m)/err_m)$
    Update the weights $w_i \leftarrow exp(\alpha_m I(y_i \neq G_m(x_i)))w_i$
end for
Model is $\hat{y} = \text{sign}(\sum_{m=1}^M \alpha_m G_m(x))$
```

Algorithm 3 describes the AdaBoost.M1 algorithm. Making the instance weights equal to $1/N$ means that the first model is built in the normal way for that algorithm, except that it is restricted in its complexity. A common approach is to use a **decision stump**, which is a decision tree with only one decision node. This model is $G_1(X)$. The error for the first model, err_1 is the proportion of data points in the training data that $G_1(X)$ classifies correctly. Note the use of the indicator function $I(y_i \neq G_m(x_i))$ which returns 1 if $y_i \neq G_m(x_i)$ and 0 otherwise.

The next step calculates $\alpha_m = \ln((1 - err_m)/err_m)$, which is the contribution to the overall model made by model $G_m(X)$. This value is used in two different places. Firstly, in the weight update step of the algorithm, the weight for training instance i, w_i is multiplied by α_m if the instance is misclassified and left unchanged if it is correct. That causes incorrectly classified instances to be more heavily weighted in the next model to be built. The second place in which α_m is used is when the finished model makes classifications of new data. Here, model m is weighted by α_m so that the output class for a binary classification, $\hat{y}$ is

$$\hat{y} = \text{sign}(\sum_{m=1}^{M} \alpha_m G_m(x)) \tag{1.70}$$

Note that $y_i \in \{-1, 1\}$ in this binary classification example.

Boosting is a very powerful technique and can produce very good results in the right circumstances. Stopping the training process early (with very few models) produces bias but adding further models (increasing M) does not necessarily lead to overfitting. In fact, it has been found that continuing to add models after the training error has reached zero can sometimes improve the validation error as the probabilities associated with each class move further towards the extremes. The individual trees do not need to be decision stumps; they can be more complex, but they should not be so complex that they overfit. Hastie, Tibshirani and Friedman [**?**] suggest trees with between 4 and 8 leaf nodes work well in many practical situations, though the optimal tree size can be searched for using cross validation. For a good discussion of the complexity, accuracy, and size of models built with AdaBoost, see Mease and Wyner [**?**], along with the responses from other practitioners, including Friedman et al.

1.11.3 Gradient Boosting

AdaBoost trains new models to add to the ensemble by weighting the training data points causing poorly classified points to be learned more than those that are correctly classified. An alternative approach, that can be used for both regression and classification, makes use of the cost gradient to train new models, effectively performing gradient descent on the cost function. This approach is known as **Gradient Boosting**. The advantage of gradient boosting is that is can be applied to any differentiable cost function.

Initially, let us consider the squared error cost function, which we will use to perform regression with a boosted ensemble of regression trees (see section 1.6.4). The errors for each tree are known as its residuals and the process, known as **forward stagewise additive modelling** involves training each subsequent tree on the residuals of the previous one. To keep the process consistent, the first model is set to simply output the average of Y for all inputs and the second model learns the residuals from that. The mean plus the modelled residuals is a better prediction than the mean alone (of course). The next model learns the residuals from the second one, and so on.

Regression trees are a good choice for the base learner as they can account for interactions among the variables. Boosting a set of multiple linear regression models (see section 1.6.1) does not bring any improvement because a single such model minimises squared error as far as is possible with a linear sum. There is no benefit in producing a linear sum of a linear sum. Incidently, if you use boosting to build a series of simple linear regression models (those with a single input variable), the result is the same as that for a single multiple linear regression model: squared error is minimised. Each model uses the one variable that is best able to model the residuals from the previous step, so it performs a stagewise variable selection. Stopping early is one way to perform variable selection. In all cases, deciding when to stop adding models can be achieved using cross validation.

Recall from section 1.3.1, that gradient descent is used to minimise a cost function by changing model parameter values iteratively based on local gradients in that cost function. Gradient boosting does the same thing by fitting each subsequent model so that adding it to the ensemble descends the cost gradient. The process using squared error described above is generalised to any cost function for which local partial derivatives can be calculated. Recall too that gradient descent makes use of a learning rate that controls the size of each parameter update. Gradient boosting does a similar thing with an update rate, commonly denoted as v, and called the **shrinkage rate**, which is greater than zero and less than one. Each new model learns at a rate controlled by v so that only a proportion of the residuals (or only a proportion of a step down the error gradient) is learned by each new model. Choosing smaller values for v slows down learning, requires a larger ensemble, but can be effective against overfitting.

1.11.4 Model Compression

Ensembles are generally more accurate than single models, but they are larger and slower too. Where speed is important and resources are limited (for example on a chip in an engine management system), a model that is compact and fast is required. A final method of interest in the discussion of ensembles is known as **model compression** [**?**]. This is a technique for training a single model to reproduce the behaviour of an ensemble. The single model is not trained on the original training data, rather training data for the single model is generated from the ensemble. If the ensemble's behaviour is deterministic (and most are) then there is a perfect mapping between the inputs and the output in the newly generated data. The noise and variance have been removed and assuming the ensemble model performed well on test data, a single model that achieves zero error when it mimics that model should perform just as well.

The compression process is as follows. First generate large quantities of unlabelled data and let the ensemble label it. With such large training sets, overfitting can be avoided. The only problem is that a large unlabelled data set may not be available. In such cases, the solution is to generate the data randomly. One might be tempted to generate the new data from a uniform distribution over all the variables, but this leads to sparse coverage of the regions of input space where the data naturally fall. A better approach is to sample from a distribution that is as close to that of the training data inputs as possible. A method for this is known as **MUNGE**, and involves generating new data points that are close to those in the original data set, but have randomly set differences.

The original compression paper trains neural networks (MLPs, in fact) as the compression models, exploiting the fact that they are universal approximators and should in theory be able to reproduce an ensemble of arbitrary size and complexity. The problem with neural networks, however, is that they are hard to interpret. A decision tree would be better if the final model needed to be small, fast and interpretable by human eye. There is a method that solves this problem by sampling unlabelled data, using an MLP to add labels, and then using that labelled data to train a decision tree. It is known as **tree extraction from neural networks** and one such method, called ExTree is worth describing here [**?**]. One problem with building decision trees is that the size of the partition on which nodes are split shrinks as the tree grows deeper. Extree is interesting because it allows a new set of data to be sampled and labelled by the MLP at each level, ensuring that every node is evaluated on an equally large sample of training data.

1.12 Minority Class Problems

Many classification tasks involve detecting instances of rare cases. Examples include spotting fraud, selecting prospects who will respond to direct marketing, detecting an illness and predicting a stock market crash. It is not easy to collect large numbers of examples of the class of interest and misclassification error can often be minimised by simply always selecting the most common class. Take the direct marketing example: a 5% response rate might be considered very good for a campaign, which means that 95% of the data from a previous campaign are labelled No and only 5% are labelled Yes. Imagine two possible models, one classifies the 5% of Yes responses correctly, but also says Yes to another 30% of the test set leading to an error rate of 30%. Another classifies all of the test set as No, giving it an error rate of only 5%. Which model is most useful? Obviously, the model that never says Yes is useless so the model with the larger error should be preferred in this case.

1.12.1 Resampling Methods

Resampling the training data is one way to balance the output classes prior to learning. The two simple options are to down sample the majority classes or to up sample the minority classes. A new data set can be generated by selecting instances from the original data set with probabilities that are the inverse of their proportions in the data. The resulting data set will contain the same number of instances from each of the possible output classes. The problem with down sampling is that it discards what might be useful data from the majorit class and the problem with up sampling is that it can lead to overfitting of the minority classes as they are duplicated exactly.

A better approach, known as Synthetic Minority Over Sampling Technique (**SMOTE**) aims to solve this problem by generating new data that is similar, but not identical to the existing examples in minority classes. The process of generating new minority class examples is to take each instance from that class in turn, identify its k nearest neighbours, and generate new points on the line segment that joins the chosen instance to each of its neighbours. This evens the distribution of class labels, avoids overfitting the minority class values and extends the boundaries of the minority class to fill the space among them.

1.13 Data Visualisation

The human eye is very good at spotting patterns and understanding large quantities of data if they are presented in the right way. Data can be presented as scientific charts, which you will see in research papers and text books like this one. Networks of connected objects can be displayed as graphs, which might represent routes in an airline schedule or the relationships between pairs of people in a social network. Both charts and graphs are also the inspiration for infographics, which mix graphic design and data representation to present data in an attractive and informative way.

This section first describes the correct way to present scientific charts and graphs, and then shows how those guidelines are broken, sometimes to good effect, sometimes unwisely, and sometimes with the intention to mislead. Before we start, let us define a naming convention. A **plot** displays data as points at coordinates in space, a **chart** uses shapes and areas to represent data and a **graph** can either represent a function as a continuous line or a network of connnected nodes.

1.13.1 Scientific Charts

Many scientific charts involve two axes, which relate to a variable each, and a representation of how those two variables interact. The horizontal axis is known as the X axis and the vertical as the Y axis. When describing a chart, it is usual practise to say "Y is plotted against X", so "Price plotted against time" means that price is on the Y axis and time is on the X axis. The following sections describe some examples in detail, but which ever method you use, make sure you include the following:

- Make sure both axes are labelled with their variable name and the units being measured (Height in cm for example).
- If you plot more than one series of data, include a legend that identifies each series. For example, use different coloured lines, different shaped markers or solid, dashed or dotted lines. Consider whether the chart might be printed in black and white when using colours alone.
- Label the axes with a numeric scale or a set of nominal values. Consider whether the scale should start at zero. If it does not, consider the consequence of that choice on what the chart appears to show. Small differences can be magnified by manipulating the origin of the Y axis in particular.

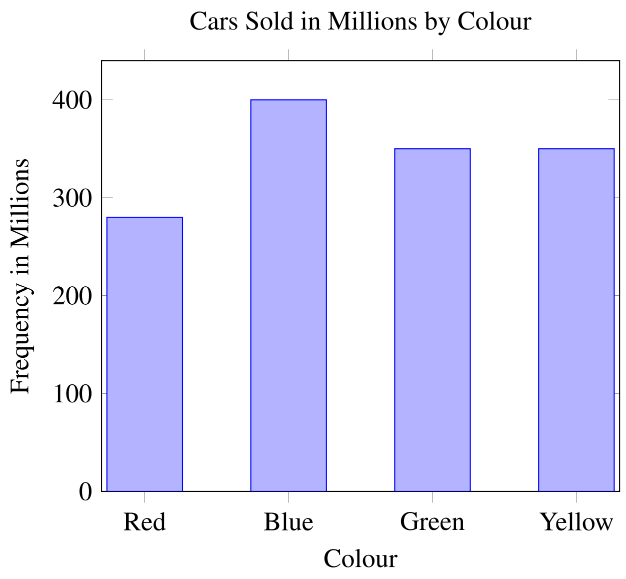


Figure 1.21: An example bar chart showing the number of cars of different colours sold in millions.

- Include a title and a caption. If the origin is not at zero, consider mentioning the fact in the caption.
- Finally, the data may have a source that needs to be cited in the caption or title. For example, you might write "Unemployment figures from 2000 to 2017. *Source: The Office for National Statistics*."

Representing Frequency Counts

The most popular methods for representing frequency counts are **bar charts** when the variable being counted takes discrete or nominal values and **histograms** when the variable takes continuous values.

Bar charts can be presented as vertical or horizontal bars, in which the length of the bar is proportional to the frequency count and the width of the bars are all the same. Figure 1.21 shows an example bar chart showing frequency counts up the Y axis. Bar charts can also be used when one variable is discrete or nominal and the other is an aggregation such as a sum or an average. For example, you might plot average life expectancy against country of residence with a bar chart. The bars in a bar chart are presented with spaces between them to highlight the fact that the input axis is discrete.

Histograms are used when the variable being counted is continuous. This variable is split into a set of contiguous subranges (called bins) so that counting (or some other aggregation) can take place. Each bin has an associated bar in the histogram, but unlike a bar chart, the quantity is represented by the *area* of the bar. Most commonly, the bins are all the same width in a single histogram, however. The bars in a histogram are plotted without gaps between them, highlighting the continuous nature of the input range. Figure 1.22 shows an example.

When building a histogram, you must decide how many bins to split the data into. If you choose too few, the bars are too wide and hide the detail of the distribution. If there are too many, you can end up with a very small number of data points in each, producing a wide and low histogram. You should recognise this as an other instance of the bias-variance trade-off. The more bins you use, the lower the bias and the higher the variance would be across histograms built from different random samples from the same population. There are a few common rules of thumb for choosing the number of bins, including the square root of the number of data points and Sturges' formula, which rounds up $\log_2 N + 1$. You may need to experiment with different bin sizes to find the right one. This can be dangerous as you may find that your beliefs (or hopes) about the data are supported better by one choice than another.

How ever many you choose, the bins should cover the range that is greater than or equal to the bin minimum and less than its maximum. In mathematical notation, that is $[min, max)$. The exception is the final bin, where the range is $[min, max]$. The Y axis represents frequency, but can also represent proportion. In a **normalised histogram** the Y axis represents the proportion or percentage of the data that falls in each bin. The heights add up to one (or 100 for percentages) and also represent the probability of a randomly picked data point falling into each of the bins.

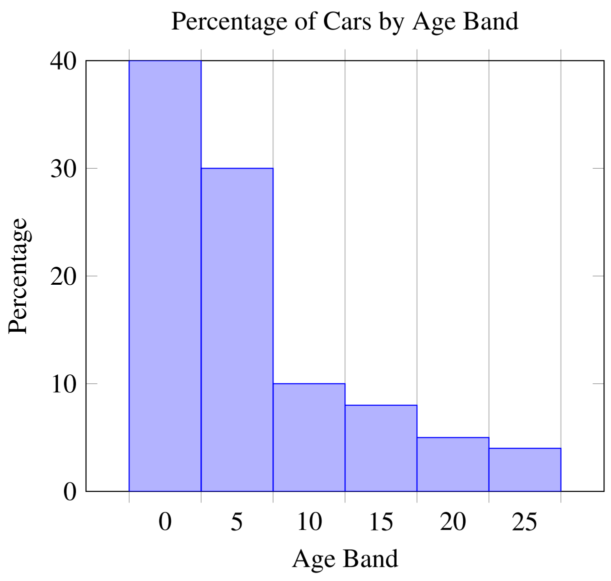


Figure 1.22: An example of a normalised histogram showing the percentage of cars on the road of different age bands.

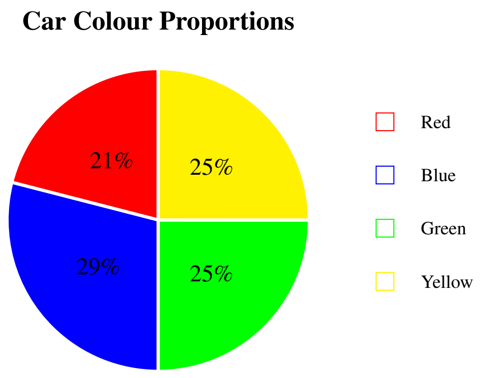


Figure 1.23: A pie chart showing the proportion of cars on the road by colour.

Pie charts represent frequencies as areas too, but instead of rectangles, the areas are segments of a circle. Pie charts have a number of drawbacks, which make then less than ideal for scientific use. Figure 1.23 shows the same car colour data from figure 1.21 as a pie chart. It can be difficult for the observer to compare the areas of two segments that are similar, particularly if they are not adjacent in the pie. Each segment needs to be a different colour (or pattern) so they are not suited to data with more than a small number of different values. Labelling a pie chart can be messy, especially when segments are small and care is needed that the order of items in the legend is the same as the order of segments in the pie. Some software packages allow you to make matters worse by creating 3D pie charts that tilt away from you at a jaunty angle. This distorts the angles in the chart itself, making it even harder to read. Pie charts can be useful if used correctly. They work well as infographics, where exact measurements are perhaps less important than presentation. For example, a pie chart that shows how the top three companies in a market account for three quarters of all revenue, but that the final quarter of revenue is shared among 120 small companies can be presented well as a pie chart as the size of the chart does not vary with the number of values being plotted as it does in a histogram or bar chart.

Heat Maps show aggregations such as totals, sums and averages using colour or depth of colour. A common use for a heat map is to plot data on a geographical map, which is often split into discrete contiguous regions. Examples include the total population of each state or the average income for each county. Some use a **rainbow** colour scheme, which maps the number being represented to a colour in the visible spectrum. Such schemes can be confusing as

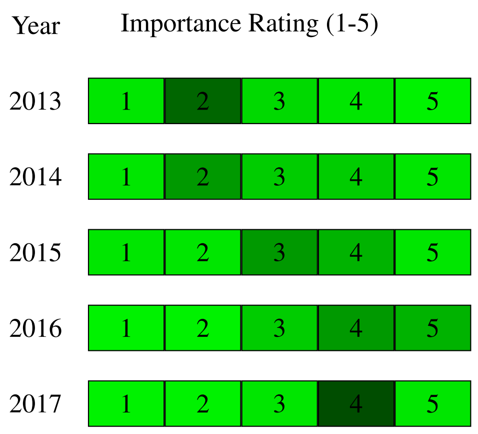


Figure 1.24: Using a heat map to show how the frequency of responses to a question has changed over time. Depth of colour indicates frequency. Using colour instead of bar heights shows the pattern of change clearly and uses less space.

the eye does not naturally see certain colours as being greater or less than others on a scale. The alternative is to pick a single colour and vary its depth with the number being represented. This has the advantages of being more visually intuitive and working well in greyscale. A simpler heat map can be a useful alternative to a barchart when space is limited. Each bar is replaced with an equal sized box that is filled with the appropriate colour, which allows multiple charts, all representing the same measurement from different sources (or at different times) to be compared. Figure 1.24, for example shows how responses to the same five point attitude question change over time. The same method can be used to summarise responses to surveys that use discrete numeric scales to measure attitudes towards many questions in the same format. Consumer satisfaction surveys are a good example as the answer format (usually something like five points from Very Unsatisfied to Very Satisfied) is the same for every question.

Density Graphs use a line to represent a density function that has been fitted to a continuous variable, either by fitting a parametric model or by smoothing a histogram. Sometimes, the histogram and the density line are presented on the same graph to indicate the level of fit between the function and the data, as in figure 1.25.

Plotting two Variables or More

Histograms and bar charts plot aggregations (usually counts, but others such as averages may also be presented in this way). When the raw data are to be presented, it is more common to use **scatter plots** or **line plots**.

Scatter plots present the raw data of two (or more, in some cases) variables so that each point that is plotted represents a single data point. Most commonly, both variables are numeric and continuous, but third and even fourth variables can be incorporated into a plot and these can be nominal or numeric. Two of the variables are represented by their coordinates on the plot. This is the standard method in a scatter plot, as illustrated in figure 1.26.

Further variables can be added by manipulating the shape, colour and size of the points that are plotted. To plot two continuous variables against a third nominal variable, for example, you could choose a different shape or colour for each of the nominal values. A legend must be included to show the meaning of the shapes and colours. Figure 1.27 shows an example that uses the X and Y axes to represent two numeric variables with colour and shape to represent two further nominal variables. Three continuous variables can be plotted by varying the colour of points on a scatter plot. The first two dimensions are a straight forward scatter and the third is represented by colour using a scheme similar to that used in heat maps. A third continuous dimension may sometimes be represented by varying the size of the points, but only in cases where there are sufficiently few data points and they are sufficiently spread out so that the larger points do not obscure the smaller ones. These are known as **bubble charts**.

When the variables plotted on a scatter plot are discrete, there can be a problem when many points are identical. In such cases, **jitter** can be applied to the points, which moves each point by a small random amount in a random direction. This spreads the points out and makes it easier to see how many points there are at different places in the plot.

Another approach to plotting two or more discrete variables is to show the relationships between the values in

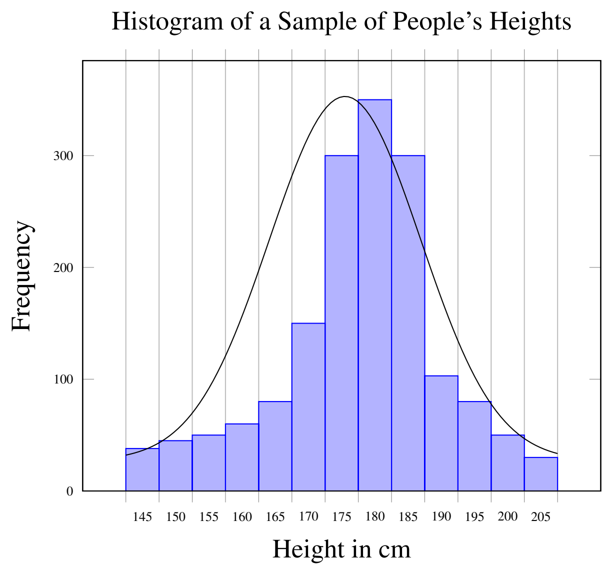


Figure 1.25: An example of a histogram and a density graph. The histogram represents the frequency of data in each of a set of subranges and the density graph is a normal distribution with the same mean and variance as the data (mean = 178, variance=130).

pairs of those variables. This is known as a **Sankey diagram** and represents a directed graph where the nodes are possible values that the variables can take. To plot a Sankey diagram between X, which takes values in $\{a,b,c\}$ and Y, with values in $\{d,e,f\}$, create a graph with three nodes on one side to represent $\{a,b,c\}$ and three on the other side for $\{d,e,f\}$. This is a bipartite graph, which means connections can only go from one side to the other, not within the same side. The edges that connect the nodes can carry values, which are indicated by the thickness of the line. For example, you could represent the distance between pairs of cities with line thickness or the result of football matches over a season where the node sets are home team and away team and the line thicknesses show goal difference. You can also allow the line thickness to represent the frequency with which the joined values of X and Y coincide in the data. Figure 1.28 shows an example Sankey diagram of the relationship between the number of seats in a person's car and the number of children they have.

Scatter plots can help you choose which variables to include in a model, but care is needed. The presence of a relationship in a scatter plot is good evidence of a relationship in the data but including two inputs that are correlated with both the output and with each other adds very little to a model that is not captured by just one of the inputs. This means that simply picking variables that look useful in a scatter plot may lead to a larger set of inputs than a more sophisticated variable selection method would (see section 1.5.2 for more details on this). What is more, a scatter plot that shows no relationship between an input variable and the output (something like a round cloud of points, for example) does not provide evidence that the variable is not useful. Variables that appear to provide no predictive power on their own can provide strong predictive power in combination (think of the XOR function for a simple example).

When two variables from a multi-dimensional space are plotted against each other, the resulting plot suffers from being what is known as a **projection** of the high dimensional data on a two dimensional surface. Think of flat shadows on a wall from a three dimensional room: it is impossible to tell whether two shadows that appear next to each other as projected on the wall come from objects in the room that are adjacent or not as they may be next to each other, or one may be at the front and the other at the back. Either way, their shadows could touch. Projection can be addressed using different shapes and colours to differentiate points in higher dimensions, but only up to a point.

Line Plots are used to plot one continuous variable against another, where each value on the X axis has a single measure or aggregated value associated with it on the Y axis. The points are joined by a line to emphasise the continuous nature of the X axis and to highlight trends and other patterns. Each point is only joined to one or two others (the *next* point in some sense) so there needs to be a natural ordering to the X axis. An X axis that represents time is a common example as each point is joined to those closest to it in time. The Y axis is often an aggregation, for example plotting total sales against time or average temperature against month. In any case, each series (a single

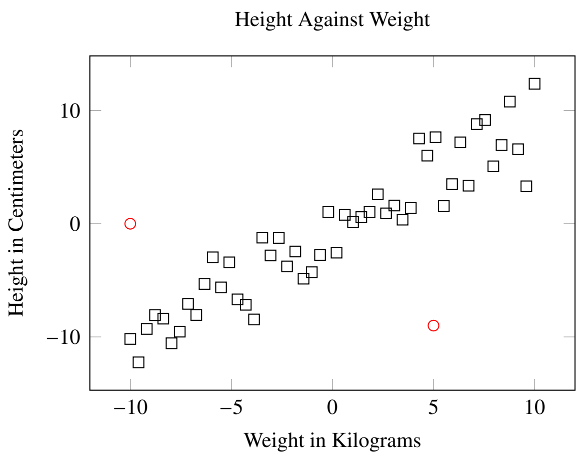


Figure 1.26: A scatter plot of height against weight from a sample of people. A visual representation of the nature and strength of the relationship is provided along with identification of outliers (can you see them both?).

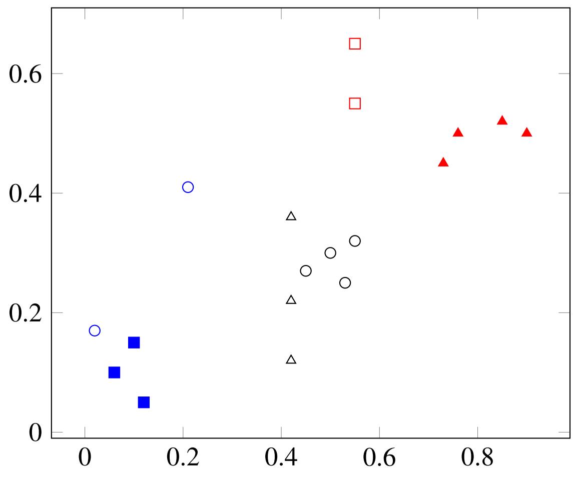


Figure 1.27: A scatter plot in which plots two numeric (the two axes) and two nominal (shape and colour) variables.

line of the chart) should have only one Y value for each point on the X axis. Multiple series can be plotted as separate lines, often distinguished by colour, style (dotted, dashed, solid, etc.) and point style (round, square, filled, open, etc.).

Generally, the joining lines should be straight. You are representing the data, not modelling it and while curved lines might look nicer, they can distort the data in an attempt to smoothly join the points. Talking of the points, always mark them on the line plot so the viewer knows what are data and what are joining lines. Figure 1.29 shows an example line plot that exhibits these qualities. Note that both axes are continuous, which means joining the dots makes sense, and that the X axis (Time) is ordered, which dictates the order in which point are joined.

Graphing Networks

Any set of objects that are related in pairs can be considered as a network and drawn as a graph of nodes, which represent the objects, and edges, which represent the relationships. Examples include geographic networks such as the routes of an airline or the roads between towns; social networks like family relationships or groups of friends; commercial networks such as buyers and sellers in a market; and communications networks such as the phone network or the internet. We have already met Sankey diagrams, which graph a particularly restricted type of network, but now we will look at the more general case.

Simple networks such as who knows who in your circle of friends can be drawn and analysed in detail by visual

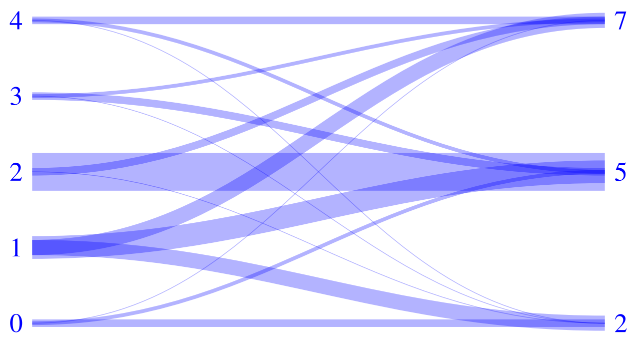


Figure 1.28: A Sankey diagram showing the relationship between the number of seats in a person's car and the number of children they have from a sample of data. The thickness of the lines indicate frequency of the co-occurrence in the data. You can see that the majority of cases involve two children and five seats. Note how reducing the opacity of the lines helps the viewer follow lines that overlap.

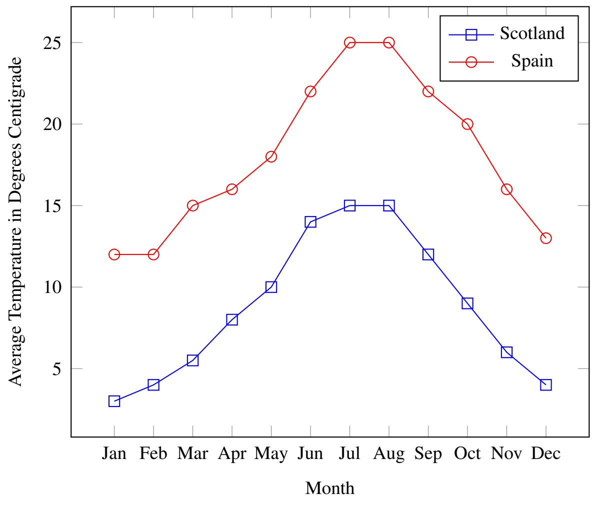


Figure 1.29: Average temperature in degrees centigrade across the year for Scotland (blue line, square points) and Spain (red line, round points).

inspection. Larger networks may still be drawn to give a visual impression of the overall structure, but individual nodes and connections become more difficult to examine as the network grows. When drawing a graph, one challenge is to choose where to place each node. With the exception of geographical networks, most graphs allow the nodes to be placed anywhere within the bounds of the image. A common approach is to attempt to keep connected nodes near each other and unconnected nodes apart. Another goal is to minimise the number of edges whose lines cross. A common approach is to use a **force based layout** algorithm, which incrementally adapts a layout based on a metaphor of spring like forces among the nodes. Unconnected nodes repel each other and connected nodes attract so that the lengths of the edge lines are kept short.

An alternative layout, known as an **arc diagram**, as shown in figure 1.30 places the nodes on a line and joins them with part-circular arcs of varying size that are placed above and below the line. More complex diagrams can be created by allowing line thickness or colour to represent variables such as frequency or score.

1.13.2 Infographics

When presenting data in newspapers, magazines and posters, people have taken the ideas of scientific plotting and adjusted them with the intention of making them more visually appealing. This section describes some of the ingredients that go into infographics. It also issues some warnings about errors that should be avoided when making infographics.

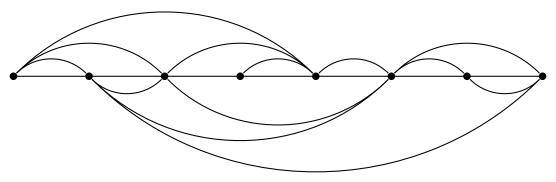


Figure 1.30: A simple arc diagram showing connections between pairs of points arranged on a line.

Representing Quantity

A popular method for representing frequency counts on an infographic is to use a row or column of repeated icons. For example, the number of people in different categories can be represented by rows of little images of people. The frequency count can only be a fixed number of icons, unless the last in the row is sliced to show a proportion of the image. When showing large differences, this is less of a problem and the fact being presented is that there is a large difference, rather than the exact numbers.

Bar charts can also be approximated by images of varying size, but care is needed because the bars need to all be the same width and only vary in length. Simply scaling an image so that the length is proportional to the frequency but allowing the width to vary with the length exaggerates differences as the area of the image grows faster than the length.

Another approach to comparing frequencies on an infographic involves containers that are filled to differing levels. For example, you might compare the drinking habits of people from different countries using pictures of beer glasses filled to a height that represents the quantity drunk on average per person.

Word clouds are a popular method of representing the frequency counts of words in a document or other collection of words. For example, you might represent all the words in all the reviews of a product in a word cloud or summarise each chapter of a book (like I have done for this one in figure 1.31). Word clouds are formed by writing each word in a font whose size is proportionate to the frequency with which it occurs. There are online word cloud generators that allow you to upload text or provide a URL of a webpage from which the graphic should be generated. They allow you to choose other features of the graphic such as font style and colour.

The images for infographics can be created in software but photography can be used to great effect too, with scenes being composed from real objects and then photographed. Obvious examples include piles of coins to represent quantities of money or piles of books to represent literacy rates. You could build a bar chart out of Lego and photograph that to represent data about children or slice a real pie if your data are about food.

An infographic is generally more than a single chart or picture. Many are presented as posters or full pages of graphics and make use of colour, large fonts, images and the data represention methods described above to present a theme supported by data. A good infographic tells a story about the data, rather than presenting a loose collection of charts and pictures. You can lead the reader from one part to the next explicitly with lines and arrows and you can include small amounts of text. The best way to get a good idea of the range of infographics that have been produced is to simply type *infographics* into your favourite search engine and look at the images page. There are plenty of tools for generating infographics, for example Piktochart [**?**], Canva [**?**] and Venngage [**?**].

1.14 Text Mining

With the rise of social media and Twitter in particular, interest in automatically extracting meaning from text has grown rapidly. Allowing a computer to fully understand natural language is still an ongoing reseach topic with many smaller goals such as automated translation, automated document summary and the ability to engage in dialogue with a human. Language and the meanings conveyed by it have components that follow rules such as grammar but that is not the full story. The meaning of a sentence is also determined by more than the role the words play and their definition. Consequently, methods involve a mixture of linguistic rule processing and statistical modelling. This chapter is concerned with statistical modelling of data and so this section's scope will be limited to statistical text mining. In particular, we will look at one application area: text classification.

A topic that has grown in importance recently, known as **sentiment analysis**, attempts to classify statements

Figure 1.31: An example word cloud built from the words in this chapter. The larger the font, the more common the word. Note that some stop words (will and may, for example) have not been removed. Removing them would produce a more insightful summary of the topic of the chapter.

in terms of the sentiment they express. The most common form is a binary classification with classes that can be considered *good* or *bad*; or *positive* or *negative*. The approach is used widely to summarise the feelings of large numbers of humans towards a specific topic or product. Popular example applications include text mining of movie or product reviews, tweets during sporting events, and attitudes towards certain stocks and shares. This section takes you through the steps of a typical sentiment analysis project but is by no means a full treatment of the subject.

1.14.1 A Text Mining Project Methodology

A text mining project can follow the CRISP-DM methodology described in section 1.4.1 with some steps that are specific to text mining. This section presents such an approach.

Business Understanding

This stage might start with a simple statement: "*I want to understand what social media are saying about our new movie*", for example. The sentence tells us a lot. It identifies a source of data (social media) and defines something about the task to be performed. What do we mean by "*what social media are saying*" though? Let us be more precise. A standard sentiment analysis project could answer the question "*What is the balance of feeling for or against the movie on social media?*". It might then go on to ask "*what are the most common positive / negative words being used about the movie?*". It might be used to flag reviews that should be read by a human so that more sense can be made of public opinion.

This stage should also look ahead to the deployment stage and ask how the results will be used. Some text mining projects simply produce a report. For example, a student of mine plotted the attitudes of UK politicians towards the gulf war over a period of time by performing sentiment analysis on Hansard, which contains the records of what is said in parliament. The result was an interesting report, but nothing was deployed. On the other hand, a project that summarises reviews on the Amazon web site would require a deployment that learned as more reviews were written and showed more than the simple star rating system to potential customers of each product.

Data Understanding

The raw material of a text mining project is the existing text from which the statistical model will be built. This is often known as a **corpus** and is a collection of documents made up of written words. The documents might be newspaper articles, books, product reviews, social media posts or comments left on a website. One distiction that must be made is between **tagged** and **untagged** text. Tagged text includes meta-data such as classification labels, which you will need if you want to use supervised learning methods to build a sentiment classifier. There are plenty of existing corpora that you can download and experiment with.

Data Preparation

You cannot use full documents or even full sentences as inputs to a statistical model of text. The text needs to be cleaned and pre-processed and features need to be extracted before learning can take place.

The first step usually involves splitting documents into small units for analysis and is called **tokenising**. A **token** is a single unit, so you might split a document into sentences or words. The most common unit for tokens is the word. There are a great many words so reducing the number of unique words involved in the analysis is important. There are a number of ways of pre-processing the data to reduce the number of unique words involved, and they are described next. Tokenising can also involve ensuring that tokens contain no punctuation and that they are all in lower case.

Stemming involves reducing words to a common stem, for example by making plurals singular, representing verbs in their infinitive form and removing suffixes such as *ly*. So *buses* would become *bus*, *running* would become *run* and *quickly* would become *quick*. This attempts to reduce the number of unique words with minimal loss of meaning. The simplest way to stem words is to remove endings such as *ing* wherever they are found, but this can lead to non-words being generated (treating the word *swing* in this way produces *sw*) and fail to fix others such as *ate*.

The version of a word you would find in a dictionary (its canonical version) is known as a **lemma** and the process of **lemmatisation** involves replacing each word with its appropriate lemma. Rather than using a stemming approach, lemmatisation is generally done by maintaining a dictionary of words and their lemmas.

Stop words are defined as being those that should not be included in the analysis. This might involve removing short function words such as *a*, *the*, or *to* or any list of words that have been defined for the task at hand. There are lists of general stop words available in software packages and libraries. For example, the Python library, NLTK has a list of stop words you can use.

The variety of words to be analysed can be reduced further by replacing synonyms with a single common word. Some subtly of meaning might be lost if we replace **stroll**, **hike**, **amble** and **wander** with **walk**, but we gain more examples of the same basic meaning and reduce the size of the input vector.

Once the data is clean, feature extraction can take place. The simplest approach, known as **bag of words** creates a single variable for every word in the corpus. The total set of words from the corpus is known as a dictionary and is represented as a vector in which each word in the dictionary is a single variable. The values that the variables take when encoding a single document can either be binary, where 1 indicate the presence of the word and 0 its absence, or frequency counts of the occurrence of each word. The binary representation leads to simpler models and can be sufficient in some circumstances. For example, when performing sentiment analysis on Twitter data the documents are so small that words are not often repeated and the presence of a negative word is more important than how often that word appeared.

The size of the input vector can get large as there is a single variable for each word in the corpus. If the corpus is large but the documents to classify are small (tweets, for example) then the input is always sparse, with many words having a count of zero. Keeping the size of the dictionary down is an important consideration, which is one reason why measures such as stemming and stop word removal are important.

The bag of words approach assumes that the location or order of words in a document is not important (or at least does not need to be used). An improvement can be made by treating contiguous sets of words as single units. These sets are called **grams** so two words next to each other are called a **bigram** and in general *n* words an **n-gram**. Using bigrams can make an important difference to the meaning of sentences, particularly in sentiment analysis as it can capture the difference between things like "*good*" and "*not good*".

Another improvement comes by comparing the frequency of words in a single document with their frequency in the entire corpus. This measure is known as the **Term Frequency Inverse Document Frequency** or **TF-IDF** and is calculated by multiplying the frequency of a word in a single document (TF) by an inverse measure of its frequency in the corpus (IDF). The most common measure for TF is simply a count of the number of times the word appears in the document. Bias due to document length can be avoided by dividing the count by the number of words in the document. IDF is calculated as the log of the number of documents in the corpus divided by the number of documents that contain the term. Terms that are common in one document but rare across the corpus have a higher TF-IDF score than those that are common in both or uncommon in the document. This approach has a natural effect of reducing the weighting given to common words like stop words as they appear in all documents. Imagine you analyse one million movie review tweets and 800,000 of them contain the phrase "*this movie*", you would not place much value on that phrase when distinguishing one movie from another as it is so common.

Representing each word as a binary vector (also known as **one hot encoding) produces large, sparse input spaces, which require large amounts of data. This problem is addressed using word embedding models, which learn a representation of a large number of words is a smaller, more dense space. This involves projecting the one hot encoded representation onto a smaller space so that words that often appear together in sentences are close together in the new space. A well known example of this is Google's word2vec embedding [?], which uses a neural network to compress the word space. The inputs and outputs of the network are one hot encoded and the hidden layer is fixed to contain a small number of units. Word2vec uses two different network structures. One is called Common Bag of Words (CBOW) and predicts the current word from a bag of surrounding words. The other, the skip gram model, predicts the surrounding words from a single word. Once learning is complete, the hidden layer from this network is used to recode one hot encoded words, ready for input into your favourite machine learning algorithm. The Word2vec model training is unsupervised, so you do not need annotated data to train it. You can download and use pre-trained models from Google, who have used over 100 billion words to build their model.**

Modelling

In section 1.9.4 we met a simple classifier known as naïve Bayes, which can be used for document classification from a bag of words feature set quite nicely. Recall that a naïve Bayes classifier assumes independence among the input variables and models the distribution of each one separately. To classify a new example, Bayes rule is used to calculate the likelihood of the data having come from each class. The example data is then assigned to the class with the highest likelihood. Let us define things formally.

W is the set of all words in the corpus, in other words, the dictionary. Let $W \in \mathbf{W}$ be a variable that can take any word in W as its value. C is the set of classes to which a document might belong and $P(C)$ is the prior probability of a document belonging to class C calculated without reference to the contents of the document. D is the bag of words from a single document and contains a count for each word or (in the binary case) a value of one to indicate the word is present.

The probability distribution of individual words over documents in class c is $P(W|c)$ and the probability distribution of whole documents in class c under the assumption of independence is the product of the probabilities of the individual words it contains: $P(\mathbf{D}|c) = \prod_{w \in \mathbf{D}} P(w|c)$.

The prior probability of a document belonging to a class is simply the number of documents in that class divided by the total number of documents:

$$P(C) = \frac{|C|}{\sum_{c \in \mathbf{C}} |c|} \tag{1.71}$$

where $|C|$ is the number of examples in class C and the sum equals the number of documents in the corpus.

The probability distribution over the words in class c is calculated by counting how often each word appears in class c and dividing it by the total number of (non-unique) words in class c:

$$P(W|c) = \frac{|W, c|}{\sum_{w \in \mathbf{D}} |w, c|} \tag{1.72}$$

Note that the counts are across all the documents in the class. Whether you count word frequencies or just note the presence or absence of a word (which is achieved by clipping the counts at one within a document) the modelling process is the same. Bayes rule tells us that

$$P(C|\mathbf{D}) \propto P(C)P(\mathbf{D}|C) \tag{1.73}$$

which is

$$P(C|\mathbf{D}) = P(C)\prod_{w\in\mathbf{D}} P(w|c) \tag{1.74}$$

and the choice of C is that which maximises the likelihood calculated in equation 1.74. There are two problems with this approach that we need to solve, however. The first is the fact that there may be a word, w in the dictionary that does not appear in any of the training documents of class c, but could appear in a new document to be classified. As the probability of that word in the distribution for c, $P(w|c)$ would be zero, the product in equation 1.74 would be made equal to zero, regardless of how good an example the rest of the words made it. The solution is known as Laplace smoothing and involves adding one to all of the counts

$$P(W|c) = \frac{|W, c| + 1}{\sum_{w\in\mathbf{D}}(|w, c| + 1)} \tag{1.75}$$

which keeps the probabilities of words that are in the dictionary but not in class c small but not zero. The second problem concerns numeric underflow. If the probabilities are small to start with and we multiply a lot of them together, the computer running the algorithm will run out of floating point accuracy and all products will end up as zero. The solution is to take the sum of the log of each probability, which does not change the ordering and keeps the numbers in a range that is safer from numeric underflow. The score associated with each class, $S(C)$ becomes

$$S(C) = \log P(C) + \sum_{w\in\mathbf{D}} \log P(w|C) \tag{1.76}$$

Other classifiers can also be built from the bag of words input vector, including those described in this chapter such as logistic regression and multilayer perceptrons.

Evaluation

The same rules for training, validating and testing that apply to any machine learning project apply in text mining and sentiment analysis. The validation data are used to evaluate different models and different approaches to pre-processing such as n-gram size and which words to include in the analysis.

1.15 Convolutional Neural Networks

All of the methods we have discussed so far in this chapter have made use of structured data. That means the data are represented in tabular form with columns representing named variables and row representing single data points. Data such as images, video, text and sound are all known as unstructured data. That means they are not structured in tabular form (they still have structure, of course, images are matrices of pixels and text is an ordered list of characters, but we call it unstructured all the same because it not made up of named variables). There is a popular type of neural network that can handle unstructured data, known as a convolutional neural network (CNN), and this section describes them.

The observation that motivates the design of CNNs is that unstructured data are generally organised so that the meaning of any observed value depends more on the values of its close neighbours than the values that are further from it. All the pixels in a single face in an image tend to be near each other. Ideas are expressed in text

by words that are next to each other (like this!). We can think of these local groups of values as being features to be understood. Let's stick to image processing in the rest of this discussion so that we can be a little more specific. The concepts apply well to words and sound too.

Treating small groups of nearby pixels as features allows image processing neural networks to be a lot simpler than they would need to be if we treated pixels as inputs to a multilayer perceptron in which every pixel in the input would need to be connected to each hidden unit. CNNs are still feed forward neural networks (mostly) and they still have a design made up of layers, but in a CNN, we introduce new types of layers that act as filters, processing local patches of the layer before them. Let's say we want to build a CNN to classify images (most CNNs are used for classification at the moment). Some classes of image (faces, for example) have certain features (eyes and noses) that other classes (such as trees) do not possess. The job of the CNN is to identify these local features and learn how they combine to make larger features that finally combine to determine the class of the object.

This leads to a pipeline of filters that start very general and become more specific layer by layer. The first layer of filters might just look for edges of different orientations or corners in their small patch of image. Later layers combine those edges and corners into higher level features (maybe the corner of an eye), then later layer combine those (perhaps to make a whole eye) and then they are combined again until a set of features (an eye, a nose) determine the class (a face!). This final classification is done by fully connected layers that are the same as those we learned about earlier in the MLP. The output for a classificationn CNN is either a softmax or a set of logistics, just as in an MLP.

This is a beautiful idea, and it gets better because rather than somebody having to design all these filters, the CNN learns them for itself. Learning is similar to that of an MLP. Errors are calculated at the output and propagated backwards through the network, changing weights as they go. Batch gradient descent is used most commonly.

1.15.1 Convolutional Neural Network Architecture

We haven't described the filters or how they work. Let's fix that now. The mathematical operation you need to know about to understand CNNs is called (you guessed it) convolution. I'll describe the maths of convolution in a moment, but first we need to understand what it is used for. Take a simple pattern over three by three pixels, like that in figure 1.32. It has a vertical stripe down one side. Convolving this image with another produces a single number, and the higher that number is, the better the two images match. So, if we pass the filter over a large image in a moving window, producing a score for every patch it covers, we get a map of where in the original image there are vertical stripes (or whatever the filter is looking for).

So how is the filtering done? Take two images, both $n \times n$ **pixels in size, and call them** A **and** B**. The single value that is the result of convolving** A **with** B **is the sum of each pixel value in** A **multiplied by the pixel in the same location in** B**. Equation 1.77 states this formally.**

$$c(A, B) = \sum_i A_i B_i \tag{1.77}$$

To filter a whole image, we pass the convolution filter (also known as a kernel, by the way) over the image one place at a time in a moving window. A new image can be produced in which each pixel of the new image is the result of convolving a filter with the patch that the pixel is at the centre of. Figure 1.32 shows the result of passing the 3×3 **kernel shown over an image. See how the resulting image highlights the vertical edges of the original.**

So now we understand filtering. Each layer of a CNN has several (sometimes very many) different filters, so it produces many new images at the next layer. We don't want to keep filtering at the same level of detail though. We want to *zoom out* so that later filters cover a larger area of the original image. We also need a way of reducing the number of filters that are used as we progress through the network. This is done by something known as max pooling, which produces a single value from a patch of the layer before, but instead of filtering, max pooling simply produces the maximum value in the patch. This is a crude but effective way of shrinking the image and filtering over larger areas in later layers. CNNs make use of activation functions too. The most common in the convolutional layers is the ReLU, and it is applied to the filtered values before pooling. That is the basic structure of a CNN - convolutions followed by ReLU activation and then max pooling.

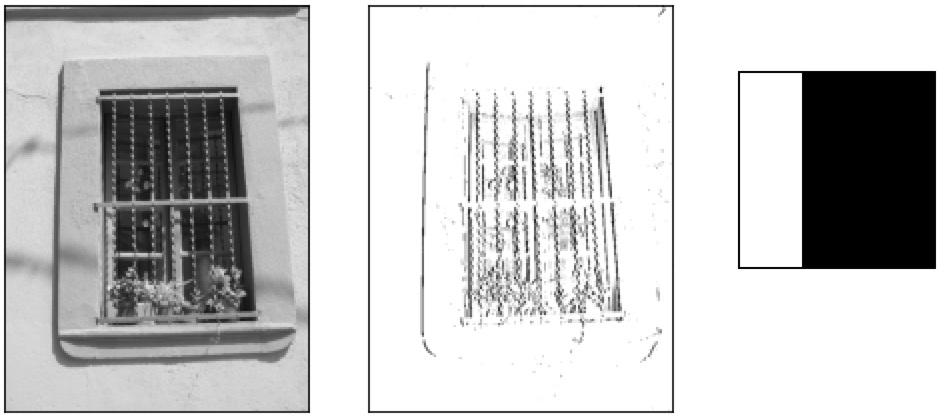


Figure 1.32: The image in the middle is the result of filtering the image on the left with the 3×3 filter on the right. The filter is three rows of [10, –4, –4]

Note that each layer produces a set of filtered images, one for each filter in the layer before. This means that each layer has a depth as well as the two dimensional image representation. You can think of the layers of the CNN as 3D volumes - stacks of filtered images. We call the images in the stack channels and refer to this dimension as depth. For colour image processing, the input layer has a depth of three - one for each colour channel. The convolutions in a CNN are done on 3D matrices (so called tensors). The operation in three dimensions is identical in principle to that in two. The filter is a volume and the convolutions are on patches of the same shape as that volume in the 3D volume of the incoming layer. You already know that two of the dimensions of the filter are vertical and horizontal pixel patterns. The depth of the filter matches the depth of the incoming volume.

Now you can imagine a small volume (the filter) sliding through a larger volume (the incoming image) and producing with each convolution a single value that is the sum of each value in the filter multiplied by the value in the place it currently occupies in the input volume. The convolutions don't quite work at the edges of the input as they would stick out over the edge, so it is generally padded with zeros to give extra space for convolving the pixels at the edges.

Finally, we should mention that CNN layers have bias weights too, one for each filter (that is the depth of the filter cube, remember). There is a lot going on there, so to help you see whether you understand it all, here is an example. Imagine a 32×32 input image with 3 colour channels. That is a $32 \times 32 \times 3$ volume. Now let's say we have five filters at the first convolutional layer, each one being 4×4 pixels big. These filters need a depth of three so that they can convolve the three input channels, so the filters are $4 \times 4 \times 3$ volumes, and there are five of them. The filters have a bias weight per filter, so there are 5 of those too. In total, then, the first convolutional layer has $4 \times 4 \times 3 \times 5 + 5 = 245$ parameters.

The size of the output from this convolutional layer is $30 \times 30 \times 5$. The 30×30 is because we lost the edge pixels as we didn't use padding, and the 5 is because we had five filters. We might then apply max pooling to produce a new volume of size $15 \times 15 \times 5$. See how pooling reduces the height and width, but not the depth.

To summarise for a convolutional layer:

- The input is a volume that is $height \times width \times depth$ where depth is the number of filters in the layer before or the number of input channels if this is the first layer (3 for full colour images).
- The number of filters is a hyperparameter chosen as part of the network design
- Each filter has the same depth as the input volume and a height and width chosen as part of the design (3×3 is common).
- The output resulting from filtering a padded $h \times w \times d$ input with a $f \times g \times d$ filter is $h \times w \times 1$ (or just $h \times w$).
- The output from convolving a padded $h \times w \times d$ input with n filters of size $f \times g \times d$ is a volume of size $h \times w \times n$

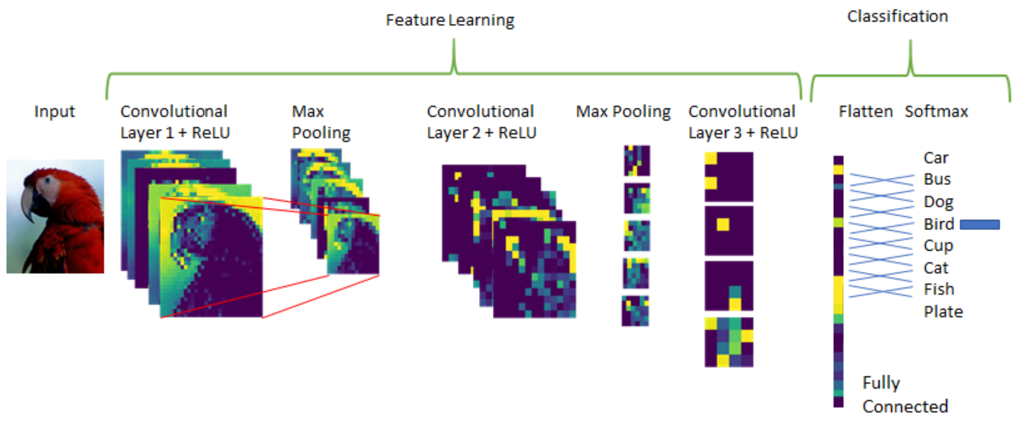


Figure 1.33: The structure of a convolutional neural network for image processing. Data flows from left to right.

- **Max pooling is then used to down sample this volume to $h \times w \times m$ where $m < n$ and that is the size of the input to the next convolutional layer (if there is one). Max pooling reduces height and width, but not depth.**

There is a lot to learn about CNNs, but I hope this has given you an insight into their structure and use. If you are competent with Python, you should take a look at Tensorflow and Keras. They can help you start to build CNNs quickly.

1.16 Running a Data Driven Project

If you are a consultant selling data mining or machine learning driven projects, or if you work for a company and you want to win managerial buy-in for your data driven idea, you will need to know how to describe, sell and run such a project. The CRISP-DM methodology will help you once the project is sold, but a lot of the hard work happens before that. There are technical, cultural and commercial barriers to getting a data mining project successfully off the ground and you will need a strategy for over coming them. This section describes them and offers some (hopefully) useful advice.

One problem is that the success of data mining projects depends on the quality and appropriateness of the data that drives it. If you cannot control the collection of that data (and you often cannot) then it is difficult to make any guarantees about the quality of the solution you will produce. If you commission a web design company to build you a website, it is reasonably easy to define the specification, agree on the functionality and identify the point at which the project has been successfully completed. If you under take a project designed to predict daily product sales based on social media analysis, you can agree what will be attempted, but you cannot guarantee it will work. The question of when the project is considered to be complete and when payment is made must be decided before the project starts. If the first attempts do not work, how much time do you spend trying other methods? Who pays for that time? What happens if the data simply do not support the task being attempted? These things can all be agreed, but it is important to do so before the project is signed off.

This brings us to the next point, the process of selling the idea or service of data mining to a client or sceptical manager. The temptation is to over sell—look at the miracles machine learning has performed, from cameras that can recognise faces to self driving cars! The line between what computers can and cannot do becomes more blurred every day and it is getting easier to believe that computers can learn anything from data. If you over sell, you will have greater problems when it comes to dealing with the first point, that of uncertain project outcomes. However, you want the job so you do not want to push that point too hard or too early. Identifying what might be possible is not as easy as it used to be, even for experts, so your manager or client will need help.

They will need to understand what it means for a computer to learn. When I started out in the business, this was an alien concept for many clients. People found it hard to believe that a computer could learn better than a human and even harder to believe that a computer could learn to do their job better than them! Selling a machine learning approach to insurance risk assessment to a professional underwriter is never going to be

easy. The next difficult concept is the idea that computers that learn also make errors. As you now know, when a machine learning practitioner talks about errors, they do not mean bugs that need to be fixed, they mean the natural variation between the real world and their statistical model. It is vital that a client understands the difference and over use of the word *error* in a sales pitch can be an unsettling experience for both client and seller.

I generally start with a bit of education. Establish what the client already understands and if they are not where they need to be, help them to get there before you start selling. The main things are that machine learning is used widely and successfully by many companies, it operates under a different paradigm from traditional IT projects and so involves a different set of risks and rewards. Explain how machine learning is task oriented, meaning it aims to produce a system capable of automatically performing a task, rather than being about analysing some data and producing a human readable report. The task often involves an automated decision which is used either as the final action or as advice to a human decision maker. Pitching a solution as a support tool for a human expert can ease fears of being replaced in that human expert. Either way, the decisions will not always be correct. In many cases, that is fine because the current situation is either that the decision cannot be made at all or that humans with a similar (or worse) success rate are making the decision. An important thing to establish early on in a project is what that success rate is and how much better your system needs to be. Work with the client to agree what the cost of wrong (or unmade) decisions is at the moment—that will help you establish some potential levels of reward if the project is a success. From that you can start to calculate potential return on investment. Once the reward is identified, you can be candid about the risks. The main one, of course, is that the data do not contain the information required to perform the task to an acceptable standard.

Once the client understand the risks and rewards you can agree how they are shared. The price of the risk is the cost of carrying out the project—that is how much will be lost if the data fail to deliver. The rewards are the reduced costs or increased revenue a successful project can bring. There are plenty of well publicised success stories in many sectors now, so finding some to cite as examples is a useful practise. In the most straight forward scenario, the client takes on all the risk and takes all the reward. They simply pay for the project to be carried out and accept it might not be successful. You can present a methodology (CRISP-DM for example) and agree on a maximum number of hours work. Payment for the project can be authorised based on an agreed outcome such as a report outlining the methodology that was followed and the results at each stage. The client should agree that this is taken as proof that the work was carried out in the event of a project not being successful.

Shared risk and reward models involve a reduced or even waived consultancy fee in return for a share of savings made if the project is successful. This is the data mining equivalent of the no win, no fee basis on which some legal cases are fought. Care is needed to define the measure of success as it can be difficult to isolate the exact monetary gains of a single project in a complex business operation. A simple method is to agree a target performance measure (perhaps on a data set that the client withholds until the agreed test is performed). The structure can ramp the fee up for every percentage point improvement in accuracy, for example. Another option is to allow the client to decide whether or not to deploy. If they say no, then you keep the models and they pay the reduced fee. If they want to deploy the model, then they share the successful outcome with a larger fee than your usual rate.

Selling your data science skills is getting a lot easier. More companies know that there is value in their data and more are willing to pay to explore how that value can be extracted. Ultimately, that is what you are mining the data for—value. That is what drives business decisions, whether it is buying your consultancy time or allocating company resource to the project you have proposed. In that respect, data mining projects are the same as any other, but I hope my advice in this section might help smooth your progress all the same.

1.16.1 Summary

This chapter described some of the theory and practise of building statistical models from data. You should now understand what it means for an algorithm to learn a model and how such models can be deployed to make automatic decisions. Two main methodologies were introduced. The CRISP-DM methodology describes the major steps of a data mining project from conception to deployment and provides a useful framework to follow. At the model building stage of that process, a methodology of model building, validation and testing is used to attempt to maximise the ability of the model to perform with the highest achievable accuracy from the given data.

The types of task that machine learning can perform were described, including prediction, classification, novelty detection, clustering and density estimation. A number of machine learning methods were described, including regression models, decision trees and neural networks. Each of them has a method for representing the model, processes for learning the model and controls on the complexity of the model.

I hope I have imparted some knowledge, but becoming a good machine learning practitioner (or, if you work for a new trendy start up, a *data ninja*) requires practice. You need to use the techniques on real data. Experiment with each of them, compare their strengths and weaknesses on different real world problems, and see for yourself what is involved. Luckily, there are plenty of resources out there to help you. Take some online courses, then download some data and build some models. Kaggle [] is a great place to start: it is a website full of data challenges set by a variety of organisations. You can download the data and compare the performance of your model with that of others who have tried before you. The software you need is free. You could make a start using Weka [] or dive straight in with Python libraries such as scikit-learn and NLTK. Youtube has lots of good tutorials on how to use these products.

The whole process can be addictive—the challenge of building a better model, of gaining a new insight, of applying a new technique and the satisfaction when it works. I hope you enjoy it as much as I do and I hope you are sufficiently intrigued by the potential in building increasingly intelligent machines.