MAS369/MAS61007

Machine Learning

Dr M. A. Juárez
m.juarez@sheffield.ac.uk

Dr Wei Xing
w.xing@sheffield.ac.uk

2023–24
(typeset: September 4, 2023)

Contents

1 What is Machine Learning?

We learn by being exposed through our senses to a lot of experiences and sensations; the aim of machine learning is similar, except we want to provide a computer with a lot of data, and hope that it will “learn” something about the nature of the data, possibly to make predictions of future data, or to make extra sense of the data it is given.

Wikipedia defines *machine learning* to be “a subset of artificial intelligence in the field of computer science that often uses statistical techniques to give computers the ability to "learn" (i.e., progressively improve performance on a specific task) with data, without being explicitly programmed.”

1.1 The problems of machine learning

Machine learning divides into many areas. Some of these will hardly be treated in this module, whereas others will be our main focus. Partly because of the slightly more statistical flavour of the required background, we will focus mostly on “supervised” learning, especially with a view to classification problems for data. However, we will also want to do some “unsupervised” learning, since this is an important part of Exploratory Data Analysis (EDA).

Before we introduce these areas further, perhaps now is a good point to comment on other areas of machine learning, which we won’t cover at all

Reinforcement learning Here, the computer has to learn a strategy, often for playing games. The computer makes random moves, is rewarded when moves have a successful outcome, and possibly penalised when moves have an unsuccessful outcome. The computer changes its strategy to play moves which are more often successful, with the eventual result, after playing many games, that the computer develops a highly successful strategy.

Successes have included various computer games, chess, go, and others.

Natural Language Programming In this area of machine learning, the computer analyses large bodies of words or texts (a *corpus* is the name of such a collection; *corpera* is the plural). While there are many techniques in the module which we can use (e.g., we might learn to classify email messages as spam or not spam), there are also techniques here which are used in their own right. For example, Naive Bayes is a fairly simple technique which can be used for classification, and which is used quite often in the natural language setting. We will develop some basic theory of neural networks later in the module, but there are specific neural network architectures which are used for sequential data, such as sequences of characters or words, and we shall say very little about these (if you are interested, you could look up methods like Recurrent Neural Networks, and Long Short-Term Memory Networks).

Topological Data Analysis I mention this here because it is a more mathematical framework in which to do data analysis. "Real life" machine learning practitioners don't tend to use these methods. But if you want to look at a way of considering data from a more pure mathematical point of view, this is a recent mathematical extension to machine learning theory.

Machine Learning is a fast-moving area; quite a lot of the progress isn't really related to mathematics, but to computational advances, and it is often the case that one doesn't need to understand the mathematics underlying the algorithms in order to use them. Nevertheless, the extra understanding is probably useful if you hope to develop new algorithms.

Let's now discuss our main areas: supervised learning and unsupervised learning.

1.1.1 Supervised learning

Here, we have data where every observation has outputs depending on inputs. Probably the most common problem is then to predict future outputs from future inputs. But there might be other problems also; we might not really care too much about future data, but might have more interest in the way that the output depends on the inputs—can we see from the data that the output only really depends on a small number of the inputs, or can we determine particular input variables that have a particularly strong impact on the output?

Here's a simple example. Suppose you take several modules, and you measure how much revision you did in total for each, and how much sleep you got the night before the exam. The results might be:

	hours of revision	hours of sleep	mark
Module 1	9	5	67
Module 2	12	7	79
Module 3	8	6	71
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Perhaps we want to work out an optimal revision and sleeping strategy for our next set of exams. Then we need to develop some way to predict an exam score from the other two variables. This is an example of a *regression* problem, where one or more numerical *output* variables depend on one or more *input* variables.

One natural way to do this is to do a linear regression on the data to try to make some prediction. But this might not be appropriate (the relationship is almost certainly not linear, for example), and we shall see more flexible ways to make predictions. We will see that there are a number of algorithms for this sort of problem; we could program a computer to try each of them and see which appeared to do best on our data set. Then we could hope to use that to make predictions for future data.

There are already some issues here—how do we measure the success of a method on our data set? how can we be confident that the method will perform equally well on future data? We will think about these questions in this module.

Alternatively, can we deduce from the data that actually the amount of sleep has no real effect on the final module mark?

Typically in machine learning, we let the computer do the thinking—we program some general models, but with unknown parameters, and the machine uses the data to make optimal choices of the parameters.

Of course, exams at Sheffield also have a Pass/Fail boundary, and it might be that we are more interested in knowing which combinations of values of input variables will be sufficient that one should expect to pass. Then we have a problem where the output variable is not numerical, but a class. This is a *classification* problem rather than a regression problem.

We could then try to make our machine learn how to predict combinations of values of the input variables which lead to a pass. In fact, we will focus even more strongly on the classification problem, partly because regression problems are well treated in other modules; also, most algorithms we shall give for regression also have counterparts which are used for classification, but the reverse is not always true, and there are methods we shall use which work best for classification problems.

1.1.2 Unsupervised learning

We'll talk a little less about this aspect of machine learning. But in unsupervised learning, there is no response variable we can use to look at individual pieces of data. Instead, we have to consider the data as a whole, rather than as individual observations, and to ask more general questions about the whole data set. For example:

1. If the data set is p-dimensional, is it close to a q-dimensional space, for some $q < p$? (Then we might be able to compress the data, and view it as q-dimensional.) How much information would be lost if we viewed it with fewer dimensions?
2. Can we view high dimensional data in such a way as to bring out similarities in observations?
3. Is there structure in the data—does it divide into more than one piece in a natural way?

As an example of the final question, we might have quite a few measurements of patients with a particular disease. We might wonder whether the disease has some subvariants, which can be distinguished by these observations? Are there variables which seem unimportant to this problem?

1.2 Computing

As the name suggests, machine learning is largely done by computer, although we'll try to look at some small "toy" examples which we can analyse by hand. You will need to use a computer quite a lot for this module.

Data science professionals seem largely to split into those who use Python and those who use R. Python is perhaps beginning to move into the ascendancy, but R is still dominant in academia, and some areas of statistics, while Python is generally the language of choice for machine learning practitioners.

However, R is the language of the MSc, and we won't be developing anything in this module for which Python is essential. I will try to present algorithms using both languages. Having said that, there are other languages too in which one can do data analysis. Some algorithms mentioned here are programmed in Excel, for example.

If you have your own computer, you should install at least one of R and Python , and it will do no harm to install both.

1.2.1 R

You should download R from https://cran.r-project.org/, and there's a nice interface developed by RStudio, from https://www.rstudio.com/products/rstudio/download/, which you should also download. The free version is perfectly adequate for us!

R is a piece of statistical software—it consists of some core commands, and some additional packages which are part of the base R which you can use to enhance basic programs. Additional to this are many many extra packages from other developers that can be downloaded to add extra functionality. There are a huge number of these, and most can be downloaded from within R; if, say, you want to explore a particular technique—eg neural networks—you can simply do an internet search for "R neural networks", and you will almost certainly find a package which you can then download from within an R session.

In fact, there is a package nnet which is part of base R, and if you want to use it in a calculation, you simply type `library(nnet)` at the start of the calculation.

If nnet doesn't meet your needs, you might look further at the search output, and discover a further package neuralnet which might appear to be useful. This isn't part of base R, so you would need to install it explicitly by typing `install.packages("neuralnet")` from within an R session. Then you can use it by typing `library(neuralnet)` as above.

The first lab sheet gives instructions on setting up R for this module. You may like to set up R now.

Although R is sufficient for this module, machine learning developments seem to be taking place more in Python. However, as we shall see later, there are ways to "bind" R to Python, so that some machine learning Python packages can be used within R.

1.2.2 Python

Python is a more general purpose computing language than R; it seems to be on the rise amongst data scientists, and is almost exclusively used in the machine learning community. It is a bit of an oversimplification to say that `Python` is used for programs and `R` for functions, but there's an element of truth in this—it does seem to be easier to write substantial programs in `Python`. However, there are aspects where it still seems (to me, at least) that `R` is in the lead—`R` has some very nice visualisation capabilities which are now fairly mature, whereas `Python` seems less stable in this regard, with more graphics packages being developed constantly.

`Python` has the same structure as `R`: some basic commands, with a library of core modules which you can import into a program for further use, and then many many further modules developed by others which can be installed (not quite so easily as for `R`, I think) for use in other programs.

You can install `Python` as a stand-alone program, from `https://www.python.org/`, but it is easier to install `Python` together with all the useful modules, and I recommend downloading and installing some version of the `Anaconda` distribution. Details are on the first lab sheet.

There was a change to `Python` between version 2 and version 3—not too major, but enough that programs in either version need a bit of tweaking to get them to work in the other—there are still a small handful of additional modules which have not been updated to work in `Python 3`, but everything we need in this module will work fine on `Python 3`.

1.2.3 Hardware

Many machine learning procedures are rather computationally intensive. If you are looking to buy a new computer with a view to doing machine learning, you might like to consider a gaming computer. With large datasets, you often need to do the same thing to each observation in your data, and if you can *vectorize* your program (which means operating on your data as a vector, with the same operation carried out to each in parallel), you can get huge speed-ups. Computers have CPUs (central processing units) for general purpose calculations; gaming computers also have GPUs (graphical processing units) whose role is to update the screen. CPUs have a few 'cores' which can do complicated things; GPUs have many far less sophisticated cores which can do only really simple things. But if you can get your programs running on the GPUs, with lots of simple things running in parallel at each step, your programs may run much faster.

At the moment, you should look for gaming computers with Nvidia cards, since the manufacturers have released code for machine learning.

Having said all that, computers are fast enough that even without a graphics card, you should be able to implement everything in this module on moderate data sets, even though the increased speed of a graphics card will allow you to work faster and therefore with larger datasets.

1.2.4 Programming

You might like to try programming the topics in this module yourself. You will get a lot out of this, and it is extremely instructive. In fact, this is the approach taken by the book of Grus mentioned below.

But your programs will be extremely slow! Commands available from within `R` and Python make complete use of vectorisation; are programmed in lower-level and much faster languages (e.g., C); and are completely optimised for speed and memory. They will work much faster than any program you write.

So you won't need to do any programming, since all the topics are already programmed extremely efficiently by others; it will be more a question of understanding the underlying technique, and applying the methods using pre-existing `R` or `Python` commands.

1.2.5 These notes

The material in this course has been checked under the following versions:

`R`	4.2.1 (September 2022)
`Python`	3.7.3 (July 2019)

1.3 Books

You do not need to buy any books! Hopefully these notes are a good start; in this topic, the internet is a great resource, and there are a lot of nice instructional videos as well as blog pages etc. But, should you want to look through texts, there are some decent options.

For books with a statistical background, I recommend:

- G.James et al. (2013), *An Introduction to Statistical Learning*, Springer, a download is at `http://web.stanford.edu/~hastie/pub.htm`
- T.Hastie et al. (2011), *The Elements of Statistical Learning*, 2nd. ed., Springer, a download is at `http://web.stanford.edu/~hastie/pub.htm`

The first is an introduction to the second; both are free to download (although you can buy paper copies), and are strongly recommended. The first book introduces the material with R code, while the second, at a level beyond this course, is theoretical. These two books are probably the ones I most recommend for statisticians interested in machine learning—especially since they are free!

For other fairly introductory books with a statistical flavour on analysing high-dimensional data, try:

- T.Cox (2005), *An Introduction to Multivariate Analysis*, Arnold
- C.Chatfield, A.J.Collins (1980), *Introduction to Multivariate Analysis*, Chapman and Hall

Nice introductions specifically using `R` are:

- B.Everitt (2005), *An R and Splus Companion to Multivariate Analysis*, Springer, see also `http://biostatistics.iop.kcl.ac.uk/publications/everitt`
- H.Wickham, G.Grolemund (2016), *R for Data Science*, O'Reilly

A more advanced (but still excellent) treatment with `R` (or rather its precursor `S`) is:

- W.Venables, B.Ripley (2002), *Modern Applied Statistics with S*, 4th. ed., Springer, support material available at http://www.stats.ox.ac.uk/pub/MASS4

There are similarly a number of books on machine learning with `Python`. How about

- A. Géron (2017), *Hands-on Machine Learning with Scikit-Learn and TensorFlow*, O'Reilly
- J. Grus (2014, 2nd. ed. 2019), *Data Science from Scratch*, O'Reilly
- W. McKinney (2012, 2nd ed., 2017), *Python for Data Analysis*, O'Reilly
- J. VanderPlas (2016), *Python Data Science Handbook*, O'Reilly

VanderPlas's book gives an excellent introduction to all the packages in the Python data stack. McKinney is the author of the `pandas` package, which is an attempt to bring R's notion of a "data frame" into Python. His book is best for `pandas` (unsurprisingly), but I think VanderPlas is a better overall introduction.

Grus's book is a nice introduction to machine learning in `Python`. The second edition is written in `Python` 3.6 (the first was in `Python` 2.7). Simple `Python` scripts are given for many machine learning algorithms, although, for the reasons mentioned above, you wouldn't want to use any of them for serious data sets.

Géron's book is slightly more advanced, and leads the reader into deep learning, which we will merely touch on.

2 Notation

This chapter introduces some notation for the future, and develops some of the prerequisite background knowledge. I won't lecture too much of this, but if you aren't sure about this knowledge, do read the relevant sections.

2.1 Basic notation

We deal with *observations* $\boldsymbol{x} \in \mathbb{R}^p$. It will generally be convenient to store the observations in a matrix, $X \in \mathcal{M}_{n\times p}$. Not only is this a convenient form to store the data, but matrix methods are going to play an important role in the module; manipulating the data will be easiest if it is already in matrix form. The i-th observation (row) of X then is $\boldsymbol{x}_i = (x_{i1}, \ldots, x_{ip})$.

If $\boldsymbol{x}$ is a column vector, write $\boldsymbol{x}'$ for the transpose of $\boldsymbol{x}$; it is a row vector. We use the same notation for matrices.

X contains the data. However, there is confusion throughout sources between whether to treat X or its transpose, the $p \times n$ matrix X', as the "data matrix" (obviously it doesn't really matter, but we do need to fix a choice so that our formulae work!). Partly because of the way `R` stores the information, as a *data frame* (and imitated by the `Python` module `pandas`), we will refer to X as the *data matrix*. Thus the data matrix is the $n \times p$ matrix:

$$X = \begin{pmatrix} x_{11} & \ldots & x_{1p} \\ x_{21} & \ldots & x_{2p} \\ \vdots & \vdots & \vdots \\ x_{n1} & \ldots & x_{np} \end{pmatrix},$$

in which each row consists of a multivariate observation. (As already remarked, this choice is not universal in the literature, and you should be aware of this if you consult any books!)

Define the *sample mean vector* as $\bar{\boldsymbol{x}} = (\bar{x}_1, \ldots, \bar{x}_p)$, with

$$\bar{x}_j = \frac{1}{n}\sum_{i=1}^{n} x_{ij}, \qquad j = 1, \ldots, p.$$

As usual, this sample mean is a sort of *location* of the data points.

There is a very succinct way to write $\bar{\boldsymbol{x}}$: it is equal to $\frac{1}{n}\mathbf{1}'X$, where $\mathbf{1} \in \mathbb{R}^n$ has 1 in all its entries. (You should try to understand this succinct definition; the easier you find this sort of matrix manipulation, the easier the course will become! After all, we are really interested in the sample mean of the data matrix, i.e., the mean for each variable, and X has one column for each variable; the sample mean ought to as well).

We also define the *(sample) variance* (or *variance-covariance matrix* or *covariance matrix*) as

$$S = \mathbb{V}[X] = \frac{1}{n-1}(X - \bar{X})'(X - \bar{X}), \tag{2.1}$$

where $\bar{X} = (\bar{\boldsymbol{x}}, \ldots, \bar{\boldsymbol{x}})$ is the $n \times p$ matrix with all columns equal to $\bar{\boldsymbol{x}}$. (Again, some authors use $\frac{1}{n}$ in place of $\frac{1}{n-1}$; the justification for using $\frac{1}{n-1}$ is the same as in the univariate case, as we shall explain later.)

Exercise 2.1 Use (2.1) to check that

$$s_{ij} = \frac{1}{n-1}\sum_{k=1}^{n}(x_{ki} - \bar{\boldsymbol{x}}_i)^2.$$

Notice that S is a $p \times p$ matrix whose diagonal entry s_{ii} is the *sample variance* of the ith variable, and the entry s_{ij} for $i \neq j$ is the *sample covariance* between the ith and jth variables. In particular, S is symmetric (this should be clear from (2.1)).

Provided none of the variables is a linear combination of any of the others (so the columns are *linearly independent*), S will be non-singular (i.e., invertible, or equivalently have non-zero determinant) and will be positive definite.

The variance matrix S can also be written as

$$\begin{aligned} S &= \frac{1}{n-1}\sum_{i=1}^{n}(\boldsymbol{x}_i - \bar{\boldsymbol{x}})'(\boldsymbol{x}_i - \bar{\boldsymbol{x}}) \\ &= \frac{1}{n-1}\left(\sum_{i=1}^{n}\boldsymbol{x}_i'\boldsymbol{x}_i - n\bar{\boldsymbol{x}}'\bar{\boldsymbol{x}}\right). \end{aligned}$$

For the first equality, note that $X - \bar{X}$ is a matrix with columns $\boldsymbol{b}_i = \boldsymbol{x}_i - \bar{\boldsymbol{x}}$, and so $B = X - \bar{X}$ has columns $(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n)$, and $B'B = \sum_{i=1}^{n}\boldsymbol{b}_i'\boldsymbol{b}_i$.

The second equality follows directly from multiplying out the terms in the sum sign (keeping the ordering the same, and taking the transpose inside the brackets, noting that $\bar{\boldsymbol{x}}$ is a constant, and the sum of the $\boldsymbol{x}_i$ in $n\bar{\boldsymbol{x}}$).

We will also use the *sample correlation matrix* R, which is simply the same as the variance matrix, but with the entries scaled so that

$$r_{ij} = \frac{s_{ij}}{\sqrt{s_{ii}s_{jj}}}. \tag{2.2}$$

If L denotes the diagonal matrix whose entries are $s_{11}, \ldots, s_{pp}$, and $L^{1/2}$ is then the diagonal matrix with entries $\sqrt{s_{11}}, \dots, \sqrt{s_{pp}}$, then

$$S = L^{1/2} R L^{1/2}.$$

Let's quickly make some comments on the correlation matrix R. From the (2.2), it is easy to see:

1. R is a symmetric $p \times p$ matrix;
2. $r_{ii} = 1$ for all i;
3. $-1 \le r_{ij} \le 1$ for all i, j. As usual, if $r_{ij} > 0$, then there is a tendency for variables i and j to be larger at the same time, whereas if $r_{ij} < 0$, there is a tendency for variable j to be smaller when variable i is larger, and vice versa. Geometrically, r_{ij} is the cosine of the angle between the vectors of deviations of observations of the ith and jth variables from the mean.

There are a lot of general results about real symmetric matrices, which will apply to both S and R. So suppose A is a real symmetric $p \times p$ matrix.

1. The eigenvalues of A are all real;
2. There are always p linearly independent eigenvectors (even if some of the eigenvalues are the same);
3. Further, we can always even find p *orthogonal* eigenvectors for A.

Students needing a reminder about eigenvalues and eigenvectors should consult the internet or other sources.

Abbreviating data samples with matrices has a lot of advantages. Not only does it allow a concise way to refer to the data, but it turns out—and you will notice examples above already—that we can use techniques from matrix algebra to develop our methods. The theory of matrices and vectors is, however, a huge subject. It is part of most undergraduate courses, and we will be assuming it in this module, so if you haven't come across the ideas before, keep a copy of a linear algebra text nearby.

2.2 More sample variance and correlation matrices

Here is a result which justifies the use of the scalar factor $\frac{1}{n-1}$ (some texts use $\frac{1}{n}$):

Proposition 2.1 The sample mean and variance of independent observations are unbiased estimates for the population mean and variance.

Proof This is a direct generalisation of the standard proof in the univariate case. So we suppose that the data, $X = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\}$ is generated as i.i.d. random variables with mean $\boldsymbol{\mu} \in \mathbb{R}^p$ and variance $\Sigma \in \mathcal{M}_{p\times p}$. Then

$$\begin{aligned}
\mathbb{E}[\bar{\boldsymbol{x}}] &= \mathbb{E}\big[\tfrac{1}{n}\boldsymbol{x}_1 + \cdots + \tfrac{1}{n}\boldsymbol{x}_n\big] \\
&= \tfrac{1}{n}\,\mathbb{E}[x_1] + \cdots + \tfrac{1}{n}\,\mathbb{E}[x_n] \\
&= \tfrac{1}{n}\boldsymbol{\mu} + \cdots + \tfrac{1}{n}\boldsymbol{\mu} \\
&= \boldsymbol{\mu}.
\end{aligned}$$

For the variance, we again argue as in the univariate case. Firstly, we see that if $\boldsymbol{x}_i$ is a vector with $\mathbb{E}[\boldsymbol{x}_i] = \boldsymbol{\mu}$ and variance $\Sigma = \mathbb{E}[(\boldsymbol{x}_i - \mu)'(\boldsymbol{x}_i - \boldsymbol{\mu}))$, then multiplying this out gives $E(\boldsymbol{x}_i'\boldsymbol{x}_i) - \boldsymbol{\mu}'\boldsymbol{\mu} = \Sigma$, and so

$$\mathbb{E}[\boldsymbol{x}_i'\boldsymbol{x}_i] = \boldsymbol{\mu}'\boldsymbol{\mu} + \Sigma. \tag{2.3}$$

In particular, $\bar{\boldsymbol{x}} = \frac{1}{n}(\boldsymbol{x}_1 + \cdots + \boldsymbol{x}_n)$ also has expected value $\boldsymbol{\mu}$, and one easily shows that $\mathbb{E}[\bar{\boldsymbol{x}} - \boldsymbol{\mu})'(\bar{\boldsymbol{x}} - \boldsymbol{\mu})] = \frac{1}{n}\Sigma$, so we deduce that

$$\mathbb{E}[\bar{\boldsymbol{x}}'\bar{\boldsymbol{x}}] = \boldsymbol{\mu}'\boldsymbol{\mu} + \frac{1}{n}\Sigma. \tag{2.4}$$

Now

$$S = \frac{1}{n-1}\left(\sum_{i=1}^{n} \boldsymbol{x}_i'\boldsymbol{x}_i - n\bar{\boldsymbol{x}}'\bar{\boldsymbol{x}}\right),$$

so

$$\begin{aligned}
\mathbb{E}[S] &= \frac{1}{n-1}\left(\sum_{i=1}^{n} \mathbb{E}[\boldsymbol{x}_i'\boldsymbol{x}_i] - n\,\mathbb{E}[\bar{\boldsymbol{x}}'\bar{\boldsymbol{x}}]\right) \\
&= \frac{1}{n-1}\left(n\big(\boldsymbol{\mu}'\boldsymbol{\mu} + \Sigma\big) - n\left(\boldsymbol{\mu}'\boldsymbol{\mu} + \frac{1}{n}\Sigma\right)\right) \\
&= \frac{1}{n-1}(n-1)\Sigma \\
&= \Sigma,
\end{aligned}$$

as required.

Let's also list a couple of simple (but useful) results of matrix manipulations. We will be applying matrix operations freely, and will need this sort of result often.

Let's imagine that the columns of our data matrix correspond to variables $v_1, \ldots, v_p$. We might want to get some single number to summarise all this data, and one way to do this is to take some weighted average of the variables. For example, if we had two variables, `height` and `weight` of a person, we might measure their *size* as an of average of these. But this would depend on the units we used to measure the variables, and this is unsatisfactory; we can solve this using a weighted linear combination. So if $w_1, \ldots, w_p$ are numbers, we can associate to each observation the single quantity

$$w_1 \times \text{(value of first variable)} + \cdots + w_p \times \text{(value of } p\text{-th variable)}.$$

For the i-th observation, the values of the variables in X are $x_{1i}, \ldots, x_{pi}$, so the ith observation gets the quantity

$$w_1 x_{i1} + \cdots + w_p x_{ip}.$$

Thinking about how matrix multiplication works, this is the i-th entry of the vector $X\boldsymbol{w}$. Consequently, the linear combination of the vectors for all the observations is simply the vector $X\boldsymbol{w}$, and this is the weighted linear combination of the variables.

Lemma 2.1 If $\boldsymbol{w} \in \mathbb{R}^p$ is any vector, then $\mathbb{V}[X\boldsymbol{w}] = \boldsymbol{w}'\,\mathbb{V}[X]\boldsymbol{w} = \boldsymbol{w}'S\boldsymbol{w}$.

Proof Write $Y = X\boldsymbol{w}$. This follows directly from the expression for $\mathbb{V}[Y]$, putting $\boldsymbol{y}_i = X_i\boldsymbol{w}$ etc.:

$$\begin{aligned}
\mathbb{V}[Y] &= \frac{1}{n-1}\sum_{i=1}^{n}(\boldsymbol{y}_i - \bar{\boldsymbol{y}})'(\boldsymbol{y}_i - \bar{\boldsymbol{y}}) \\
&= \frac{1}{n-1}\sum_{i=1}^{n}(X_i\boldsymbol{w} - \bar{\boldsymbol{x}}\boldsymbol{w})'(X_i\boldsymbol{w} - \bar{\boldsymbol{x}}\boldsymbol{w}) \\
&= \frac{1}{n-1}\sum_{i=1}^{n}\boldsymbol{w}'(\boldsymbol{x}_i - \bar{\boldsymbol{x}})'(\boldsymbol{x}_i - \bar{\boldsymbol{x}})'\boldsymbol{w} \\
&= \boldsymbol{w}'S\boldsymbol{w}.
\end{aligned}$$

So the lemma just proved explains how the variance matrix of the linear combination (a scalar) is related to the variance matrix of the whole dataset (a $p \times p$-matrix).

We can take several different linear combinations; taking q different combinations leads to taking a $p \times q$-matrix A, and then we have an analogous result for these combinations:

Lemma 2.2 If A is any $p \times q$ matrix, then $\mathbb{V}[XA] = A'\,\mathbb{V}[X]A = A'SA$ (a $q \times q$-matrix).

Proof Similar to the last result.

3 Mathematical background

This chapter is provided as background, and I won't lecture it in any detail; I will, however, draw on this material during the module. If there's anything that you aren't sure about, you might like to revise it now! Some proofs have been included for completeness, but you will never need to know these—the results may be used later, however.

You might like to recall Chapter 2 for a reminder about the notation.

3.1 Linear algebra

Abbreviating data samples with matrices has a lot of advantages. Not only does it allow a concise way to refer to the data, but it turns out—and you will notice examples above already—that we can use techniques from matrix algebra to develop our methods.

However, linear algebra—the theory of matrices and vectors—is a huge subject. There are many books on the subject, but one written by the former lecturer of this module, Nick Fieller, called "Basics of Matrix Algebra for Statistics with R" (Chapman and Hall, 2015), might be particularly valuable. It is part of most undergraduate courses, and we will be assuming it in this module, so if you haven't come across the ideas before, keep a copy of a linear algebra text nearby.

There is a section later in this chapter on eigenvectors and eigenvalues, which will be very useful, but we give a summary of the ideas of some other relevant concepts here too.

Given three vectors $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{w}$ in $\mathbb{R}^3$, there will usually not be any plane that contains all three vectors. This means that we can get from the origin to any point by travelling a certain (possibly negative) distance in the direction of $\boldsymbol{u}$, then a certain distance in the direction of $\boldsymbol{v}$, then a certain distance in the direction of $\boldsymbol{w}$. The case where $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{w}$ all lie in a common plane will have special geometric significance in any purely mathematical problem, and will often have special physical significance in applied and engineering problems. Three vectors lie in the same plane occurs when one of the vectors, $\boldsymbol{w}$ say, can be written in terms of the other two, so that $\boldsymbol{w} = \alpha\boldsymbol{u} + \beta\boldsymbol{v}$ for some real numbers α and β. That is, there is a *relationship* between $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{w}$. Alternatively, we say that $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{w}$ are *linearly dependent*, and otherwise we say they are *linearly independent*. Along the same lines, we say that an expression $\boldsymbol{w} = \alpha\boldsymbol{u} + \beta\boldsymbol{v}$ is a *linear combination* of $\boldsymbol{u}$ and $\boldsymbol{v}$. The same terminology applies to anything where we can add two terms and multiply each by a number, so for example, $2x + 3x^2$ is a linear combination of x and x^2, and $2\sqrt{2} - \sqrt{3}$ is a linear combination of $\sqrt{2}$ and $\sqrt{3}$.

More formally:

Definition 3.1 Three vectors $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{w}$ in $\mathbb{R}^3$ are *linearly independent* if whenever $\alpha\boldsymbol{u} + \beta\boldsymbol{v} + \gamma\boldsymbol{w} = 0$, then $\alpha = \beta = \gamma = 0$. If a non-trivial solution exists, then the vectors are *linearly dependent*.

You will probably have seen these ideas when solving linear equations. If there is a linear dependency, then one equation can be written in terms of the others, so is redundant. As an example, let's take:

$$\begin{aligned} x + y + z &= 2 \\ 2x + 3y - z &= 1 \\ x + 4z &= 5, \end{aligned}$$

where one can see that the third equation is got by tripling the first and subtracting the second. Each of these equations represents a plane in three dimensions, and asking for a simultaneous solution is equivalent to finding the intersection of the three planes. Usually, when you write down three "random" planes, you will get a single point of intersection. But when we write down three planes corresponding to dependent equations, we only really have two equations, and these will generally intersect in a line.

More generally, we can see that whenever the left-hand sides of the three equations are linearly dependent, we should be able to take some combination which makes the left-hand side of one of the equations all 0s. So this equation now becomes $0x + 0y + 0z = ?$; if the right-hand constant is 0, then we basically end up with two equations in three variables, and expect *infinitely* many solutions, while if the right-hand constant is non-zero, the equation can never hold, and there are *no* solutions.

In general, given n equations in n unknowns, we have an $n \times n$-matrix of coefficients. If the rows of this matrix (i.e., the vectors of coefficients of the simultaneous equations) are linearly independent, then there is a unique solution. If, however, the rows of the matrix are linearly dependent, this means that we can get the left-hand side of one equation by combining other equations, and either the equation is then redundant (meaning that there are infinitely many solutions) or contradictory (so that there are no solutions).

Essentially, you should get the idea that if we have n equations, there may really be n distinct ones (when they are linearly independent), or there may in fact be some redundancy. If there are really just r equations, then the system of n equations is said to have *rank* r. In the same way, given n vectors, if we can write them as linear combinations of a smaller set of r vectors, then we should think of the set as being at most r dimensional, and the *rank* is the actual dimension of the space of all linear combinations of the vectors.

Two things we will use from time to time is that if $\boldsymbol{x} = (x_1, \ldots, x_n)$ is a vector of length n, then

- $\boldsymbol{x}'\boldsymbol{x}$ is a scalar:
$$\begin{pmatrix} x_1 & \cdots & x_n \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = x_1^2 + \cdots + x_n^2$$
and this gives the square of the Euclidean length of $\boldsymbol{x}$.

- $\boldsymbol{x}\boldsymbol{x}'$ is an $n \times n$-matrix of rank 1. Indeed,
$$\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \begin{pmatrix} x_1 & \cdots & x_n \end{pmatrix} = \begin{pmatrix} x_1^2 & \cdots & x_1x_n \\ \vdots & \ddots & \vdots \\ x_1x_n & \cdots & x_n^2 \end{pmatrix},$$
so that the ijth entry is just x_ix_j. In particular, every row is a multiple of $(x_1, \ldots, x_n)$ and every column is a multiple of $\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$ (which both imply that the matrix has rank 1).

3.2 Basic properties of eigenvectors and eigenvalues

Let A be a real $p \times p$ matrix.

Definition 3.2 The *eigenvalues* of A are the roots of the degree p polynomial (the *characteristic polynomial*) in $\boldsymbol{\lambda}$ given by $q(\boldsymbol{\lambda}) = \det(A - \boldsymbol{\lambda}\mathrm{I}_p) = 0$.

Remark 3.1 Suppose these eigenvalues are $\lambda_1, \ldots, \lambda_p$. Then $q(\boldsymbol{\lambda}) = \prod_i(\lambda_i - \boldsymbol{\lambda})$. Comparing the coefficients of $\boldsymbol{\lambda}^{p-1}$ in these two expressions for $q(\boldsymbol{\lambda})$ gives
$$\sum_{i=1}^{p} \lambda_i = \mathrm{tr}(A).$$
Further, putting $\boldsymbol{\lambda} = 0$ gives $\prod_{i=1}^{p} \lambda_i = \det(A)$.

Definition 3.3 Suppose that λ_i is an eigenvalue of A. Since $A - \lambda_i\mathrm{I}_p$ is singular (i.e., has zero determinant), there is (at least one) nonzero vector $\boldsymbol{x}_i$, called an *eigenvector* of A, such that $(A - \lambda_i)\boldsymbol{x}_i = 0$, i.e., $A\boldsymbol{x}_i = \lambda_i\boldsymbol{x}_i$. (Note that if $\boldsymbol{x}_i$ is an eigenvector, then so is any nonzero multiple of it, so we sometimes want to normalise eigenvectors so that a particular entry is 1, or it has length 1 etc.)

Generally, if λ_i is an eigenvalue of A with multiplicity m_i, the maximal number of linearly independent eigenvectors of A can be anywhere between 1 and m_i. In statistical applications, this is generally only an issue for the eigenvalue $\lambda = 0$ (for example, if there are not enough observations that the $p \times p$ matrix arising can possibly have rank p).

The following fact is rather useful:

Fact If A is a real symmetric $p \times p$ matrix, then there are always p linearly independent eigenvectors.

This is a well-known result from matrix theory. Let's quickly outline the proof, which will work by induction on p (the result is trivial for $p = 1$):

1. Let $\boldsymbol{x}_1$ denote an eigenvector of A of length 1, with eigenvalue λ_1.

   For this, recall that the characteristic polynomial has a complex root λ_1, and that there is an eigenvector with eigenvalue λ_1; we scale it to have length 1. The argument of proposition 3.2 below tells us that λ_1 is real, and then x_1 is easily seen to be real also.

2. Let $x_1, y_2, \ldots, y_p$ denote any orthogonal set of real vectors of length 1, and let M denote the matrix with these as columns. Then M is an orthogonal matrix, $M^{-1}AM = M'AM$ is again real, and easily seen to be symmetric. Further, we calculate $M^{-1}AM(e_1) = M'A(x_1) = M^{-1}(\lambda_1 x_1) = \lambda_1(M^{-1}x_1) = \lambda_1 e_1$, and this means that the first column of $M^{-1}AM = M'AM$ is just $\begin{pmatrix} \lambda_1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$.

3. Since $M'AM$ is symmetric, we see that

$$M'AM = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & A_1 & \\ 0 & & & \end{pmatrix}$$

   where A_1 is a $(p-1) \times (p-1)$ real symmetric matrix. By induction, A_1 has $p-1$ eigenvectors. Then it is easy to see that e_1 and the $p-1$ eigenvectors of A_1 (enlarged to length p by putting a 0 at the top) are a set of p eigenvectors of $M'AM$; applying M to each of these gives a set of p eigenvectors of A.

Much more is known, and we'll derive what we need in the next page or so.

We will often want to transform our variables by means of some linear transformation, and will need to know how eigenvalues and eigenvectors of the variance matrix (for example) are affected by this. As we shall see, transforming the variables by a matrix M leads to a conjugation of the sample variance matrix by M.

Proposition 3.1 If M is any $p \times p$ nonsingular square matrix, then A and MAM^{-1} have the same eigenvalues. Furthermore, if $\boldsymbol{x}_i$ is an eigenvector of A with eigenvalue λ_i, then Mx_i is an eigenvector of MAM^{-1} with eigenvalue λ_i.

Proof The first statement follows from the equality $\det(A - \boldsymbol{\lambda}\mathrm{I}_p) = \det(MAM^{-1} - \boldsymbol{\lambda}\mathrm{I}_p)$. For this, recall that $\det(AB) = \det(A)\det(B)$, and, in particular,

$$\begin{aligned}\det(MAM^{-1} - \lambda\mathrm{I}_p) &= \det(MAM^{-1} - \lambda MM^{-1})\\ &= \det(M(A - \boldsymbol{\lambda}\mathrm{I}_p)M^{-1})\\ &= \det(M)\det(A - \boldsymbol{\lambda}\mathrm{I}_p)\det(M^{-1})\\ &= \det(A - \boldsymbol{\lambda}\mathrm{I}_p).\end{aligned}$$

If $A\boldsymbol{x}_i = \lambda_i\boldsymbol{x}_i$, then $(MAM^{-1})(M\boldsymbol{x}_i) = MA\boldsymbol{x}_i = M(\lambda_i\boldsymbol{x}_i) = \lambda_i(M\boldsymbol{x}_i)$, so the eigenvectors of MAM^{-1} are $M\boldsymbol{x}_i$.

We have already stated one useful result about real symmetric matrices. Here is another.

Proposition 3.2 Suppose that A is a real symmetric matrix, i.e., $A = A'$. Then the eigenvalues of A are real.

Proof Indeed, if $A\boldsymbol{x} = \boldsymbol{\lambda}\boldsymbol{x}$, then $\bar{\boldsymbol{x}}'A\boldsymbol{x} = \boldsymbol{\lambda}\bar{\boldsymbol{x}}'\boldsymbol{x}$; applying transposition, and taking complex conjugation implies that also $\bar{\boldsymbol{x}}'A\boldsymbol{x} = \bar{\boldsymbol{\lambda}}\bar{\boldsymbol{x}}'\boldsymbol{x}$. As $\boldsymbol{x} \neq 0$, we get $\boldsymbol{\lambda} = \bar{\boldsymbol{\lambda}}$.

In the same way that we can measure lengths of vectors in $\mathbb{R}^2$ and $\mathbb{R}^3$ using (essentially) Pythagoras's Theorem, giving $\|(a, b)\| = \sqrt{a^2 + b^2}$, there is an analogous statement in $\mathbb{R}^n$:

$$\|(x_1, \ldots, x_n)'\| = \sqrt{x_1^2 + \cdots + x_n^2}.$$

In fact, this extends to the notion of an *inner product* on $\mathbb{R}^n$:

$$\langle x, y\rangle = x_1y_1 + \cdots + x_ny_n,$$

so that $\|\boldsymbol{x}\| = \sqrt{\langle \boldsymbol{x}, \boldsymbol{x}\rangle}$. The notation $\langle\cdot, \cdot\rangle$ is traditional in pure mathematics, but statisticians tend not to use it—instead, notice that if $\boldsymbol{x} = \begin{pmatrix} x_1\\ \vdots\\ x_n\end{pmatrix}$ and $\boldsymbol{y} = \begin{pmatrix} y_1\\ \vdots\\ y_n\end{pmatrix}$, then

$$\boldsymbol{x}'\boldsymbol{y} = x_1y_1 + \cdots + x_ny_n;$$

we shall just use the notation $\boldsymbol{x}'\boldsymbol{y}$ in this module.

Just as the existence of the dot product in $\mathbb{R}^3$ allows one to define angles etc., between vectors, the same holds in $\mathbb{R}^n$; in particular, we say that two vectors $\boldsymbol{x}$, $\boldsymbol{y}$ are *orthogonal* if $\boldsymbol{x}'\boldsymbol{y} = 0$. The last main result we need about real symmetric matrices is the following:

Proposition 3.3 Suppose that A is a real symmetric matrix. If λ and $\boldsymbol{\mu}$ are distinct eigenvalues and $\boldsymbol{x}$ and $\boldsymbol{y}$ are corresponding eigenvectors, then $\boldsymbol{x}$ and $\boldsymbol{y}$ are orthogonal.

Proof Indeed, we have $Ax = x$ and $A\boldsymbol{y} = \mu\boldsymbol{y}$. So $\boldsymbol{y}'A\boldsymbol{x} = \lambda\boldsymbol{y}'\boldsymbol{x}$ and $\boldsymbol{x}'A\boldsymbol{y} = \mu\boldsymbol{x}'\boldsymbol{y}$. Taking transposes of the latter gives $\boldsymbol{y}'A\boldsymbol{x} = \mu\boldsymbol{y}'\boldsymbol{x}$. As $\lambda \neq \mu$, we get $\boldsymbol{y}'\boldsymbol{x} = 0$.

Let's summarise the main properties of real symmetric matrices:

1. If A denotes a real symmetric $p \times p$ matrix, then A has p linearly independent eigenvectors.
2. All the eigenvalues of A are real.
3. Eigenvectors corresponding to distinct eigenvalues are orthogonal.

The (sample) variance matrix will be the main focus for many of our techniques, and the properties above will often be important.

Corollary 3.1 If A is a real symmetric $p \times p$ matrix, and if X denotes the matrix whose columns are normalised eigenvectors of A, then X is an orthogonal matrix (i.e. $XX' = \mathrm{I}$, or $X' = X^{-1}$).

Proof This simply follows from the observation that the columns of X have length 1, and are orthogonal.

Recall that if A is any $p \times p$ matrix with p linearly independent eigenvectors (e.g., any matrix with p distinct eigenvalues, or any real symmetric matrix), then if X denotes the matrix whose columns are eigenvectors of A, then $X^{-1}AX = \Lambda$, where Λ is the diagonal matrix whose diagonal entries are the eigenvalues. (Indeed, this is easily verified by considering how matrix multiplication works, and computing AX; the product is evaluated by multiplying A by each column in turn, and since each column is an eigenvector, we easily find $AX = X\Lambda$.)

The decomposition $X^{-1}AX = \Lambda$ is known as the *spectral decomposition*, and can easily be seen to be equivalent to

$$A = \lambda_1 \boldsymbol{x}_1\boldsymbol{x}_1' + \cdots + \lambda_p \boldsymbol{x}_p\boldsymbol{x}_p',$$

where $\boldsymbol{x}_i$ is an eigenvector of A with eigenvalue λ_i.

Corollary 3.2 Positive definite real symmetric matrices (i.e., real symmetric matrices with positive eigenvalues) possess positive definite real symmetric square roots.

Proof If A is a positive definite real symmetric $p \times p$ matrix with orthogonal eigenvectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_p$ corresponding to eigenvalues $\lambda_1, \ldots, \lambda_p$, we know that $X^{-1}AX = \Lambda$, i.e., that $A = X\Lambda X^{-1}$. Then we pick $\Lambda^{1/2}$ to be the diagonal matrix whose entries are the positive square roots of each λ_i. Put $B = X\Lambda^{1/2}X^{-1}$; then it is easy to see that $A = B^2$. Since $X' = X^{-1}$ (if we take the eigenvectors to be normalised), we can also write $B = X\Lambda^{1/2}X'$.

We can apply this because of the following result:

Lemma 3.1 The eigenvalues of the variance matrix S are non-negative.

Proof Suppose λ is an eigenvalue of S, with corresponding eigenvector $\boldsymbol{x}$. Then $S\boldsymbol{x} = \lambda\boldsymbol{x}$. It follows that $\boldsymbol{x}'S\boldsymbol{x} = \lambda\boldsymbol{x}'\boldsymbol{x}$; it is easy to see that $\boldsymbol{x}'S\boldsymbol{x}$ is a non-negative scalar (why?), and that $\boldsymbol{x}'\boldsymbol{x}$ is a positive scalar.

Summarising all the discussion so far, the variance matrix has a unique positive definite symmetric square root. We won't need this often.

One last thing we will eventually are that matrices of rank 1 (like those of the form $\boldsymbol{x}\boldsymbol{x}'$ mentioned above) have only 1 nonzero eigenvalue. Indeed, this is an equivalent definition of the rank of a square matrix: the rank of a matrix A is the number of nonzero eigenvalues (but the definition above, as the maximal number of linear independent rows or columns, works also for non-square matrices).

3.3 Differentiating with respect to vectors

As you will have gathered already, many techniques in the module involve matrices and vectors. Often we will be interested in the best choice of vector to optimise some function. Classically, you know that to optimise a function of one variable, we try to differentiate it, and set the result to zero. We will need something similar for vectors.

Suppose that $\boldsymbol{x} = \big(x_1, \ldots, x_p\big) \in \mathbb{R}^p$. We say that $f = f(\boldsymbol{x})$ is a *scalar function* of $\boldsymbol{x}$ if it is a function depending on $\boldsymbol{x}$, but which produces a number as the output. For example, we could take $f(x) = \boldsymbol{x}'\boldsymbol{x}$, the square length of $\boldsymbol{x}$.

Then we define the *derivative* of f with respect to $\boldsymbol{x}$, written $\nabla f = \frac{\partial f}{\partial \boldsymbol{x}}$, as the vector $(\frac{\partial f}{\partial x_1} \cdots \frac{\partial f}{\partial x_p})'$ (assuming all these partial derivatives exist).

The most useful special case is given by the following result.

Lemma 3.2 Suppose that S is a symmetric $p \times p$-matrix, and that $f(\boldsymbol{x}) = \boldsymbol{x}'S\boldsymbol{x}$. Then $\nabla f = \frac{\partial f}{\partial \boldsymbol{x}} = 2S\boldsymbol{x}$.

Proof Write $S = (s_{ij})$. Then

$$f(\boldsymbol{x}) = \boldsymbol{x}'S\boldsymbol{x} = \sum_{i=1}^{n}\sum_{j=1}^{n} s_{ij}x_ix_j. \tag{3.1}$$

We can work out $\frac{\partial f}{\partial x_k}$ by the product rule. We'll consider 4 cases, depending whether i or j is equal to k.

- If $i \neq k$, and $j \neq k$, then the term in (3.1) coming from i and j differentiates to 0:

$$\frac{\partial(s_{ij}x_ix_j)}{\partial x_k} = 0.$$

- If $i = k$, and $j \neq k$, then the term in (3.1) coming from i and j differentiates as:
$$\frac{\partial(s_{kj}x_k x_j)}{\partial x_k} = s_{kj}x_j.$$
- If $i \neq k$, and $j = k$, then the term in (3.1) coming from i and j differentiates as:
$$\frac{\partial(s_{ik}x_i x_k)}{\partial x_k} = s_{ik}x_i.$$
- If $i = j = k$, then the term in (3.1) coming from i and j differentiates as:
$$\frac{\partial(s_{kk}x_k^2)}{\partial x_k} = 2s_{kk}x_k.$$

Thus

$$\begin{aligned}
\frac{\partial(\boldsymbol{x}'S\boldsymbol{x})}{\partial x_k} &= \sum_{j\neq k} s_{kj}x_j + \sum_{i\neq k} s_{ik}x_i + 2s_{kk}x_k \\
&= \sum_{j\neq k} s_{kj}x_j + s_{kk}x_k + \sum_{i\neq k} s_{ik}x_i + s_{kk}x_k \\
&= \sum_{j=1}^{n} s_{kj}x_j + \sum_{i=1}^{n} s_{ik}x_i \\
&= \sum_{j=1}^{n} s_{kj}x_j + \sum_{i=1}^{n} s_{ki}x_i \\
&= 2\sum_{j=1}^{n} s_{kj}x_j \\
&= 2(S\boldsymbol{x})_k.
\end{aligned}$$

Combining these gives the result.

As a special case, if $S = \mathrm{I}$, then $\frac{\partial(\boldsymbol{x}'\boldsymbol{x})}{\partial \boldsymbol{x}} = 2\boldsymbol{x}$.

The same method shows that if S is not symmetric, then $\frac{\partial(\boldsymbol{x}'S\boldsymbol{x})}{\partial \boldsymbol{x}} = (S + S')\boldsymbol{x}$, but in fact we will only ever apply this in the case where S is symmetric.

Another useful example, rather easier than the one above is the following (you could treat this as an exercise). If $\boldsymbol{a}$ is a constant p-vector, and $f(\boldsymbol{x}) = \boldsymbol{a}'\boldsymbol{x}$, then $\nabla f = \frac{\partial f}{\partial \boldsymbol{x}} = \boldsymbol{a}$.

3.4 Constrained optimisation

For several of the techniques in the module, we will need to perform a *constrained optimisation*. That is, we will need to work out the maximum value of a function *under some extra conditions*.

We first deal with the case where the constraints are equalities, and recover the very classical Lagrange multiplier theory. Then we will move on to consider the case where the constraints are inequalities; we will find similar results, known as the *Karush-Kuhn-Tucker* (KKT) multipliers.

Let's first state the Lagrange multiplier theory.

Suppose $\boldsymbol{x} = (x_1, \dots, x_n)$. To maximise $f(\boldsymbol{x})$ (a scalar function of $\boldsymbol{x}$) subject to k scalar constraints $g_1(\boldsymbol{x}) = 0, \dots, g_k(\boldsymbol{x}) = 0$ with $k < n$, we use *Lagrange multipliers* $\lambda_1, \dots, \lambda_k$. Define $\Omega = f(\boldsymbol{x}) - \sum_{j=1}^{k} \lambda_j g_j(\boldsymbol{x})$, and maximise/minimise Ω with respect to the $n + k$ variables $x_1, \dots, x_n, \lambda_1, \dots, \lambda_k$.

Example 3.1 If we are asked to minimise $x_1^2 + x_2^2$, subject to $x_1 + x_2 = 1$, we write $\Omega = x_1^2 + x_2^2 - \lambda(x_1 + x_2 - 1)$. Then $\frac{\partial\Omega}{\partial x_1} = 2x_1 - \lambda$, $\frac{\partial\Omega}{\partial x_2} = 2x_2 - \lambda$, and $\frac{\partial\Omega}{\partial\lambda} = x_1 + x_2 - 1$. It is easy to see that the solution to all these equations is $x_1 = x_2 = \frac{1}{2}$.

Here's a more complicated statistical example: **Example 3.2** Suppose that $t_1, \dots, t_n$ are unbiased estimates of θ with variances $\sigma_1^2, \dots, \sigma_n^2$; let's try to find the best linear unbiased estimate of θ.

Take a linear combination $\tau = \sum \alpha_k t_k$. We want to choose the coefficients α_i so that τ has minimum variance subject to the constraint of being unbiased. Now $\mathbb{E}[t_i] = \theta$ for all i, so $\mathbb{E}[\tau] = \theta$, and so we have the constraint $\sum \alpha_k = 1$. Also, $\mathbb{V}[\tau] = \sum \alpha_k^2 \sigma_k^2$. Let

$$\Omega = \sum \alpha_k^2 \sigma_k^2 - \lambda\Big(\sum \alpha_k - 1\Big).$$

Then

$$\frac{\partial\Omega}{\partial\alpha_i} = 2\alpha_i\sigma_i^2 - \lambda, \qquad \frac{\partial\Omega}{\partial\lambda} = \sum \alpha_k - 1.$$

Setting these to 0, the first equations give $\alpha_i = \frac{1}{2}\frac{\lambda}{\sigma_i^2}$, and then the final one gives $\lambda = 2/\sum \frac{1}{\sigma_k^2}$. Substituting back gives

$$\alpha_i = \frac{1}{\sigma_i^2}\left(\sum \frac{1}{\sigma_k^2}\right)^{-1},$$

and the BLUE estimate of θ is

$$\widehat{\theta} = \left(\sum \frac{t_k}{\sigma_k^2}\right) \Big/ \left(\sum \frac{1}{\sigma_k^2}\right).$$

Let's indicate why Lagrange multipliers work.

First, consider what it means for a point (x_0, y_0) to be a maximum of a function $f(x, y)$. The idea is that any move away from the point, to $(x_0 + h, y_0 + k)$ say, will produce a *smaller* value of f. That is,

$$f(x_0 + h, y_0 + k) \leq f(x_0, y_0).$$

Assuming the differentiability of f (an essential part of the Lagrange multiplier theory), there is a

Taylor series for f:

$$f(x_0 + h, y_0 + k) = f(x_0, y_0) + hf_x(x_0, y_0) + kf_y(x_0, y_0) + \cdots .$$

Combining these two means that we need that the two partial derivatives $f_x = \frac{\partial f}{\partial x}$ and $f_y = \frac{\partial f}{\partial y}$ must both vanish at (x_0, y_0), otherwise we could choose h and k to make $hf_x(x_0, y_0) + kf_y(x_0, y_0)$ positive, which would increase the value of f. So we need

$$f_x(x_0, y_0) = 0,$$
$$f_y(x_0, y_0) = 0,$$

which is the criterion for (x_0, y_0) to be a stationary point for f. Note that if we write $\delta x = \begin{pmatrix} h \\ k \end{pmatrix}$ for the change from (x_0, y_0), then we can rewrite the Taylor series as

$$f(x_0 + h, y_0 + k) = f(x_0, y_0) + \delta x \cdot \nabla f(x_0, y_0) + \cdots ,$$

where the dot product is the usual one (but in 2 dimensions) and $\nabla f = \begin{pmatrix} f_x \\ f_y \end{pmatrix}$.

Let's work towards the theory of Lagrange multipliers by starting with a simple example in 2 variables, where we have one constraint.

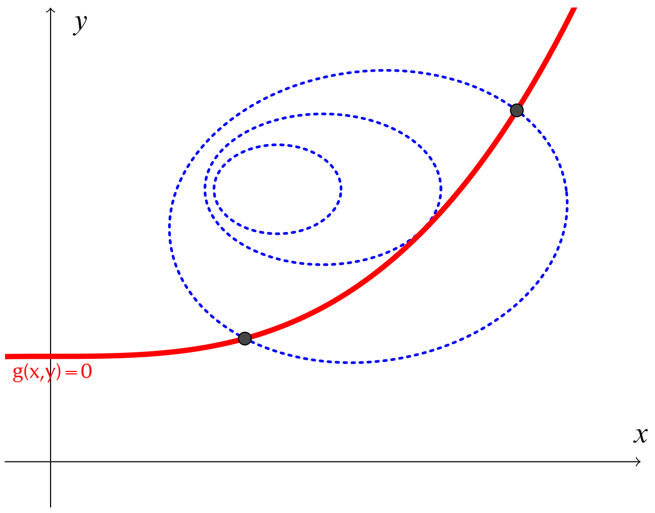
Suppose then that we want to maximise a function $f(x, y)$ of 2 variables subject to the constraint that $g(x, y) = 0$ for another function g. We need to assume both functions are differentiable for the theory to work. We follow the argument as above, except that we are no longer allowed to pick *any* h and k—we can only move along the curve $g(x, y) = 0$. So we need both that $g(x_0, y_0) = 0$ and that $g(x_0 + h, y_0 + k) = 0$. Using the Taylor series for g, we see that we must have

$$hg_x(x_0, y_0) + kg_y(x_0, y_0) = 0. \tag{3.2}$$

So we aren't allowed to pick any h and k—they must be related by equation (3.2) in order that we keep to the constraint. We can now reformulate the problem: For all h and k satisfying (3.2), we need $hf_x(x_0, y_0) + kf_y(x_0, y_0) = 0$ to find the maximum of f subject to $g = 0$.

In other words, whenever (h, k) is orthogonal to $(g_x(x_0, y_0), g_y(x_0, y_0))$, it must also be orthogonal to $(f_x(x_0, y_0), f_y(x_0, y_0))$. This happens when $(f_x(x_0, y_0), f_y(x_0, y_0))$ and $(g_x(x_0, y_0), g_y(x_0, y_0))$ are parallel, i.e., when $(f_x(x_0, y_0), f_y(x_0, y_0)) = \lambda(g_x(x_0, y_0), g_y(x_0, y_0))$.

We can see this in the following picture. Here, the blue dotted lines indicate level curves for f, and the red line is the set of (x, y) where $g(x, y) = 0$. If the constraint $g = 0$ is *not* tangent to a level curve of f (like at either of the two marked points), we can move along $g = 0$ in a direction which increases or decreases the value of f, and so we cannot be at a maximum or minimum. So we need the graph $g = 0$ to be tangent to the level curve of f:



This means that:

$$f_x(x_0, y_0) = \lambda g_x(x_0, y_0),$$
$$f_y(x_0, y_0) = \lambda g_y(x_0, y_0),$$

or alternatively

$$f_x(x_0, y_0) - \lambda g_x(x_0, y_0) = 0,$$
$$f_y(x_0, y_0) - \lambda g_y(x_0, y_0) = 0.$$

So we consider $\Omega = f - \lambda g$, and look for points (x_0, y_0) where $\frac{\partial \Omega}{\partial x} = 0$ and $\frac{\partial \Omega}{\partial y} = 0$.

Exactly the same argument works with more variables: we just recall the Taylor series for functions of several variables, and argue as we just did.

Let's now think about the situation where we have more than one constraint. We work in the 3-variable case here, and have 2 constraints, but you should persuade yourself that the argument will work for any number of variables, and any number of constraints. So we are trying to maximise a function $f(x, y, z)$ subject to $g_1(x, y, z) = 0$ and $g_2(x, y, z) = 0$. Again we want to think about small moves away from a maximum (x_0, y_0, z_0) to a nearby point $(x_0 + h, y_0 + k, z_0 + l)$. In order to keep satisfying the constraints, we need

$$hg_{1x}(x_0, y_0, z_0) + kg_{1y}(x_0, y_0, z_0) + lg_{1z}(x_0, y_0, z_0) = 0,$$
$$hg_{2x}(x_0, y_0, z_0) + kg_{2y}(x_0, y_0, z_0) + lg_{2z}(x_0, y_0, z_0) = 0,$$

so that (h, k, l) is orthogonal to both $(g_{1x}(x_0, y_0, z_0), g_{1y}(x_0, y_0, z_0), g_{1z}(x_0, y_0, z_0))$ and $(g_{2x}(x_0, y_0, z_0), g_{2y}(x_0, y_0, z_0), g_{2z}(x_0, y_0, z_0))$. As before, we want to find (x_0, y_0, z_0) such that whenever these hold, we have also that

$$hf_x(x_0, y_0, z_0) + kf_y(x_0, y_0, z_0) + lf_z(x_0, y_0, z_0) = 0,$$

and it is easy to see that this is equivalent to

$$f_x(x_0, y_0, z_0) = \lambda_1 g_{1x}(x_0, y_0, z_0) + \lambda_2 g_{2x}(x_0, y_0, z_0),$$
$$f_y(x_0, y_0, z_0) = \lambda_1 g_{1y}(x_0, y_0, z_0) + \lambda_2 g_{2y}(x_0, y_0, z_0),$$
$$f_z(x_0, y_0, z_0) = \lambda_1 g_{1z}(x_0, y_0, z_0) + \lambda_2 g_{2z}(x_0, y_0, z_0),$$

and this leads to consideration of $\Omega = f - \lambda_1 g_1 - \lambda_2 g_2$.

Now we'll turn to the case where we are maximising f subject to a constraint of the form $g \geq 0$.

So we suppose that we are at a point (x_0, y_0) which is at a maximum of f subject to the constraint $g(x_0, y_0) \geq 0$.

There are two cases: either $g(x_0, y_0) > 0$ or $g(x_0, y_0) = 0$.

In the first case, it is clear that we can move at least a small amount in any direction from (x_0, y_0) and the constraint will remain satisfied. So the only requirement is really that f has a maximum there, so that $f_x(x_0, y_0) = f_y(x_0, y_0) = 0$.

The second case is a little different. Here we can only move to $(x_0 + h, y_0 + k)$ if $g(x_0 + h, y_0, +k) \geq 0$. Again using the Taylor series, this requires that

$$hg_x(x_0, y_0) + kg_y(x_0, y_0) \geq 0. \tag{3.3}$$

So at a maximum as above, we ask that for all (h, k) such that (3.3) holds, we have $hf_x(x_0, y_0) + kf_y(x_0, y_0) \leq 0$.

Further, if we move in a direction along the boundary of the constraint, so that $hg_x(x_0, y_0) + kg_y(x_0, y_0) = 0$, we still have $hf_x(x_0, y_0) + kf_y(x_0, y_0) = 0$ for exactly the same reasons as when the constraint was an equality. So we still need that if $\Omega = f - \lambda g$, that (x_0, y_0) satisfies $\frac{\partial \Omega}{\partial x} = 0$ and $\frac{\partial \Omega}{\partial y} = 0$.

Let's also think about what the above discussion implies for $f - \lambda g$.

If (x_0, y_0) is a maximum of the constrained optimisation problem, then we might as well insist that $\lambda g(x_0, y_0) = 0$. After all, either (x_0, y_0) satisfies $g(x_0, y_0) > 0$, in which case there is no constraint on the direction we move in, and this gives no constraint on the original problem; we might as well be maximising f—or $g(x_0, y_0) = 0$.

Further, taking the partial derivatives of $\Omega = f - \lambda g$, and multiplying by h and k, we get that $h\Omega_x(x_0, y_0) + k\Omega_y(x_0, y_0) = 0$. On the other hand,

$$\begin{aligned} h\Omega_x(x_0, y_0) + k\Omega_y(x_0, y_0) &= h(f - \lambda g)_x(x_0, y_0) + k(f - \lambda g)_y(x_0, y_0) \\ &= (hf_x + kf_y)(x_0, y_0) - \lambda(hg_x + kg_y)(x_0, y_0). \end{aligned}$$

But we said that whenever $hg_x(x_0, y_0) + kg_y(x_0, y_0) \geq 0$, we have $hf_x(x_0, y_0) + kf_y(x_0, y_0) \leq 0$, and it follows that we must have $\lambda < 0$.

One can extend to more dimensions, and to combinations of several constraints, which might be mixtures of equalities and inequalities, and find the following:

> To maximise $f\colon \mathbb{R}^n \to \mathbb{R}$ subject to $g_1, \ldots, g_m \geq 0$ and $h_1, \ldots, h_n = 0$, form the objective function $\Omega = f - \sum_{i=1}^m \lambda_i g_i - \sum_{j=1}^n \mu_j h_j$.
>
> At a maximum $\boldsymbol{x}^*$, we require $\nabla\Omega(\boldsymbol{x}^*) = 0$, $g_i(x^*) \geq 0$ $(i = 1, \ldots, m)$, $h_j(x^*) = 0$ $(j = 1, \ldots, n)$, $\lambda_i < 0$ for $i = 1, \ldots, m$ and $\lambda_i g_i(x^*) = 0$ for $i = 1, \ldots, m$.

These are the *Karush-Kuhn-Tucker relations* for this problem. One can find similar relations for the minimisation problem for f or for constraints of the form $g_i \leq 0$ by replacing f or g_i with $-f$ or $-g_i$ where appropriate.

3.5 Gradient descent and Stochastic gradient descent

Very often in machine learning, we want to find the best choice of parameters for some model. "Best" indicates that we are choosing parameters so that some measure of total error is minimised. This error is a function of the parameters, and so we are trying to minimise some function of the parameters.

There are well-established methods in numerical analysis for this, as long as the error function is differentiable.

Take a function f of several variables $(x_1, \ldots, x_n)$. We aim to find a local minimum for f. We can see how the function varies in any particular direction by working out the partial derivatives $\frac{\partial f}{\partial x_i}$. It is standard to see that the direction of steepest gradient is in the direction of

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right).$$

Choose any point $\boldsymbol{x}_0 = (x_1, \ldots, x_n)$ to start from. As long as $\nabla f(\boldsymbol{x}_0) \neq \boldsymbol{0}$, we can decrease f by moving from $\boldsymbol{x}_0$ to $\boldsymbol{x}_0 - t\nabla f(\boldsymbol{x}_0)$ for small values of t. We can choose a t_0 which makes this negative, and define $\boldsymbol{x}_1 = \boldsymbol{x}_0 - t_0 f(\boldsymbol{x}_0)$, and we have found a point $\boldsymbol{x}_1$ such that $f(\boldsymbol{x}_1) < f(\boldsymbol{x}_0)$. We can repeat this to get a sequence of points $\boldsymbol{x}_2, \boldsymbol{x}_3, \ldots$, and there are conditions on the choice of the t_i which guarantee that this process will converge to a local minimum of f.

In the same way, we can use gradient *ascent* to find a local maximum.

So gradient descent is a very simple way to find a local minimum of a function. Convergence can be slow; for example, for a function of 1 variable, the Newton-Raphson method is a substantially faster method in general. However, iterative descent methods like this, and its variants, are common in machine learning.

In general, the error function in machine learning problems mentioned in the previous section is evaluated by summing over all the data points, and working out all of the individual errors. This is slow and time-consuming for large data sets!

Instead, we can take a small sample of data points—perhaps in random samples of 100 points, or even just a single point—and compute the gradient purely for these points. Then we can move in the direction which most quickly reduces the total error for the sample of points. This is *stochastic gradient descent*, since we make random choices for our samples. (Sometimes using the whole data set is called *batch* gradient descent.)

We will see that gradient descent methods are used in logistic regression, neural networks, support vector machines and iterative multidimensional scaling algorithms; in practice, stochastic variants are very frequently used.

3.6 The multivariate Gaussian distribution

This section concerns the multivariate Gaussian distribution. This is a multivariate distribution which generalises the well-known univariate Gaussian distribution.

There are multivariate generalisations of all the standard univariate distributions, as well as distributions that only really make sense in a multivariate context (imagine points distributed uniformly on the surface of a sphere, for example).

But the multivariate Gaussian distribution has a distinguished place amongst all distributions:

1. it is relatively easy to work with, and can be manipulated to give useful results;
2. it seems to be a reasonable model quite often in practice, even if there is no obvious reason why data should be Gaussian;
3. there is a multivariate central limit theorem, which means that the mean of a collection of i.i.d. variables looks more and more Gaussian, so it does have a distinguished position amongst all the distributions.

Although the Level 3 version of this module will not require knowledge of statistical tests based on the distribution, the Level 4 and MSc versions will. When we perform statistical tests, we need to make some assumptions on how the data is generated. Statistical tests compare the actual data with what ought to be produced if the null hypothesis on the distribution is satisfied.

But it is important that everyone has some idea about what the multivariate Gaussian distribution looks like. There are machine learning procedures which rely on some sort of model fitting, and we try to imagine that the data looks like it is produced by the multivariate Gaussian in some sense (or some mixture, if there are several groups, for example).

So let's move to the definition.

Definition 3.4 (Multivariate Gaussian distribution) Suppose that $\boldsymbol{\mu} \in \mathbb{R}^k$, and that $\Sigma \in \mathcal{M}_{k\times k}$ is positive definite symmetric. Then the *multivariate Gaussian distribution* $\mathrm{N}_k(\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma)$ is the distribution with probability density function (pdf)

$$f(\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma) = \frac{|\Sigma|^{-1/2}}{(2\pi)^{k/2}} \exp\left[-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})'\Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right].$$

Notice that when $p = 1$, we recover the usual Gaussian distribution $\mathrm{N}(x \mid \mu, \sigma^2)$, where $\Sigma = \sigma^2$; the quantity $(\boldsymbol{x}-\boldsymbol{\mu})'\Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu})$ simplifies to $\frac{(x-\boldsymbol{\mu})^2}{\sigma^2}$.

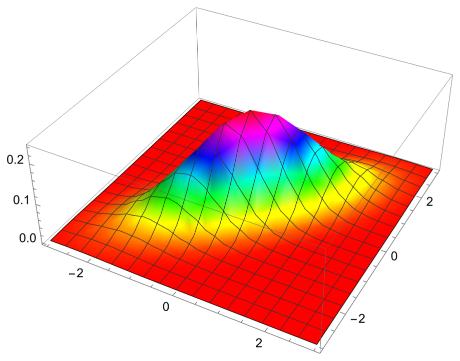
Let's first imagine that Σ is a diagonal matrix (i.e., the variables are uncorrelated). Then it is easy to see that for any positive constant c, the surface $(\boldsymbol{x}-\boldsymbol{\mu})'\Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu}) = c$ is an ellipsoid, centred on $\boldsymbol{\mu}$, and with axes in the direction of the co-ordinate axes.

More generally, if Σ is an arbitrary (positive definite symmetric) matrix, the multivariate Gaussian density is constant on surfaces where $(\boldsymbol{x}-\boldsymbol{\mu})'\Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu})$ is constant, and these are again ellipsoids centred at $\boldsymbol{\mu}$. The axes of each ellipsoid of constant density are in the direction of the eigenvectors of Σ^{-1} (recall that these are the same as the eigenvectors of Σ, but if $\Sigma\boldsymbol{x} = \lambda\boldsymbol{x}$, then $\Sigma^{-1}\boldsymbol{x} = \lambda^{-1}\boldsymbol{x}$), and their lengths are proportional to the reciprocals of the square roots of the eigenvalues of Σ^{-1}.

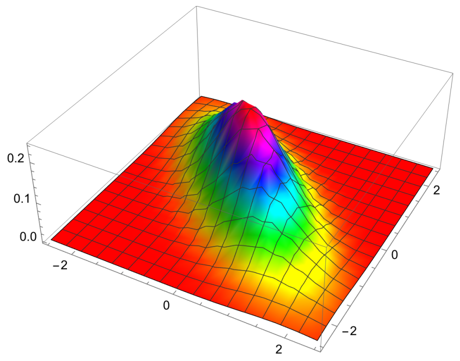
The main thing to take away here is that:

Contour plots of data coming from the multivariate Gaussian distribution are ellipsoidal

Figure 3.1 illustrate this fact for $k = 2$, and when there is (large) positive and negative correlation between the two variables.



(a) *Positive correlation*



(b) *Negative correlation*

Figure 3.1. *PDF of a Gaussian distribution centred at the origin with positive correlation on the left and negative correlation on the right. Note the level curves are elliptical with the main axis in the direction of the correlation.*

In the special case, when $\Sigma = \mathrm{I}$, the level curves are indeed circular as illustrated in Figure 3.2, indicating independence between both variables.

The quantity $(\boldsymbol{x}-\boldsymbol{\mu})'\Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu})$ gives a notion of *squared statistical distance* of $\boldsymbol{x}$ from $\boldsymbol{\mu}$. If one component has a much larger variance than others, it will contribute disproportionately to the Euclidean distance, and we would prefer to scale the variables so that this is avoided, and contribute less. Similarly, two highly correlated random variables will contribute too much to the Euclidean

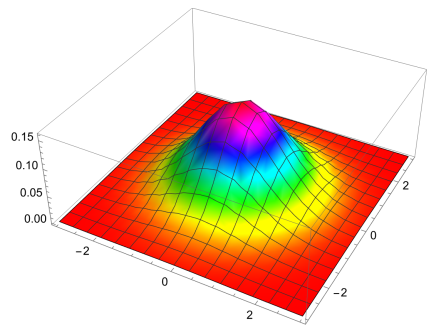


Figure 3.2. *The pdf of a bivariate standard Gaussian distribution. Level curves are circles centred at the origin.*

distance—if two observations differ significantly in one of the two variables, then it is likely that they will differ more in the other variable, and so the Euclidean distance between them will be larger. With this "statistical distance", both these problems are avoided. Often this, and related quantities, are named after *Mahalanobis*, the statistician who remarked on its importance.

We can use this Mahalanobis distance to detect outliers; as well as plotting the points, if possible, to detect unusual observations visually, we can calculate the squared statistical distances from the mean for each point, and consider whether any are unusually far away.

Lemma 3.3 If $\boldsymbol{x} \sim \mathrm{N}_k(\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma)$ then $\mathbb{E}[\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma] = \boldsymbol{\mu}$ and is $\mathbb{V}[\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma] = \Sigma$.

Proof Assume $\boldsymbol{x} \sim \mathrm{N}_k(\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma)$ and let $\boldsymbol{y} = \Sigma^{-1/2}(\boldsymbol{x} - \boldsymbol{\mu})$, where $\Sigma^{-1/2}$ is as defined earlier, then $(\boldsymbol{x} - \boldsymbol{\mu})'\Sigma^{-1}(\boldsymbol{x} - \boldsymbol{\mu}) = \boldsymbol{y}'\boldsymbol{y} = \sum_{i=1}^{p} y_i^2$. Now the density of y is

$$f(\boldsymbol{y} \mid \boldsymbol{\mu}, \Sigma) = \frac{|\Sigma|^{-1/2}}{(2\pi)^{p/2}} \exp\left[-\frac{1}{2}\boldsymbol{y}'\boldsymbol{y}\right] J_{xy},$$

where J_{xy} is the Jacobian of the transformation given by $\left|\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}\boldsymbol{y}}\right|$, where $\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}\boldsymbol{y}} \in \mathcal{M}_{k\times k}$ with ij-th element $\frac{\partial x_i}{\partial y_j}$.

Now $y = \Sigma^{-1/2}(\boldsymbol{x} - \boldsymbol{\mu})$, so $\boldsymbol{x} = \Sigma^{1/2}\boldsymbol{y} + \boldsymbol{\mu}$, so $\frac{dx}{dy} = \Sigma^{1/2}$. Thus $J_{xy} = |\Sigma|^{1/2}$, giving

$$f(\boldsymbol{y} \mid \boldsymbol{\mu}, \Sigma) = \frac{1}{(2\pi)^{k/2}} \exp\left[-\frac{1}{2}\boldsymbol{y}'\boldsymbol{y}\right].$$

Thus the $y_i \sim \mathrm{N}(y_i \mid 0, 1)$, independent.

Now if $\boldsymbol{x} \sim \mathrm{N}_k(\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma)$, and $\boldsymbol{y} = \Sigma^{-1/2}(\boldsymbol{x} - \boldsymbol{\mu})$, then

$$\mathbb{E}[\boldsymbol{y} \mid \boldsymbol{\mu}, \Sigma] = \Sigma^{-1/2}(\mathbb{E}[x] - \boldsymbol{\mu}), \qquad \mathbb{V}[y] = \Sigma^{-1/2}\, \mathbb{V}[x]\Sigma^{-1/2}.$$

As $E[y] = \mathbf{0}$ and $\mathbb{V}[y] = \mathrm{I}$, we see that $\mathbb{E}[\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma] = \boldsymbol{\mu}$ and $\mathbb{V}[\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma] = \Sigma$.

Although the definition of the multivariate Gaussian distribution looks complicated, it is only the exponential term which is important—the other term is just there to normalise things so that the function has volume 1. It is an exercise for you to check that this, so that it really is a probability distribution; this is not too hard, once you recall that $\boldsymbol{y} = \Sigma^{-1/2}(\boldsymbol{x} - \boldsymbol{\mu})$ is distributed as $\mathrm{N}_k(\boldsymbol{y} \mid \boldsymbol{0}, \mathrm{I})$; then the integral breaks up into a product of p integrals which arise in the analogous univariate calculation, and the Jacobian of the transformation is $|\Sigma|^{-1/2}$. (More details are got by understanding the proof of the lemma.)

We list some properties of the multivariate normal distribution, without proof.

1. Suppose that $\boldsymbol{x} \sim \mathrm{N}_k(\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma)$, and that $\boldsymbol{w} \in \mathbb{R}^k$. Then the linear combination $y = \boldsymbol{w}'\boldsymbol{x} \sim \mathrm{N}(y \mid \boldsymbol{w}'\boldsymbol{\mu}, \boldsymbol{w}'\Sigma\boldsymbol{w})$.

   In particular, if we let $\boldsymbol{w}_j \in \mathbb{R}^k$ be a vector with 1 in the j-th position and zero elsewhere, we see that $y_j = \boldsymbol{w}_j'\boldsymbol{y} \sim \mathrm{N}\big(y_j \bigm| \mu_j, \Sigma_{jj}\big)$, i.e. the marginal distributions of the individual variables of a multivariate Gaussian distribution are univariate Gaussian distributions.

2. More generally, suppose that A is a $q \times k$-matrix. Then $\boldsymbol{y} = A\boldsymbol{x} \sim \mathrm{N}_q(\boldsymbol{y} \mid A\boldsymbol{\mu}, A\Sigma A')$.

   Again, as a special case, if $\boldsymbol{x} \sim \mathrm{N}_k(\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma)$, and $\boldsymbol{y}$ denotes a subset of variables of size q, then $\boldsymbol{y} \sim \mathrm{N}_q\big(\boldsymbol{y} \bigm| \boldsymbol{\mu}_{\boldsymbol{y}}, \Sigma_{\boldsymbol{y}}\big)$, where $\boldsymbol{\mu}_{\boldsymbol{y}} \in \mathbb{R}^q$ denotes the vector formed by taking the entries of $\boldsymbol{\mu}$ corresponding to $\boldsymbol{y}$, and $\Sigma_{\boldsymbol{y}} \in \mathcal{M}_{q\times q}$ is the matrix formed from Σ in the same way.

3. Assume $\boldsymbol{z} \in \mathbb{R}^{k+q}$ can be partitioned as $\boldsymbol{z} = \{\boldsymbol{x}, \boldsymbol{y}\}$, with $\boldsymbol{x} \in \mathbb{R}^k$ and $\boldsymbol{y} \in \mathbb{R}^q$, such that
$$\begin{pmatrix}\boldsymbol{x}\\ \boldsymbol{y}\end{pmatrix} \sim \mathrm{N}_{k+q}\left(\begin{pmatrix}\boldsymbol{x}\\ \boldsymbol{y}\end{pmatrix} \middle| \begin{pmatrix}\boldsymbol{\mu}_x\\ \boldsymbol{\mu}_y\end{pmatrix}, \begin{pmatrix}\Sigma_X & 0\\ 0 & \Sigma_Y\end{pmatrix}\right)$$
   then $\boldsymbol{x}$ and $\boldsymbol{y}$ are independent.

4. More generally, if
$$\begin{pmatrix}\boldsymbol{x}\\ \boldsymbol{y}\end{pmatrix} \sim \mathrm{N}_{k+q}\left(\begin{pmatrix}\boldsymbol{x}\\ \boldsymbol{y}\end{pmatrix} \middle| \begin{pmatrix}\boldsymbol{\mu}_x\\ \boldsymbol{\mu}_y\end{pmatrix}, \begin{pmatrix}\Sigma_X & \Omega\\ \Omega & \Sigma_Y\end{pmatrix}\right)$$
   so that Ω is the correlation between $\boldsymbol{x}$ and $\boldsymbol{y}$, then the (conditional) distribution of $\boldsymbol{x} \mid \boldsymbol{y}$ is multivariate Gaussian, with
$$\mathbb{E}[\boldsymbol{x} \mid \boldsymbol{y}] = \boldsymbol{\mu}_x + \Omega\Sigma_y^{-1}(\boldsymbol{y} - \boldsymbol{\mu}_y)$$
   and
$$\mathbb{V}[\boldsymbol{x} \mid \boldsymbol{y}] = \Sigma_x - \Omega\Sigma_y\Omega^{-1}$$
   (See, for example, Chatfield-Collins, p.98.)

5. If $\boldsymbol{x} \sim \mathrm{N}_k(\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma)$, then $s = (\boldsymbol{x} - \boldsymbol{\mu})'\Sigma^{-1}(\boldsymbol{x} - \boldsymbol{\mu})$ is distributed as $\chi^2_{(k)}(s)$, the chi-squared distribution with k degrees of freedom. This follows from the proof of Lemma 3.3, but we sketch a proof next: If $\boldsymbol{x} \sim \mathrm{N}_k(\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma)$, then $\boldsymbol{y} = \Sigma^{-1/2}(\boldsymbol{x} - \boldsymbol{\mu})$ follows a standard k-variate Gaussian, so
$$s = (\boldsymbol{x} - \boldsymbol{\mu})'\Sigma^{-1}(\boldsymbol{x} - \boldsymbol{\mu}) = \boldsymbol{y}'\boldsymbol{y} = \sum_{i=1}^{j} y_i^2 \sim \chi^2_{(k)}(s),$$

where each y_i^2 is the square of a standard (univariate) Gaussian, and the result follows.

Let's state some further results, whose proofs are in the extra statistical handout.

Proposition 3.4 Suppose that $\boldsymbol{x} \sim \mathrm{N}_k(\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma)$, and that $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ are a random sample. Define $\bar{\boldsymbol{x}} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i$. Then

$$\bar{\boldsymbol{x}} \sim \mathrm{N}_k\big(\bar{\boldsymbol{x}} \mid \boldsymbol{\mu}, \tfrac{1}{n}\Sigma\big).$$

In fact, this is a special case of the more general multidimensional central limit theorem:

Theorem 3.1 (Multivariate Central Limit Theorem) Let $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ denote independent observations from any distribution with mean $\boldsymbol{\mu}$ and finite covariance Σ. Then $\sqrt{n}(\bar{\boldsymbol{x}} - \boldsymbol{\mu})$ has approximately an $\mathrm{N}_k(\bar{\boldsymbol{x}} \mid \mathbf{0}, \Sigma)$ distribution when n is large, and large relative to p.

Mostly in this course, we will consider data-analytic techniques, i.e., methods that depend only on the data received, and which make no assumptions on an underlying distribution of the data generated.

This exploratory data analysis is crucial in data science—it allows the researcher to get a feel for the data set, and might simplify later more formal statistical analyses.

But there are many techniques available, and there are some "model-based" techniques that assume that data comes from multivariate Gaussian distributions. For example, we could imagine that there are two groups of patients with different medical conditions, and that patients with one condition might expect to have their readings distributed by means of one multivariate Gaussian distribution, while the other group might expect their readings to be generated by a multivariate Gaussian distribution with a different mean and variance. Such population distributions are called *Normal-mixture models*, and there are a number of methods to consider the problems below which work under this sort of assumption.

4 EDA AND DATA VISUALISATION

The aim of data science is to look at a data set, and to try to understand its important features. One should start with an informal examination of the data set, and try to spot interesting patterns and features; after that, there may be scope for testing whether the features you spot are statistically significant. We won't be doing much of that in the Level 3 version of the module, but will expect a little more from Level 4 and the MSc versions, who have an additional chapter on this material.

Data scientists report that most of their time is spent cleaning data, to get it into a useful form. We'll say virtually nothing about this, beyond a few remarks on the lab sheets. We assume throughout that we have a nicely formed data set, in CSV format; such data is sometimes called "tidy"—each observation forms a row of our table, and each variable is measured in a single column.

Probably the first thing to do to make sense of a new data set is to compute simple summaries of each column.

We import our data as a data frame `df` into either `R` or into `Python` (with the `numpy` and `pandas` packages). The summary table ought to have something like:

1. the minimum value of each variable;
2. the first quartile;
3. the median;
4. the mean;
5. the third quartile;
6. the maximum value.

There is a lab sheet with details on how to form these summaries, in either `Python` or `R`.

Computing numerical summaries of data is obviously valuable, but often visualising the data through pictures is a more effective way to understand the information and discover patterns. The human brain seems to be designed to find patterns in data, but sometimes it seems to do so even when the patterns are just the result of chance. This is why a proper statistical analysis can sometimes be desirable, although it often requires you to be confident that the data should be normally distributed, for example, and this can be difficult to confirm.

We can classify our data sets by the number of dimensions involved; that is, the number of measurements taken in each observation.

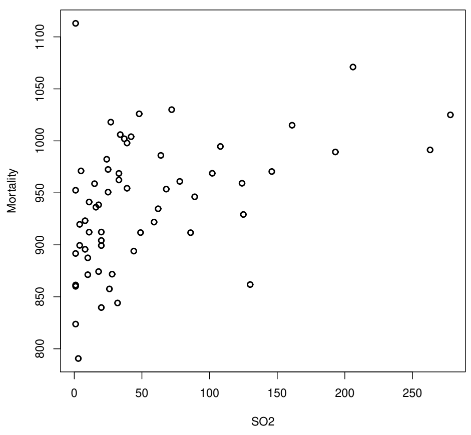
When the number of dimensions is small, there are some well-known techniques for data visualisation. Quite a lot of the techniques for data visualisation have been put on a lab sheet, and aren't addressed explicitly here.

As the great American statistician and mathematician John Tukey wrote, in his book *Exploratory Data Analysis* (1977), “There is no excuse for failing to plot and look.”

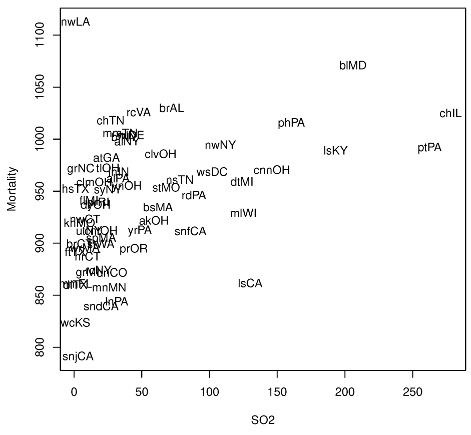
4.1 Scatterplots etc.

Of course, given a multivariate data set, we can plot each component individually using the usual univariate methods: histograms, stem-and-leaf plots and box plots. Certainly the main way to consider data where there is more than one variable is by making comparisons of the variables against each other, so that we examine the relationships between the variables. Since our brain works with 2-dimensional inputs, comparing one variable against another is the most common method, and scatter plots are the principal technique.

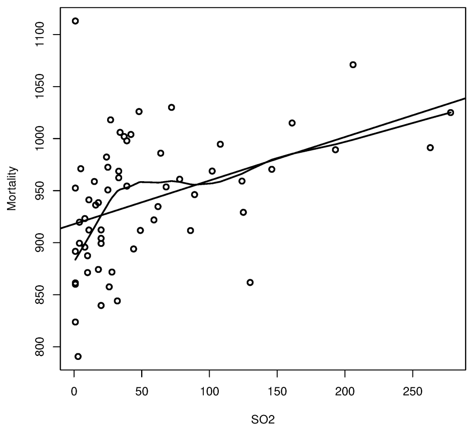
The third lab sheet deals with scatter plots in `Python` and `R`. You might like to repeat the analysis on that lab sheet with the `chap2airpoll.csv` data set used in the second lab sheet. Here is a basic scatterplot of two of the variables in the `airpoll` data set:



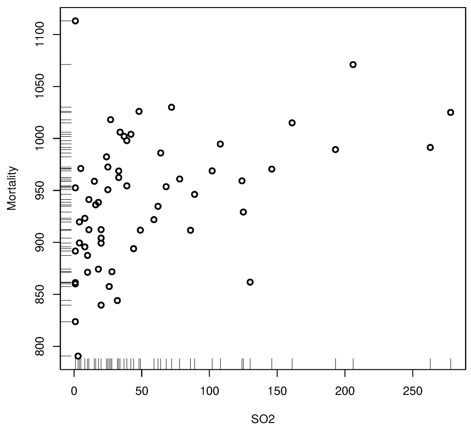
We can enhance this in various ways, e.g., by adding labels to the points:



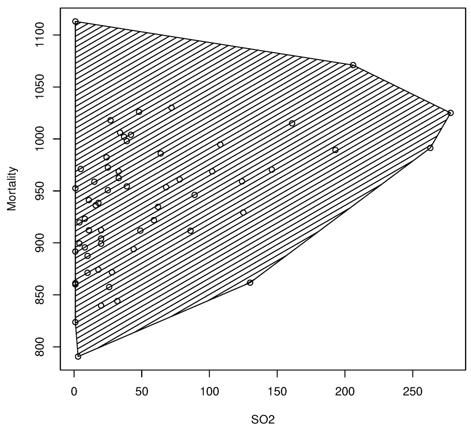
Or we can add regression lines, or other estimators:



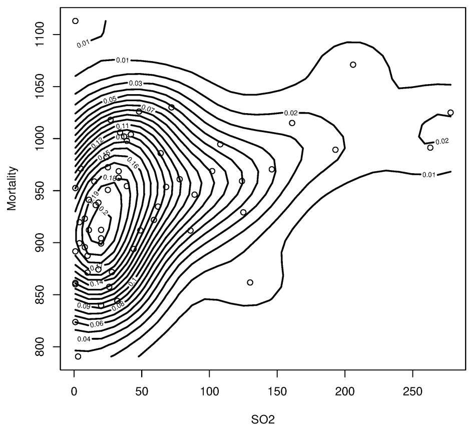
It is often nice to add marginal distributions ("rug" plots):



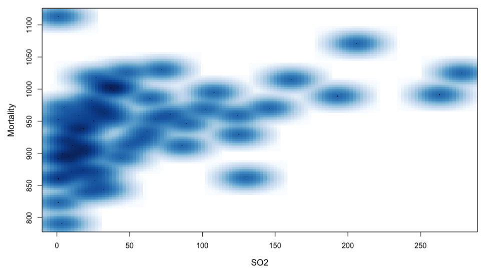
The data as a whole can be viewed using a convex hull:



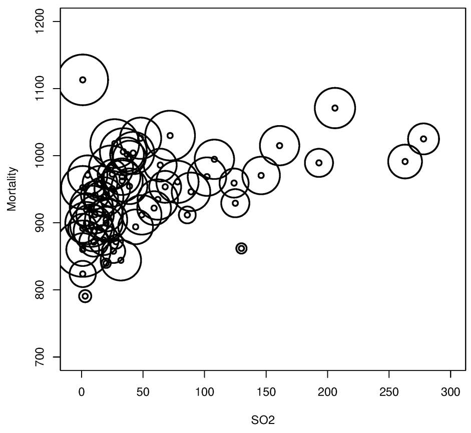
Especially with more points, it is a common technique to imagine a Gaussian placed at each data point, and added; this gives a 3-dimensional view of the data which reflects the density of the data. We can use this for a contour map:



or for a "heat" map, where the colour at a point reflects the height of the contour map:



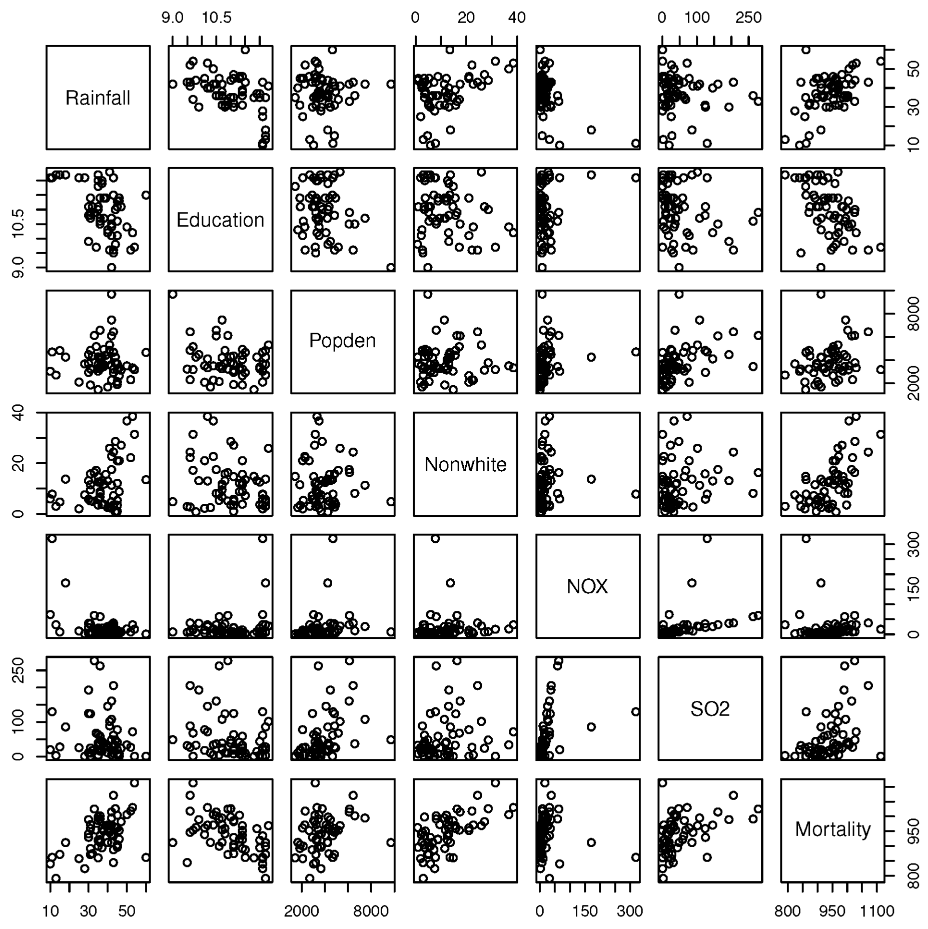
A third variable can be included—here, the radius of the circles are proportional to the `Rainfall` variable:



Data visualisation is a huge topic. Although scatter plots are excellent when there are a fairly small number of points, these other techniques are often better with more points. The lab sheet also gives some techniques for contour maps, heat maps etc. Other options include:

- bivariate histograms, drawn in perspective
- bivariate boxplots
- 2-dimensional frequency plots
- augmented scatter plots

With a few more dimensions, the key tool in displaying multivariate data is scatterplots of pairwise components, arranged as a *matrix plot*:



All these techniques are available for either `Python` or `R`. Some are addressed by the lab sheet, but for others, you will need to look at the "cheat sheets" for the appropriate packages you are using for your data visualisations (probably `seaborn` for `Python`, or base `R` graphics or `ggplot2` in `R`).

4.2 Other visualisation techniques

There are many other techniques for visualising data. We won't use any of these in this module, but you might like to be aware of some. Weather maps are good examples of data visualisation—they cover information on temperature, rainfall, atmospheric pressure, wind speed, etc., all on a single map.

With not too many observations, but a lot of variables, we can encode each observation as a function, using *Andrews plots* (see D.F.Andrews (1972) *Plots of high-dimensional data*, Biometrics **28**, 125–136).

As usual, we take the data to be vector observations $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$, so x_{ij} is the jth element of the ith observation. Then we define

$$f_{x_i}(t) = \frac{1}{\sqrt{2}} x_{i1} + x_{i2} \sin t + x_{i3} \cos t + x_{i4} \sin 2t + \cdots + x_{ip} \overset{\sin}{\underset{\cos}{}} \Big(\big(\lfloor \tfrac{p}{2} \big) \rfloor t \Big).$$

This maps p-dimensional data $\boldsymbol{x}_i$ onto 1-dimensional $\{f_{x_i}(t)\}$ for any t. If we plot $f_{x_i}(t)$ over $-\pi < t \leq +\pi$, we obtain a 1-dimensional representation of the data, with each data point encoded as a function; we can sometimes use this to distinguish groups, for example. The paper of Andrews gives more details of the statistical properties, but here are a few:

1. preserves means, i.e.,
$$f_{\bar{x}}(t) = \frac{1}{n} \sum_{i=1}^{n} f_{x_i}(t);$$
2. preserves distances, i.e., if we define the square of the distance between two functions as
$$|f_{x_1}(t) - f_{x_2}(t)|^2 = \int_{-\pi}^{\pi} (f_{x_1}(t) - f_{x_2}(t))^2 \, \mathrm{d}t = \pi \sum_{j=1}^{p} \big(x_{1j} - x_{2j}\big)^2,$$
we get $\pi \times$ the square of the Euclidean distance between x_1 and x_2, so close points appear as close functions (though not necessarily vice versa);
3. yields 1-dimensional views of the data: at $t = t_0$ we obtain the projection of the data onto the vector
$$f_1(t_0) = \Big(1/\sqrt{2},\, \sin t_0,\, \cos t_0,\, \sin 2t_0,\, \ldots\Big)'$$
(i.e., viewed from some position, the data look like the 1-dimensional scatter plot given on the vector $f_1(t_0)$, i.e., the line intersecting the Andrews plot at $t = t_0$).
4. significance tests and distributional results are available.

- the plot does depend on the order in which the components are taken. A preliminary transformation of data may be sensible (we'll see the idea of principal components in the next chapter);
- other sets of orthonormal functions are possible;
- the "coverage" decreases rapidly as the dimensionality increases, i.e., for high dimensions, many "views" of the data are missed, and we miss separation between groups of points or other important features.

For a fairly small number of variables (no more than 20, say), and not too many observations, *star plots* and *Chernoff faces* are alternatives; for star plots, we represent each observation by a polygon where the length of the vector to each vertex corresponds to the value of a particular observation. Similarities between observations are reflected in similarities of the corresponding pictures. Chernoff faces are similar; each variable is encoded as some feature of a face (e.g., length of nose, curvature of mouth etc.)—the idea is that humans' ability to recognise faces has evolved to a very high extent, and similarities between faces is easy to spot. Both are programmed in certain `R` and `Python` libraries.

4.3 Closing remarks

Simple scatter plots arranged in a matrix work well when there are not too many variables. If there are more than about 5 variables, it is difficult to focus on particular displays and so understand the structure. If there are not too many observations, there are ways to assign to each some symbol (e.g., a star or polygon) to indicate the values of the variables for that observation, and enable comparisons between the observations.

However, many multivariate statistical analyses involve very large numbers of variables; more than 50 is routine, and new areas of application often involve more than 1000. It would be great if we were able to use some geometrical intuition from 2- and 3-dimensional views, but it turns out that working in many dimensions has all sorts of issues that mean that you should be wary of generalising your intuition! For example, in 2 dimensions, we can pack circles together so that 90% of space is covered by them—once we get to 10 dimensions, the corresponding figure is less than 10%, and falls to below 0.4% by 20 dimensions.

What would be useful is a display of "all of the data" using just a few scatterplots, i.e., using just a few variables, which somehow shows most of the structure in the data. That is, we need to select "the most interesting variables". This may mean concentrating on the "most interesting" actual measurements, or it may mean combining the variables together in some way to create a few new ones which are the "most interesting". That is, we need some technique of *dimensionality reduction*.

5 PCA: Reducing dimensionality

Suppose we have a large amount of data, involving many attributes. For example, we could be surveying 10000 people, and collecting data on age, height, weight, income, blood pressure, etc., so that we associate many pieces of information with one person. Then we have many many data points which we plot in a high-dimensional space, which we want to analyse, and find patterns in.

As we have explained in the previous chapter, it is not easy to visualise data in the multivariate setting: when the number of dimensions is large, we cannot easily use graphical techniques. It is often desirable to try to view the data in smaller dimensions.

As an analogy, consider the planets of the solar system. While they naturally live in 3-dimensional space (ignoring relativistic distortions etc.!), they approximately orbit in a 2-dimensional plane. It is simpler to try to view the planets as orbiting in that 2-dimensional space.

Dimension reduction is our first example of unsupervised learning. We have a data set, and want to examine it as a whole, and try to do something useful with it—in this case, to try to view the data by looking in a smaller number of dimensions.

This leads to the problem:

> Given n points in p dimensional space $\mathbb{R}^p$, and some number $q < p$, find the "best" q-dimensional space $\mathcal{X} \subseteq \mathbb{R}^q$, and projections of the n points into $\mathcal{X}$ so that as little information is lost as possible.

If we can find this set $\mathcal{X}$, we can then regard the data points as lying (nearly) in the smaller dimensional set, and can then hope that this reduction in dimensions allows us to visualise the data better.

Using fewer dimensions for the data has other benefits too: the data can be manipulated more easily, and the data can be stored in less space (*data compression*).

Often the data can be correlated (e.g., height and weight may be related), and this means that the data points might be appropriately represented by some combination of variables (e.g., size). We therefore try to replace the variables with a smaller set (the 2 dimensions of height and weight might be replaced with a 1-dimensional measure of size), keeping as much information as we can.

It may not be appropriate to find a linear subspace, but some more general shape. This leads to a whole new class of dimensionality-reduction problems. To start with, though, we will concentrate on the reduction to a linear space, as we can use the full power of matrix techniques. It is this case which we shall focus on first.

Modern techniques tend to involve "manifold-reduction", where the data is fit onto a more general shape. We will also mention "kernel methods" in a different context, which allow some nonlinear shapes—kernel methods can be applied to many techniques in machine learning, and Principal Components Analysis (PCA) is among these.

5.1 Introduction

As usual, the data matrix is given by $X = \{x_{ij} : i = 1, \dots, n, \ j = 1 \dots, p\} = \{\boldsymbol{x}_i : i = 1, \dots, n\}$.

The objective is to find a *linear* transformation given by a "change of basis", so that new variables are linear combinations of old variables with nice properties. In particular, such linear transformations correspond to multiplying the data matrix X to get another matrix Y; we want the map $X \longrightarrow Y$ such that the first component of Y is the "most interesting", the second is the "second most interesting", etc., i.e., we want to choose a new coordinate system so that the data, when referred to this new system Y, are such that the first component contains "most information", the second contains the next "most information", etc.

Note that "most interesting" and "most information" translate to "maximum variance"; after all, if all the data looks similar in one component, that is probably not that useful.

With luck, the first few components contain "nearly all" the information in the data, and the remaining components contain relatively little information, and can be discarded (i.e., the statistical analysis can be concentrated on just the first few components – multivariate analysis is much easier in 2 or 3 dimensions, where the data can be visualised easily).

A linear transformation $X' \longrightarrow Y'$ is given by $Y' = X'A$ where A is a $p \times p$-matrix; it makes statistical sense to restrict attention to *non-singular* matrices A. If A happens to be an *orthogonal* matrix, i.e., $A'A = \mathrm{I}_p$, then the transformation $X' \longrightarrow Y'$ is an orthogonal transformation (i.e., just a rotation and/or a reflection of the n points in p-dimensional space).

Before going further into this, let's think about what this means geometrically. If X' is a data matrix, and a is a vector of length 1, then $Y' = X'a$ turns out to be the *projection* of X' onto a: values of Y' give the co-ordinates of each observation *along the vector* a.

Let's give an example of this: **Example 5.1** Suppose that $\boldsymbol{x}_1 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$, $\boldsymbol{x}_2 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$, $\boldsymbol{x}_3 = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$ and $\boldsymbol{a} = \begin{pmatrix} \frac{3}{5} \\ -\frac{4}{5} \end{pmatrix}$.

So

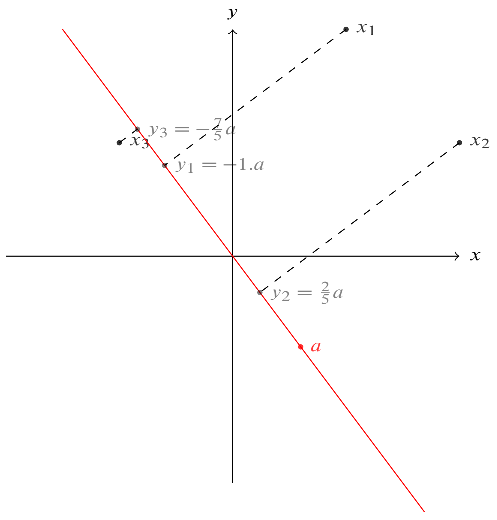
$$X' = \begin{pmatrix} 1 & 2 \\ 2 & 1 \\ -1 & 1 \end{pmatrix}.$$

Then

$$Y' = X'\boldsymbol{a} = \begin{pmatrix} -1 \\ \frac{2}{5} \\ -\frac{7}{5} \end{pmatrix}$$

These give the amounts along $\boldsymbol{a}$ of the projections of each variable onto the line given by $\boldsymbol{a}$. For example, $x_1 = -1\boldsymbol{a} + z_1$, where z_1 is orthogonal to $\boldsymbol{a}$. And $x_2 = \frac{2}{5}\boldsymbol{a} + z_2$, where z_2 is orthogonal to $\boldsymbol{a}$.

Let's draw everything geometrically on axes:



So if we project an observation $\boldsymbol{x} = (x_1, \ldots, x_p)$ onto a vector $\boldsymbol{a} = (a_1, \ldots, a_p)$ to get $\boldsymbol{y} = \boldsymbol{x}'\boldsymbol{a}$, we just get the linear combination

$$\boldsymbol{y} = \boldsymbol{x}'\boldsymbol{a} = a_1x_1 + \cdots + a_px_p.$$

Think of this as a weighted average of the variables. Applying this to all observations gives a *vector* $\boldsymbol{y}$, whose i-th component, is the linear combination a_1 multiplied by the value of the first variable for the i-th observation, plus $\boldsymbol{a}_2$ multiplied by the value of the second variable and so on.

5.2 Principal components

The basic idea is to find a set of orthogonal coordinates such that the sample variances of the data with respect to these coordinates are in decreasing order of magnitude, i.e., the projection of the points onto the first principal component has maximal variance among all such linear projections, the projection onto the second has maximal variance subject to being uncorrelated with the first, the third has maximal variance subject to being uncorrelated with the first two, etc. So the first principal component gives the direction in which the data is most spread out, for example.

We want to take the linear combination $X'\boldsymbol{a}$ of the data matrix by choosing $\boldsymbol{a}$ so that the variance of $X'\boldsymbol{a}$ is maximised. A moment's thought will show you that this is not quite enough—we could replace $\boldsymbol{a}$ by $k\boldsymbol{a}$, and the variance increases by a factor of k^2. We need to normalise the choice of $\boldsymbol{a}$ in some way.

Definition 5.1 The *first principal component* is the vector $\boldsymbol{a}_1$ such that the projection of the data X' onto $\boldsymbol{a}_1$, i.e., $X'\boldsymbol{a}_1$, has maximal variance, subject to the normalising constraint that $\boldsymbol{a}_1'\boldsymbol{a}_1 = 1$ (i.e., $\boldsymbol{a}_1$ has length 1).

Let us now explain how to find $\boldsymbol{a}_1$.

We know that $\mathbb{V}[X'\boldsymbol{a}_1] = \boldsymbol{a}_1' \, \mathbb{V}[X']\boldsymbol{a}_1 = \boldsymbol{a}_1' S\boldsymbol{a}_1$. Notice that $\boldsymbol{a}_1'$ is a $1 \times p$ matrix, S is a $p \times p$ matrix and $\boldsymbol{a}_1$ is a $p \times 1$ matrix, so $\boldsymbol{a}_1' S\boldsymbol{a}_1$ is a scalar. So the problem is to maximise the scalar $\boldsymbol{a}_1' S\boldsymbol{a}_1$ subject to the constraint that $\boldsymbol{a}_1'\boldsymbol{a}_1 = 1$.

This is the first of several points in the module that we will need to perform some maximisation procedure for functions of several variables, constrained to satisfy some conditions. The necessary technique of Lagrange multipliers and differentiation with respect to a vector are given in Chapter 2. (Since there's no exam in the module, all proofs are given for information only, but might be of interest in their own right. And it's nominally a maths module, so we ought to justify some of this sort of thing!) The general procedure is the following:

1. Introduce a Lagrange multiplier and define a new objective function;
2. Differentiate with respect to $\boldsymbol{x}$ (generally a vector) and set to 0;
3. Recognise that this is an eigenequation with the Lagrange multiplier as eigenvalue;
4. Deduce that there are *only* a limited number of possible values for this eigenvalue (all of which can be calculated numerically);
5. Use some extra step to determine which eigenvalue gives the maximum (typically use the constraint somewhere).

Remark 5.1 We need to differentiate with respect to a vector below. If $\boldsymbol{x} = \big(x_1, \ldots, x_p\big)$ is a p-vector, and $f = f(\boldsymbol{x})$ is a scalar function of $\boldsymbol{x}$, recall that $\frac{\partial f}{\partial \boldsymbol{x}}$ is the vector $(\frac{\partial f}{\partial x_1} \; \cdots \; \frac{\partial f}{\partial x_p})'$. If $f(\boldsymbol{x}) = \boldsymbol{x}'S\boldsymbol{x}$, where S is a symmetric $p \times p$ matrix, then $\frac{\partial f}{\partial \boldsymbol{x}} = 2S\boldsymbol{x}$, by Lemma 3.2.

For this problem, this procedure works as follows:

- We have already remarked that we assume that $\boldsymbol{a}_1'\boldsymbol{a}_1 = 1$.
- Define $\Omega_1 = \boldsymbol{a}_1' S\boldsymbol{a}_1 - \lambda_1(\boldsymbol{a}_1'\boldsymbol{a}_1 - 1)$ where λ_1 is a Lagrange multiplier; we now need to maximise Ω_1 with respect to both $\boldsymbol{a}_1$ and λ_1.
- Setting $\frac{\partial \Omega_1}{\partial \lambda_1} = 0$ gives $\boldsymbol{a}_1'\boldsymbol{a}_1 = 1$.

  Differentiating with respect to (the vector) a_1 gives $\frac{\partial \Omega_1}{\partial \boldsymbol{a}_1} = 2S\boldsymbol{a}_1 - 2\lambda_1\boldsymbol{a}_1$, and setting this equal to 0 gives $S\boldsymbol{a}_1 - \lambda_1\boldsymbol{a}_1 = 0$, so that $\boldsymbol{a}_1$ is an eigenvector of S corresponding to the eigenvalue λ_1.

- Providing S is non-singular, S has p non-zero eigenvalues. So which eigenvalue is λ_1?
- If $S\boldsymbol{a}_1 = \lambda_1\boldsymbol{a}_1$, then $\boldsymbol{a}_1'S\boldsymbol{a}_1 = \lambda_1\boldsymbol{a}_1'\boldsymbol{a}_1 = \lambda_1$, so to maximise $\mathbb{V}[X'\boldsymbol{a}_1] = \boldsymbol{a}_1'S\boldsymbol{a}_1$, we must take λ_1 to be the *largest* eigenvalue of S and a_1 as the corresponding eigenvector, and then the maximum value of this variance is λ_1 (which is also the required value of the Lagrange multiplier).

We want the second component to be uncorrelated with the first. This means that we need $\boldsymbol{a}_2'S\boldsymbol{a}_1 = 0$. However, $S\boldsymbol{a}_1 = \lambda_1\boldsymbol{a}_1$, and so this is equivalent to $\boldsymbol{a}_2'\boldsymbol{a}_1 = 0$, i.e., that $\boldsymbol{a}_2$ and $\boldsymbol{a}_1$ are orthogonal.

Let's work out the second principal component: $\boldsymbol{a}_2$ should have the property that projection of X' onto $\boldsymbol{a}_2$ has maximal variance subject to $\boldsymbol{a}_2'\boldsymbol{a}_2 = 1$ and $\boldsymbol{a}_2'\boldsymbol{a}_1 = 0$. We should maximise

$$\Omega_2 = \boldsymbol{a}_2'S\boldsymbol{a}_2 - \mu\boldsymbol{a}_2'\boldsymbol{a}_1 - \lambda_2(\boldsymbol{a}_2'\boldsymbol{a}_2 - 1),$$

where μ and λ_2 are Lagrange multipliers. Differentiating with respect to μ and λ_2 just expresses the constraints. Differentiating with respect to $\boldsymbol{a}_2$ gives

$$2S\boldsymbol{a}_2 - \mu\boldsymbol{a}_1 - 2\lambda_2\boldsymbol{a}_2 = 0.$$

Then

$$2\boldsymbol{a}_1'S\boldsymbol{a}_2 - \mu\boldsymbol{a}_1'\boldsymbol{a}_1 - 2\lambda_2\boldsymbol{a}_1'\boldsymbol{a}_2 = 0$$
$$2\boldsymbol{a}_2'S\boldsymbol{a}_2 - \mu\boldsymbol{a}_2'\boldsymbol{a}_1 - 2\lambda_2\boldsymbol{a}_2'\boldsymbol{a}_2 = 0$$

and we see that

$$\mu = 2\boldsymbol{a}_1'S\boldsymbol{a}_2 = 2(S\boldsymbol{a}_1)'\boldsymbol{a}_2 = 2(\lambda_1\boldsymbol{a}_1)'\boldsymbol{a}_2 = 2\lambda_1\boldsymbol{a}_1'\boldsymbol{a}_2 = 0$$

and so $2S\boldsymbol{a}_2 - 2\lambda_2\boldsymbol{a}_2 = 0$, and $\boldsymbol{a}_2$ is an eigenvector of S with eigenvalue λ_2. Further,

$$2\boldsymbol{a}_2'S\boldsymbol{a}_2 - \mu\boldsymbol{a}_2'\boldsymbol{a}_1 - 2\lambda_2\boldsymbol{a}_2'\boldsymbol{a}_2 = 0,$$

or $\mathbb{V}[X'\boldsymbol{a}_2] = \lambda_2$, so λ_2 must be the second largest eigenvalue of S, with eigenvector $\boldsymbol{a}_2$.

Proceeding in this way, it is easy to see that the same argument shows that

Theorem 5.1 The p principal components of data X' are the p eigenvectors $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_p$ corresponding to the p ordered eigenvalues $\lambda_1 \geq \cdots \geq \lambda_p$ of S, the variance of X'.

There is a slight difficulty if $\lambda_i = \lambda_{i+1} > 0$ since then $\boldsymbol{a}_i$ and $\boldsymbol{a}_{i+1}$ can be chosen in arbitrarily many ways, mutually orthogonal but anywhere in the eigenspace corresponding to $\lambda_i = \lambda_{i+1}$. However, this happens only with probability 0 in practice.

There is another difficulty if $\lambda_p = 0$, since then a_p is not uniquely defined, but this only happens when one variable is a linear combination of the others (very unlikely if $n \geq p$), and in any case, components with $\lambda_p \approx 0$ are unlikely to be interesting.

Remark 5.2 If one of the eigenvalues is 0, you should check the data set carefully. It might be that there are columns which are aggregates of others; for example, a data set consisting of exam scores might contain marks for individual modules, as well as an overall average. Clearly this latter column will be a linear combination of the individual module marks—you should ignore this column in the data set. Similarly, we might be testing the success of a programme to reduce weight, and have measurements of weight before treatment, weight after, and weight loss. Again, one of these columns is redundant and should be removed from the data set. A final alternative is that we have fewer observations than variables—you may remember from linear algebra courses that the number of independent columns and number of independent rows of a matrix are the same, so in this situation the variance matrix will be an $p \times p$ matrix with at most n linearly independent rows, and thus at most n nonzero eigenvalues.

5.3 A simple example – first attempt

We'll go through the procedure for one really simple 2-dimensional toy example, and perform all the procedures by hand for this set. The aim is to ensure that you can see, in a very simple example, how each procedure works.

Here is our data set, consisting of observations (x_i, y_i):

$$(1, 4), \quad (3, 7), \quad (5, 8), \quad (7, 11), \quad (9, 15).$$

We compute the mean of each variable, and see that the mean is $(\bar{x}, \bar{y}) = (5, 9)$. Then we get the variance matrix as $S = \begin{pmatrix} 10 & 13 \\ 13 & 17.5 \end{pmatrix}$. As you can see from the description above, the whole analysis depends only on this variance matrix. It has two eigenvalues, which are computed as $\lambda_1 = 27.2800\ldots$ and $\lambda_2 = 0.2199\ldots$. Thus $\frac{\lambda_1}{\lambda_1+\lambda_2} = 0.992$, and it follows that the first principal component contains 99.2% of the variability in the points. (This reflects the fact that the five points close to lying on a straight line; i.e. they are very highly correlated.)

Let's project the data onto a 1-dimensional space (i.e., a line). The (normalised) eigenvector corresponding to the eigenvalue λ_1 is given by $\boldsymbol{e}_1 = \begin{pmatrix} 0.6011\ldots \\ 0.7991\ldots \end{pmatrix}$, and the line we want is the line whose gradient is given by $\boldsymbol{e}_1$, and which passes through the mean. Since this line is $\boldsymbol{x} = \bar{\boldsymbol{x}} + t\boldsymbol{e}_1$, it is easy, with the help of some simple calculations, to translate each data point $\boldsymbol{x}$ onto the line.

Let's do one example: take $\boldsymbol{x}_1 = (1, 4)'$.

Then we want to work out the corresponding value of t (this should be regarded as the *co-ordinate* of the principal component $\boldsymbol{e}_1$) for $\boldsymbol{x}_1$.

Write $\boldsymbol{x}_1 = \bar{\boldsymbol{x}} + t\boldsymbol{e}_1 + u\boldsymbol{e}_2$. As $\boldsymbol{x}_1 = (1, 4)'$ and $\bar{\boldsymbol{x}} = (5, 9)'$, we have to solve $(-4, -5)' = t\boldsymbol{e}_1 + u\boldsymbol{e}_2$. We could use simultaneous equations, but it is easier to use the orthogonality of $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$; we simply pre-multiply by $\boldsymbol{e}_1'$:

$$\boldsymbol{e}_1'(-4, -5) = t\boldsymbol{e}_1'\boldsymbol{e}_1 + u\boldsymbol{e}_1'\boldsymbol{e}_2 = t.1 + u.0 = t.$$

So we compute $t = e_1'(-4, -5)' = e_1' \begin{pmatrix} -4 \\ -5 \end{pmatrix} = -4 \times 0.6011\ldots - 5 \times 0.7991\ldots = -6.4002\ldots$. We can similarly work out u, by computing $e_2'(-4, -5)'$. However, we do not need this, as we are going to ignore the e_2 component!

We see that x_1 is mapped to $\bar{x} + te_1$ under the transformation, and this is

$$\bar{x} + te_1 = \begin{pmatrix} 1.1522\ldots \\ 3.8854\ldots \end{pmatrix}.$$

This is the point that $(1, 4)'$ is mapped to on the first principal component.

Similarly, we can see where all the points are mapped to:

$$(1, 4) \text{ is mapped to } (1.1522, 3.8854)$$

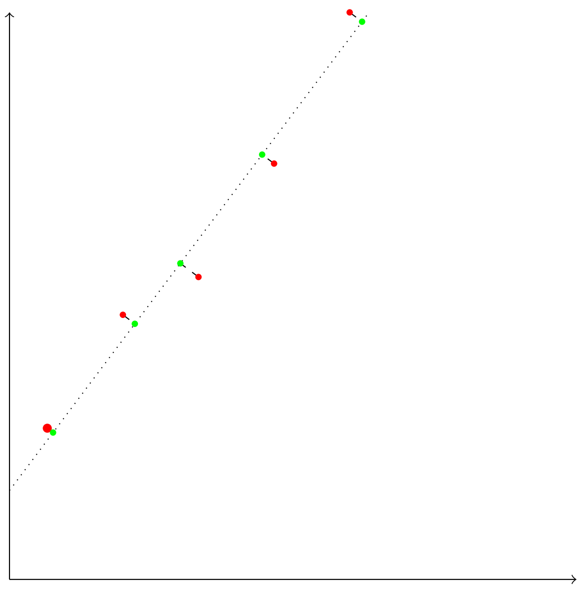
$$(3, 7) \text{ is mapped to } (3.3163, 6.7620)$$

$$(5, 8) \text{ is mapped to } (4.5195, 8.3614)$$

$$(7, 11) \text{ is mapped to } (6.6836, 11.2379)$$

$$(9, 15) \text{ is mapped to } (9.3281, 14.7531)$$

The similarities with lines of best fit should be clear! We are trying to project the data points onto a line so that as little information is lost as possible. To test your understanding, you might like to program this procedure in `R` or `Python` and try to replicate these results.



5.4 Issue – Units: covariance or correlation?

Notice that principal components are not invariant under all linear transformations of original variables; in particular, separate scaling of the variables. For example, principal components of data of people's heights, weights and incomes measured in inches, pounds and pounds sterling will not generally be the same as those on the same data converted to centimetres, kilograms and euros. This means one should be careful in the interpretation of analyses of data on mixed types. This is especially true if the variables have widely different variances, even if they are all of the same type. If one variable has a very large variance, then the first principal component will be dominated by this one variable.

It should be slightly alarming that different choices of units give different principal components (even if the components are scaled to the same units, we get different answers). However, it does tend to be the case that the many different ways allow one to draw relatively similar qualitative conclusions.

Where there is no reason to treat the variables on a similar scale, a natural way to standardise the data is to use the correlation matrix, rather than the variance matrix. This ensures that the co-ordinates are equally spread, and are originally viewed as having the same importance. (It is equivalent to *scaling* the data so that the variance of each variable becomes 1.)

Generally, if data X' are measurements of p variables, all of the same type (e.g., all concentrations of amino acids or all linear dimensions in the same units, but not, for example, age/income/weights), then the coefficients of principal components can be interpreted as "loadings" of the original variables and hence the principal components can be interpreted as contrasts in the original variables, as with some of the examples later.

This interpretation is less easy if the data are of mixed "types" or if the variables have widely different variances, even if the eigenanalysis is based on the correlation rather than the variance matrix.

In practice, however, it tends to be the case that one of the variables has much larger variance than another, or that differing scales are used, and it is the correlation matrix which is most often used (indeed, it is the default in many computer packages).

However, each individual data set should be considered individually; there is scope for adjusting the weightings associated to each variable.

Let's take the same data set as above, but do the PCA using the correlation matrix instead of the variance matrix. Here are the points:

$$(1,4),\quad (3,7),\quad (5,8),\quad (7,11),\quad (9,15).$$

Step 1 Compute the correlation matrix. This is $\begin{pmatrix} 1 & 0.9827 \\ 0.9827 & 1 \end{pmatrix}$ (since the points are so nearly collinear, they are very highly correlated).

Step 2 Calculate the *eigenvectors and eigenvalues* of the correlation matrix.

Let's work out the characteristic polynomial:

$$\det(S - \lambda I) = \det \begin{pmatrix} \lambda - 1 & -0.98270 \\ -0.98270 & \lambda - 1 \end{pmatrix} = \lambda^2 - 2\lambda + 0.0343,$$

and the roots (the eigenvalues) are 1.9827 and 0.0173. The eigenvector corresponding to the eigenvalue of 1.9827 is $\begin{pmatrix}1\\1\end{pmatrix}$ and the eigenvector corresponding to the eigenvalue of 0.0173 is $\begin{pmatrix}1\\-1\end{pmatrix}$. It is natural (and useful) to normalise these vectors to have length 1, so that the eigenvectors will be $\begin{pmatrix}1/\sqrt{2}\\1/\sqrt{2}\end{pmatrix}$ and $\begin{pmatrix}1/\sqrt{2}\\-1/\sqrt{2}\end{pmatrix}$.

(When there are only 2 columns of data, these are always the eigenvectors, as you might like to explain, but things get more interesting with more columns!)

Step 3 Choose the most *important* eigenvectors.

Clearly the dominant eigenvector is $\boldsymbol{e}_1 = \begin{pmatrix}1/\sqrt{2}\\1/\sqrt{2}\end{pmatrix}$, we shall ignore the other.

Step 4 Convert the data to the principal components.

We have some data, and now we've chosen the principal components. We need to compress our data and give it in terms of these eigenvectors.

We do the same as we did with the variance matrix. There's a slight subtlety because of the scaling implicit in the data to get from the covariances to the correlations. We'll do the first point again, before giving all the results.

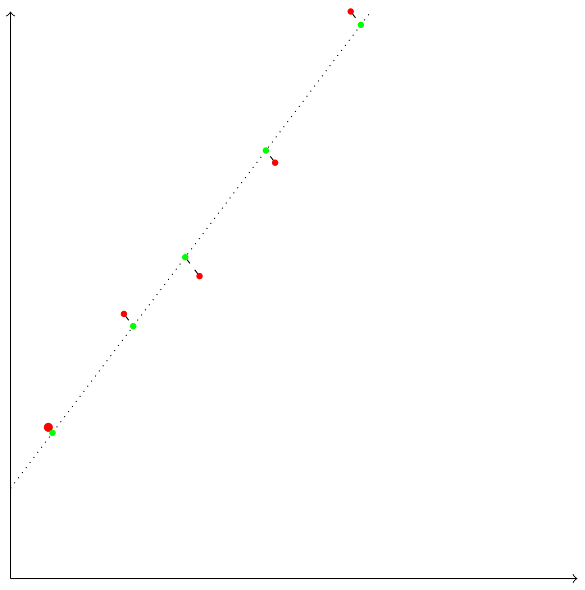
The easiest way is to start with $\boldsymbol{x}_1$, and subtract the mean, to get a point $(1, 4)-(5, 9) = (-4, -5)$. Then scale the first coordinate by its standard deviation, and the second by its standard deviation. That is, we scale all the first co-ordinates by $1/\sqrt{10}$, and all the second by $1/\sqrt{17.5}$, so that they have variance 1.

Now the shifted first point is $(-4/\sqrt{10}, -5/\sqrt{17.5}) = (-1.2649, -1.1952)$.

Then we want to write this in terms of the two eigenvectors, and especially to calculate the contribution of the main eigenvector. So we compute $e_1'(-1.2649, -1.1952) = -1.7396$, and take this multiple of $(1/\sqrt{2}, 1/\sqrt{2})$, to get $(-1.2301, -1.2301)$.

Now we translate back, by scaling the first coordinate back by $\sqrt{10}$ and the second by $\sqrt{17.5}$, and then add the mean $(5, 9)$ to get $(1.1102, 3.8542)$, and this is the projection we want.

$(1, 4)$	is mapped to $(1.1102, 3.8542)$
$(3, 7)$	is mapped to $(3.2441, 6.6771)$
$(5, 8)$	is mapped to $(4.6220, 8.5)$
$(7, 11)$	is mapped to $(6.7559, 11.3229)$
$(9, 15)$	is mapped to $(9.2679, 14.6458)$



You might notice that the projection lines don't seem to be orthogonal to the principal component—you should think why this is a consequence of the scaling of the coordinates.

In this example, the two computations of the principal components didn't lead to very different answers (but they are definitely different!). The reason is that the second component was almost negligible. In examples where components other than the first have more significance, or where the variances of the variables are more widely different, the answers can be quantitatively somewhat different. In general, this does not seem to affect the analysis qualitatively, but the data analyst needs to be aware of this issue.

5.5 Principal components analysis

So here is the algorithm, starting with a data frame:

1. Decide whether to use the variance or correlation matrix for the data frame. If the variables are in different units, or if some variables have substantially greater variance than others, use the correlation matrix. *This will almost always be the case!*
2. Work out the eigenvalues and eigenvectors of the variance or correlation matrix. List the eigenvectors in decreasing order of the eigenvalues.
3. Change the basis from the original variables into the linear combinations which are the eigenvectors.

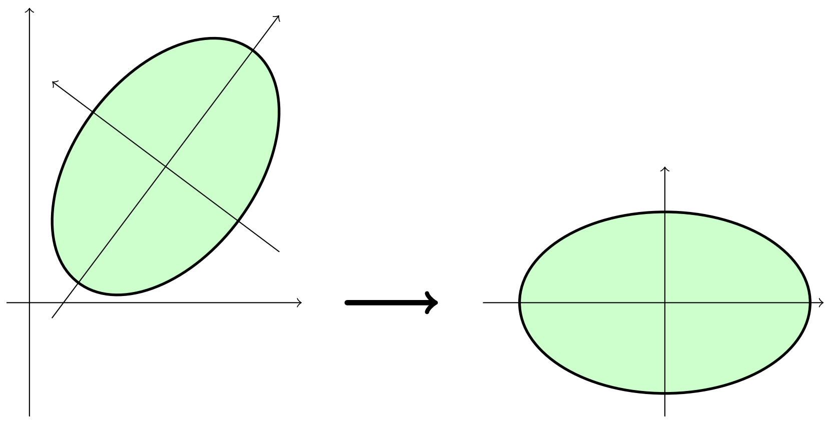
It is valuable to get some idea of what PCA is really doing geometrically, and to see also the role played by the variance matrix in the geometry of the sample.

We can explain this most easily in 2 dimensions. We begin by noting that multivariate normal distribution throws up data which is elliptical in shape. (Look at the contours for the distribution.)

We might imagine that the multivariate normal distribution is probably a reasonable approximation for a lot of random data that arises in practice.

More precisely, the data is often quite well approximated by an ellipse whose axes are the eigenvectors of the variance matrix, with corresponding lengths proportional to the square roots of the eigenvalues.

PCA simply finds the axes of the ellipse, and rotates all the data so that the new basis lies along these axes.



The same happens in more dimensions—the data is often approximately ellipsoidal in shape, where the axes are given by the eigenvectors of the variance matrix, with lengths proportional to the square roots of the eigenvalues, and PCA rotates the ellipsoid onto the standard co-ordinate axes.

Generally, if the columns are quite closely correlated, one would expect the first principal component to contain much of the information in the data set, so PCA is particularly effective in this case—we can replace the correlated columns with a single column corresponding to some pattern accounting for the correlation between the columns. So one initial test might be to compute some correlations between columns; if the columns tend to be well correlated, one would expect that fewer principal components would be required. Let's say a little more.

The principal components $\boldsymbol{a}_1, \dots, \boldsymbol{a}_p$ provide a useful coordinate system to which to refer the data and for displaying them graphically, i.e., better than the original coordinates

$$\begin{pmatrix}1\\0\\0\\\vdots\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\\\vdots\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\\\vdots\\0\end{pmatrix}, \dots, \begin{pmatrix}0\\0\\0\\\vdots\\1\end{pmatrix}$$

Let $A = (\boldsymbol{a}_1, \dots, \boldsymbol{a}_p)$, be the $p \times p$ matrix with jth column $\boldsymbol{a}_j$. As the columns are length 1 eigenvectors of a real symmetric matrix, we see that $A'A = \mathrm{I}_p$, so that A is an orthogonal matrix.

Then the projection of the data X' onto this coordinate system is $Y' = X'A$. Since A is orthogonal, the transformation $X' \longrightarrow Y'$ is a rotation/reflection (i.e., no overall change of scale or origin).

If we measure the "total" variation in the original data as the sum of the variances of the original components, i.e.,

$$s_{11} + s_{22} + \cdots + s_{pp} = \mathrm{tr}(S),$$

then the "total" variation of Y' is $\lambda_1 + \cdots + \lambda_p$, and $\sum_i \lambda_i = \mathrm{tr}(S)$ as the trace of a matrix is the same as the sum of its eigenvalues. (One can view this as a kind of "conservation of total information" in the PCA process.)

So we can interpret $\lambda_1 / \sum \lambda_i$ as the proportion of the total variation "explained" by the first principal component, and similarly

$$\frac{\lambda_1 + \cdots + \lambda_k}{\lambda_1 + \cdots + \lambda_p}$$

is the proportion of the total variation explained by the first k principal components etc.

If the first few principal components explain most of the variation in the data, then the later principal components are redundant and little information is lost if they are discarded or ignored.

Quite how much data we choose to retain depends on the data and area of application. Some areas might be satisfied if we choose enough components to make up 40% of the total variation, while others might require 95%. A figure needs to be chosen as a trade-off between the convenience of a small value of k and a large value of the cumulative relative proportion of variance explained.

If p is large, an informal way of choosing a good k is graphically, with a *scree plot*, plotting j against $(\sum_1^j \lambda_i)/\,\mathrm{tr}(S)$ against j. The graph will be monotonic and convex; it will typically increase steeply for the first few values of j and then begin to level off. The point where it starts levelling off is the point where bringing in more principal components brings less returns in terms of variance explained.

5.6 Relation to linear models

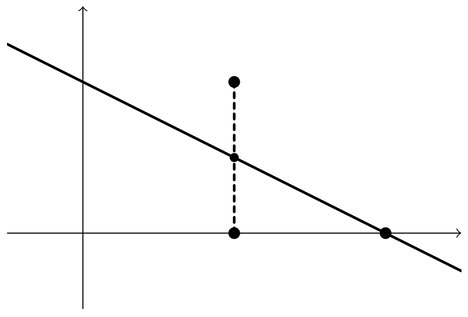
In many ways, the ideas are closely related to linear models, which is probably familiar to readers; in linear regression, we try to find a line (or hyperplane in more dimensions) such that the sum of squares of the *vertical* distances from the data points to the line/hyperplane is minimised—this is the sum of squares error. In PCA, we try to find a line/hyperplane which fits the data best, and one formulation is that we are trying to find the line/hyperplane such that the sum of squares of the distances from the data points to the line/hyperplane is minimised. Here is a quick toy example:

Example 5.2 Let's suppose that we have data points $(1, 0)$, $(1, 1)$ and $(2, 0)$. Then if we regress the y-coordinate on the x-coordinate, we are trying to find a line in the plane such that the sum of squares of the vertical distances from the points to line is minimised:

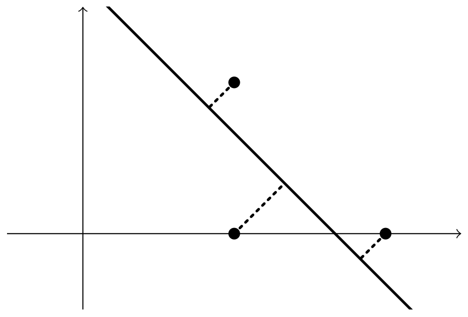
The line has slope -0.5, and minimises the sum of squares error (the sum of the squares of the lengths of the dashed vertical line segments) as:

$$(0 - 0.5)^2 + (1 - 0.5)^2 + (0 - 0)^2 = 0.5.$$

The PCA picture is similar, except that we use the perpendicular distances to the line. By



symmetry, the line should be $x + y = c$ for some c, and it is an easy calculation to see that $c = 2/3$ minimises the sum of squares error:



The total sum of squares error here (the sum of squares of the lengths of the dotted line segments) is easily seen to be $1/3$.

PCA has a smaller error, but it's trying to fit a different model, where no variable is meant to be depending on any other, and simply trying to fit all the data most closely.

5.7 Example: Turtle data

Let's look now at a couple of data sets referred to in literature. The first of these is taken from a paper "Size and shape variation in the painted turtle. A principal component analysis", by Jolicoeur and Mosimann (1960). (I haven't managed to track down a complete data set for either of these, although the variance/correlation matrix is given, and this suffices for working out the principal components.)

Measurements are taken in millimetres of length, width and height of shells of 24 female painted turtles (*Chrysemys picta marginata*) collected in a single day in the St Lawrence valley. To have $p = 3$ is unusually small for real life data sets, but it illustrates some of the calculations and interpretations. The sample variance matrix is: $S = \begin{pmatrix} 451.39 & 271.17 & 168.70 \\ 271.17 & 171.73 & 103.29 \\ 168.70 & 103.29 & 66.65 \end{pmatrix}$, with trace 689.77. Then

	a_1	a_2	a_3
length	0.8126	−0.5454	−0.2054
width	0.4955	0.8321	−0.2491
height	0.3068	0.1008	0.9465
variance (λ_j)	680.4	6.5	2.86

Checks: the sum of variances is 689.76, and we can check easily that $S\boldsymbol{a}_i - \lambda_i \boldsymbol{a}_i = 0$.

The first component accounts for 98.64% of the total variance. This is typical of data on *sizes* of items, and it reflects variation in overall sizes of the turtles. Its value for any particular turtle (or score on the first principal component) is

$$Y_1 = 0.81 \times \text{length} + 0.50 \times \text{width} + 0.31 \times \text{height},$$

and reflects the general size. The second principal component is

$$Y_2 = -0.55 \times \text{length} + 0.83 \times \text{width} + 0.10 \times \text{height},$$

and this component is dominated by variations in length and width, and is therefore a contrast between them. High values of Y_2 will come from nearly round shells and low values from elongated elliptical shells. So Y_2 is a measure of "roundedness". Finally,

$$Y_3 = -0.21 \times \text{length} - 0.25 \times \text{width} + 0.95 \times \text{height},$$

so this contrasts height against (length + width), so is a measure of how peaked or pointed the shell is. The conclusions are that the shells vary most in overall size, next most in shape, and lastly in "peakedness".

Exercise 5.1 Verify the calculations in `Python` or `R` (remember that you don't need the full data set, just the sample variance matrix), and repeat them but using the correlation matrix instead of the variance matrix.

If you wish to investigate this further, you might like to note that the paper mentioned above also gives data for 24 males for further analysis.

5.8 Example: Chicken dimensions (Wright)

The data for this example are given in Wright (1954), "The interpretation of multivariate systems", in *Statistics and Mathematics in Biology* (State University Press, Iowa, eds. O.Kempthorne, T.A.Bancroft, J.W.Gowen, J.L.Lush), 11–33.

Six bone measurements ($x_1, \ldots, x_6$) were made on each of 275 white leghorn fowl. These were x_1: skull length, x_2: skull breadth, x_3: humerus, x_4: ulna, x_5: femur, x_6; tibia (so the first two were skull measurements, the next two were wing measurements, and the last two were leg measurements). The table below gives the coefficients of the six principal components calculated from the variance matrix.

variable	$\mathbf{a}_1$	$\mathbf{a}_2$	$\mathbf{a}_3$	$\mathbf{a}_4$	$\mathbf{a}_5$	$\mathbf{a}_6$
$\boldsymbol{x}_1$	0.35	0.53	0.76	−0.04	0.02	0.00
$\boldsymbol{x}_2$	0.33	0.70	−0.64	0.00	0.00	0.03
$\boldsymbol{x}_3$	0.44	−0.19	−0.05	0.53	0.18	0.67
$\boldsymbol{x}_4$	0.44	−0.25	0.02	0.48	−0.15	−0.71
$\boldsymbol{x}_5$	0.44	−0.28	−0.06	−0.50	0.65	−0.13
$\boldsymbol{x}_6$	0.44	−0.22	−0.05	−0.48	−0.69	0.17

Note that different packages may reverse all the signs in one of the columns; recall that if $\boldsymbol{x}_i$ is an eigenvector, so is $-\boldsymbol{x}_i$, so there is some arbitrariness in the sign.

To interpret these coefficients, we will round them heavily to either just one decimal place, and ignore values close to 0, giving:

variable	$\boldsymbol{a}_1$	$\boldsymbol{a}_2$	$\boldsymbol{a}_3$	$\boldsymbol{a}_4$	$\boldsymbol{a}_5$	$\boldsymbol{a}_6$
x_1	0.4	0.6	0.7	0	0	0
x_2	0.4	0.6	−0.7	0	0	0
x_3	0.4	−0.2	0	0.5	0	0.7
x_4	0.4	−0.2	0	0.5	0	−0.7
x_5	0.4	−0.2	0	−0.5	0.6	0
x_6	0.4	−0.2	0	−0.5	−0.6	0

and then consider the signs:

variable	$\mathbf{a}_1$	$\mathbf{a}_2$	$\mathbf{a}_3$	$\mathbf{a}_4$	$\mathbf{a}_5$	$\mathbf{a}_6$
$\boldsymbol{x}_1$	$+$	$+$	$+$			
$\boldsymbol{x}_2$	$+$	$+$	$-$			
$\boldsymbol{x}_3$	$+$	$-$		$+$		$+$
$\boldsymbol{x}_4$	$+$	$-$		$+$		$-$
$\boldsymbol{x}_5$	$+$	$-$		$-$	$+$	
$\boldsymbol{x}_6$	$+$	$-$		$-$	$-$	

We can then see that the first component $\boldsymbol{a}_1$ is proportional to the sum of all the measurements. Large fowl will have all $\boldsymbol{x}_i$ large, and so their scores on the first principal component $y_1(=\boldsymbol{x}'\boldsymbol{a}_1)$ will be large; similarly small birds will have low scores of y_1. If we produce a scatter plot using the first principal component as the horizontal axis, then the large birds will appear on the right-hand side, and small ones on the left. Thus the first principal component measures *overall size*.

The second component is of the form (skull) − (wing + leg), and so high positive scores of $y_2(=\boldsymbol{x}'\boldsymbol{a}_2)$ will come from birds with large heads and small wings and legs. If we plot y_2 against

y_1, then the birds with relatively small heads for the size of their wings and legs will appear at the bottom, and those with relatively big heads at the top. So the second principal component measures *overall body shape*.

The third component is a measure of *skull shape* (i.e., skull width against skull length). The fourth is wing size compared with leg size, so is another measure of *body shape* (but one not involving the head). The fifth and sixth are contrasts between upper and lower parts of the wing and leg respectively, so y_5 measures *leg shape* and y_6 measures *wing shape*.

For comparison, we see the effect of using the correlation matrix for the principal component analysis instead of the variance matrix. The correlation matrix is:

	x_1	x_2	x_3	x_4	x_5	x_6
x_1	1.000	0.505	0.569	0.602	0.621	0.603
x_2	0.505	1.000	0.422	0.467	0.482	0.450
x_3	0.569	0.422	1.000	0.926	0.877	0.878
x_4	0.602	0.467	0.926	1.000	0.874	0.894
x_5	0.621	0.482	0.877	0.874	1.000	0.937
x_6	0.603	0.450	0.878	0.894	0.937	1.000

Then we get the principal components as:

variable	a_1	a_2	a_3	a_4	a_5	a_6
x_1	0.35	0.40	0.85	0.05	0.01	−0.03
x_2	0.29	0.81	−0.50	0.02	0.01	−0.04
x_3	0.44	−0.26	−0.11	0.50	0.60	−0.33
x_4	0.45	−0.20	−0.10	0.47	−0.60	0.41
x_5	0.45	−0.16	−0.10	−0.50	0.37	0.58
x_6	0.45	−0.21	−0.07	−0.47	−0.38	−0.62
eigenvalue	4.46	0.78	0.46	0.17	0.08	0.05

Comparing these results with those from the variance matrix, note that the overall interpretation of the individual components will be exactly the same, except that the last two swap.

Note that the sum of the eigenvalues is 6.0 (the correlation matrix had trace 6.0, since it is a 6×6 matrix with 1 in each diagonal entry). The first component accounts for almost 75% of the variance of the data—strictly speaking, it is 75% of the standardised data, and not the original data. In fact, if the analysis were based on the variance matrix, then the first principal component would appear even more dominant; this is typical with measurements of physical sizes. For dimensionality reduction in particular, it is sensible to base assessments on the analysis of the correlation matrix, or else look at the proportions of variance explained as a total of all the components *excluding* the first (you may not care about size so much as differences in species, for example, where differences in the shape of the animal may be more important than the size, which may primarily be an indicator of the age of the animal). The first three components account for 95% of the variance of the standardised data, and this might be a satisfactory initial reduction to these three components.

5.9 Remarks

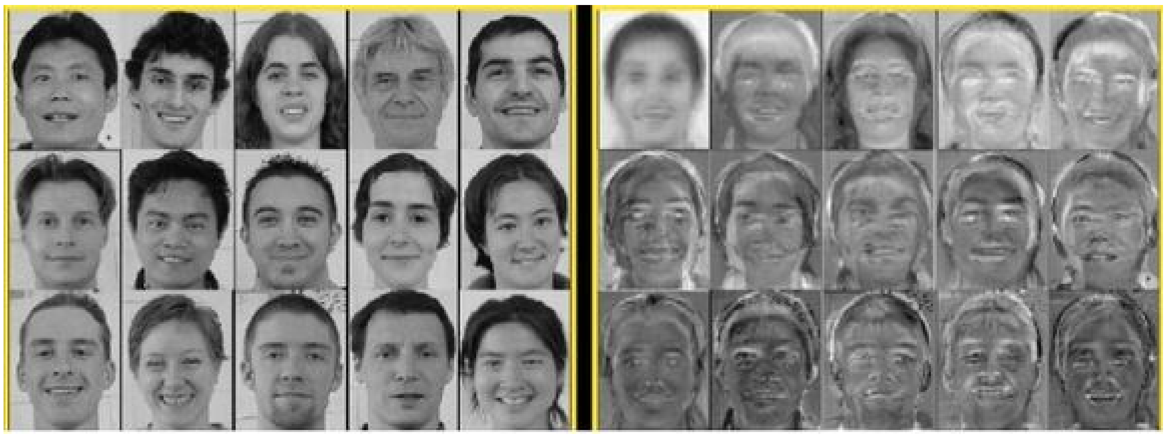
- "Experience" shows that if you include a few discrete variables with several continuous ones (or if you have a large number of discrete ones without any continuous ones), then principal component analysis based upon the correlation matrix works, i.e., you obtain interpretable answers.
- Since the key objective of principal components analysis is to extract information, i.e., to *partition the internal variation*, it is sensible to plot the data using equal scaling on the axes. This gives a more realistic view of the data, which would otherwise be stretched more on a square plot in the y-direction.
- *Supplementary data* (i.e., further data not included in calculating the principal component transformation but measured on the same variables) can be displayed on a PCA plot (i.e., apply the same transformation to the new data).
- Numerical interpretation of loadings is only viable with a small number of original variables. In many modern examples with several hundred variables, other techniques have to be used, e.g., plot loadings against variable number (assuming that there is some structure to the variable numbering, e.g., digitising spectra). This may reveal, for example, that the first few principal components are combinations from the middle of the spectra, next few from one third of the way along.
- PCA is obtained by choosing projections to maximise variances of projected data. Choosing a different criterion can give interesting views of data: one might, for example, regard "typical" data as being Gaussian, and regard projections which are highly non-Gaussian as being particularly interesting. One way to do this is with a quantity known as *kurtosis*. Those interested should consult literature on *projection pursuit methods*, and *independent component analysis* (a special case).

5.10 Application: Facial recognition systems

Since the late-1980s, PCA has been used for facial recognition systems in the field of machine learning.

We begin with a training set consisting of images of faces, all of the same resolution (which need not be particularly large), and where, ideally, the images are cropped so that the faces are centred and of approximately the same size. Usually the images are converted into black and white, so that every pixel is a single number between, say, 0 and 255. Thus a set of 100×100 pixel images will be encoded as a vector of $p = 100^2$ numbers between 0 and 255.

A PCA is performed on these sets. The eigenvectors are going to be vectors of the same size as the digitised images. When regarded as images, they often look a little like faces, and these are known as *eigenfaces*. Here is one example, where a training set of 15 faces is transformed into eigenfaces.



Generally one would use at least several hundred faces in the training set, rather than just 15.

We store the first few principal components (i.e., eigenfaces).

Now we need to apply this. Suppose we have a test set of people we wish to store in our identification system. We begin by taking images of each, and projecting them onto the first principal components. That is, we find linear combinations of the first principal components which resemble the image, and store the coefficients involved. Thus every person in the test set is encoded as a vector of perhaps 40 numbers (if we take, say, the first 40 eigenfaces).

Given a new observation, we do the same thing, reducing it to a vector of about 40 numbers. Then we ask whether it is "close" to any of those in our library for the test set.

5.11 Further reading

PCA is the most widely used dimensionality reduction technique, but there are others which may be of interest. More details are easily accessible via an internet search, for example. Some, like *projection pursuit* and *independent component analysis* were referred to above already.

But one should mention *factor analysis*. This is a method which supposes that the p observed variables are linear combinations of q underlying, or *latent* variables, at least approximately: we assume that the difference between each of the p observed variables and linear combinations of the latent variables is modelled by an error term which has a normal distribution, and we then try to minimise the total error terms. This then determines the "best" choice of the underlying variables. Again, the value of q is open to choice; unlike PCA, the results for a given value of q will differ completely if $q + 1$ is used instead. This method is widely used in social science, where researchers have tried to explain observed numerical variables by interpreting the underlying variables, but these interpretations have sometimes been controversial.

We shall return to unsupervised learning at the end of the module, and see further examples of dimensionality reduction techniques, based on nonlinear ideas. In particular, we shall mention *isomap* and *stochastic neighbour embeddings* as much more local (but nonlinear) techniques for visualising data in high dimensions.

We will now turn attention to **supervised learning** problems.

6 Supervised learning – regression

This chapter has two aims: firstly, to remind the reader briefly of linear regression, and to use this to motivate discussion of some of the main issues in machine learning.

You will all be familiar with the concept of linear models and regression; the idea is that, given data, we can model certain variables (the *response* variables) in terms of some inputs, using a linear function. The linear function has some unknown parameters, but we can read off the "best" choices for the parameters using a fairly simple formula.

This is an example of what is known as *supervised learning* in the machine learning community. We have some data, and each data point has some measurable response. The response might be numerical, like in linear regression, but can also be categorical. For example, we might want to measure whether a patient has a disease or not, depending on the numerical readings; then the output is either *Disease present* or *Disease not present*.

When the response variable is numerical, we call this a *regression* problem, and when it is categorical (often binary), we call it a *classification* problem; the issue then comes down to determining which of the (two of more) classes to assign the new data point.

In supervised learning, probably the main problem is that of using existing data to learn to make predictions about future readings. So we might want to develop a test to measure the presence of a disease, or to predict income of someone from their spending patterns, etc. Another problem is to study the relation between inputs and outputs to get more information about their correspondence. These goals, of prediction and inference, tend to be at the heart of supervised learning problems.

Based on an exploratory data analysis, probably using visualisations of the sort seen in previous chapters, we might try to make some sort of model for the known data. For example, we might predict a linear model, as above, if we have a numerical response variable. Then we can apply it to new data and see how it performs.

There is a trade-off between the complexity of the model in a regression problem and the mean square error. However, more complicated models tend to over-reflect for noise in the training data, and this *overfitting* means that the real features of the data can be obscured, and predictions on test data are distorted. Conversely, if models are too simple, then they will also not do that well on test data either, because of the deficiencies in the assumptions of the model.

Since linear models and linear regression is probably the most well-known topic in supervised learning, and is well treated in other modules, I shall say rather little about it. We will merely point out that the multivariate (linear) regression problem is easily formulated in matrices, and that the solution is essentially derived in exactly the same way—all that happens is that numbers get replaced by vectors or matrices. However, partly because it is quite familiar, we can use it as a technique for which we can discuss some of the important issues in machine learning, and which we will meet again in future techniques.

6.1 Theory of linear regression

Now that we have developed the theory of the multivariate normal distribution, we can explain the generalisation of regression analysis to the multivariate case.

There is little new in terms of statistical concepts, and the univariate mathematical development carries through almost unchanged symbolically to the multivariate case; we will see that this generalisation actually requires no new mathematics—we will be very brief! Formal statistical tests are not described here, but are essentially applications of multivariate analysis of variance.

First, recall the univariate case, with just one independent variable x. The parameters α and β in the model $\mathbb{E}[Y] = \alpha + \beta x$, or $y_i = \alpha + \beta x_i + \varepsilon_i$, with $\varepsilon_i \sim \mathrm{N}(\varepsilon_i \mid 0, \sigma^2)$ independent, have least squares estimates (and MLE) obtained by minimising $\sum_{i=1}^{n}(y_i - \widehat{\alpha} - \widehat{\beta} x_i)^2$ which yields

$$\widehat{\alpha} = \bar{y} - \widehat{\beta}\bar{x}$$
$$\widehat{\beta} = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_k - \bar{x})^2}$$

Note that the sample correlation coefficient is slightly different,

$$\rho_{xy} = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum(x_i - \bar{x})^2 \sum(y_i - \bar{y})^2}}.$$

Note also that the correlation coefficient is symmetric in x and y, but the regression coefficient is not, emphasising that the correlation coefficient is used for investigating the relationship between the x and y variables, whereas regression analysis investigates the dependence of the dependent variable y on the independent variable x (perhaps with the aim of predicting one from the other).

If there are several independent variables, $x_1, \ldots, x_q$, then the appropriate model can be written as $\mathbb{E}[\boldsymbol{y}] = X\boldsymbol{\beta}$ or $\boldsymbol{y} = X\boldsymbol{\beta} + \boldsymbol{\varepsilon}$, where $\boldsymbol{\varepsilon} \sim \mathrm{N}_n(\boldsymbol{\varepsilon} \mid \mathbf{0}, \sigma^2 \mathrm{I})$, and $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_q)$, and X is the $n \times q$ matrix of observations of the q independent variables (and usually the first $\boldsymbol{x}_1$ would be taken to be the unit vector $\mathbf{1}_n$, so the model includes a constant term).

Minimising the sum of squares of the residuals, i.e., $\boldsymbol{\omega} = (\boldsymbol{y} - X\boldsymbol{\beta})'(\boldsymbol{y} - X\boldsymbol{\beta})$, yields after differentiating $\boldsymbol{\omega}$ with respect to $\boldsymbol{\beta}$:

$$2X'(\boldsymbol{y} - X\boldsymbol{\beta}) = 0$$

and thus $\widehat{\beta} = (X'X)^{-1}X'\boldsymbol{y}$ (provided $n > p$, or more precisely that $X'X$ is nonsingular). The extension to the case when Y is a matrix of vector observations is now straightforward.

The p-dimensional multivariate linear model is $\mathbb{E}[Y] = X\boldsymbol{\beta}$ or $Y = XB + \Omega$, where Y is an $n \times p$ matrix of n observations of p-dimensional variables, X is $n \times q$, B is $q \times p$ and Ω is an $n \times p$ matrix random variable. The matrix of residual sums of squares and cross products is $W = (Y - XB)'(Y - XB)$; differentiating W with respect to B yields $2X'(Y - XB) = 0$, and thus

$\widehat{B} = (X'X)^{-1}X'Y$ (provided $n > p$, or more precisely that $X'X$ is nonsingular). Note that W is a matrix, so its differentiation may need to be taken on trust, as also the fact that this minimises both the determinant and the trace of W. This means that the p individual $\widehat{B}_i$ in $\widehat{B}$ are identical to those which would be obtained by separate multiple regressions of each $\boldsymbol{y}_i$ on the independent variables $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_q$. This is true of least squares estimation and of maximum likelihood estimation if the errors are assumed to be Gaussian with independence from one observation to the next, though they could be correlated between one variable to the next on the same observation, i.e., that the rows of Ω are i.i.d $\mathrm{N}_n(\boldsymbol{\omega}_j \mid \mathbf{0}, \Sigma)$. Predicted values of Y are given by $\widehat{Y} = X\widehat{B}$ and Σ is estimated as $\widehat{\Sigma} = (Y - X\widehat{B})'(Y - X\widehat{B})/(n-q-1)$.

Example 6.1 ***Cox*** This data concerns the price and consumption of pork and beef, and is at http://lib.stat.cmu.edu/DASL/Datafiles/agecondat.html:

YEAR	PBE	CBE	PPO	CPO	PFO	DINC	CFO	RDINC	RFP
1925	59.7	58.6	60.5	65.8	65.8	51.4	90.9	68.5	877
1926	59.7	59.4	63.3	63.3	68.0	52.6	92.1	69.6	899
1927	63.0	53.7	59.9	66.8	65.5	52.1	90.9	70.2	883
1928	71.0	48.1	56.3	69.9	64.8	52.7	90.9	71.9	884
1929	71.0	49.0	55.0	68.7	65.6	55.1	91.1	75.2	895
1930	74.2	48.2	59.6	66.1	62.4	48.8	90.7	68.3	874
1931	72.1	47.9	57.0	67.4	51.4	41.5	90.0	64.0	791
1932	79.0	46.0	49.5	69.7	42.8	31.4	87.8	53.9	733
1933	73.1	50.8	47.3	68.7	41.6	29.4	88.0	53.2	752
1934	70.2	55.2	56.6	62.2	46.4	33.2	89.1	58.0	811
1935	82.2	52.2	73.9	47.7	49.7	37.0	87.3	63.2	847
1936	68.4	57.3	64.4	54.4	50.1	41.8	90.5	70.5	845
1937	73.0	54.4	62.2	55.0	52.1	44.5	90.4	72.5	849
1938	70.2	53.6	59.9	57.4	48.4	40.8	90.6	67.8	803
1939	67.8	53.9	51.0	63.9	47.1	43.5	93.8	73.2	793
1940	63.4	54.2	41.5	72.4	47.8	46.5	95.5	77.6	798
1941	56.0	60.0	43.9	67.4	52.2	56.3	97.5	89.5	830

where

- `PBE` is the price of beef (cents/lb)
- `CBE` is the consumption of beef per capita (lbs)
- `PPO` is the price of pork (cents/lb)
- `CPO` is the consumption of pork per capita (lbs)
- `PFO` is the retail food price index (1947-1949=100)
- `DINC` is the disposable income per capita index (1947-1949==100)
- `CFO` is the food consumption per capita index (1947-1949==100)
- `RDINC` is the index of real disposable income per capita (1947-1949=100)
- `RFP` is the retail food price index adjusted by the CPI (1947-1949=100)

For this illustration, we will consider the dependence of consumption of beef and pork, i.e., $Y = [\texttt{CBE}, \texttt{CPO}]$ upon the prices of beef and pork and disposable income, and include a constant term in the regression, so $X = [1_{17}, \texttt{PBE}, \texttt{PPO}, \texttt{DINC}]$.

Thus $p = 2$, $q = 4$ and $n = 17$.

We find:

$$Y = \begin{pmatrix} 58.6 & 65.8 \\ 59.4 & 63.3 \\ 53.7 & 66.8 \\ 48.1 & 69.9 \\ 49.0 & 68.7 \\ 48.2 & 66.1 \\ 47.9 & 67.4 \\ 46.0 & 69.7 \\ 50.8 & 68.7 \\ 55.2 & 62.2 \\ 52.2 & 47.7 \\ 57.3 & 54.4 \\ 54.4 & 55.0 \\ 53.6 & 57.4 \\ 53.9 & 63.9 \\ 54.2 & 72.4 \\ 60.0 & 67.4 \end{pmatrix}, \qquad X = \begin{pmatrix} 1 & 59.7 & 60.5 & 51.4 \\ 1 & 59.7 & 63.3 & 52.6 \\ 1 & 63.0 & 59.9 & 52.1 \\ 1 & 71.0 & 56.3 & 52.7 \\ 1 & 71.0 & 55.0 & 55.1 \\ 1 & 74.2 & 59.6 & 48.8 \\ 1 & 72.1 & 57.0 & 41.5 \\ 1 & 79.0 & 49.5 & 31.4 \\ 1 & 73.1 & 47.3 & 29.4 \\ 1 & 70.2 & 56.6 & 33.2 \\ 1 & 82.2 & 73.9 & 37.0 \\ 1 & 68.4 & 64.4 & 41.8 \\ 1 & 73.0 & 62.2 & 44.5 \\ 1 & 70.2 & 59.9 & 40.8 \\ 1 & 67.8 & 51.9 & 43.5 \\ 1 & 63.4 & 41.5 & 46.5 \\ 1 & 56.0 & 43.9 & 56.3 \end{pmatrix}$$

The least squares estimate of B is

$$\hat{\beta} = \begin{pmatrix} 101.40 & 79.60 \\ -0.753 & 0.153 \\ 0.254 & -0.687 \\ -0.241 & 0.283 \end{pmatrix}.$$

You should check these calculations, and also check that the same answers would be obtained with separate univariate linear regressions.

The advantage of the multivariate approach is that the correlation between pork and beef consumption is modelled, and this allows construction of simultaneous confidence intervals.

Σ is estimated as $\hat{\Sigma} = \begin{pmatrix} 4.412 & -7.572 \\ -7.572 & 16.835 \end{pmatrix}$.

- Multivariate regression models the dependence of a p-dimensional random variable Y on a q-dimensional variable X.
- The individual equations relating the p individual Y variables on the X variables are identical to those obtained by separate univariate regressions of the $\boldsymbol{Y}_i$ on the X variables.
- Formal tests of hypothesis (e.g., whether $B = 0$) are available.

- It does not matter whether $q < p$ or $q > p$.
- It is required that $n > \max(p, q)$ and that $X'X$ is non-singular.

Whenever we introduce a model, such as a linear regression model, we need to understand how well it does. Certainly, we need to know how it does on the data set we fitted the model to (or "trained" the model on, in machine learning language). If we hope to use the model for future prediction, we will also need to estimate how well the model will do for data which is not used in the training. This leads to various issues in machine learning.

6.2 Model performance: regression

Having built our model from a data set, we want to know how good it is. This vague question can be interpreted in several ways. For example,

1. How robust is it?
2. How accurate is it?

For the first question, we might be interested to know, for example, how much variability the model has. For example, if we were to delete one member of the training set, repeat the method of fitting the model, and get something very different, that would tell us something interesting about the instability of the method of fitting the model, or that the member which was deleted was an outlier.

It is more likely that the second question is more interesting, but this already breaks up into two parts—how accurate is it for the training set? how accurate is it likely to be for new data? We will consider the problem of predicting outputs for unseen data later in the chapter.

A natural measure for how well the model does on the training set is given by the mean of the square of the differences between the actual values and the values predicted by the model. That is, the *mean square error* is given by

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \widehat{y}_i)^2,$$

where y_i is the actual value of the variable for the ith sample, and $\widehat{y}_i$ the value predicted by the model. There are other ways of measuring this; another fairly common choice is the *mean absolute error*:

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n}|_i - \widehat{y}_i|.$$

However, there are many procedures in machine learning that work by *gradient descent*, which mean that we adjust the parameters to minimise the error measure in a direction coming from the derivative of the error measure. The MSE has a more natural derivative than the MAE—for linear regression, however, there is no need for gradient descent, since we have a closed formula for the optimum parameters. (In machine learning terminology, one might refer to the coefficients of the linear model as *hyperparameters*; these are the parameters worked out—often by the computer—from the data, rather than those set by the data analyst as part of the model specification.)

6.3 Over- and under-fitting

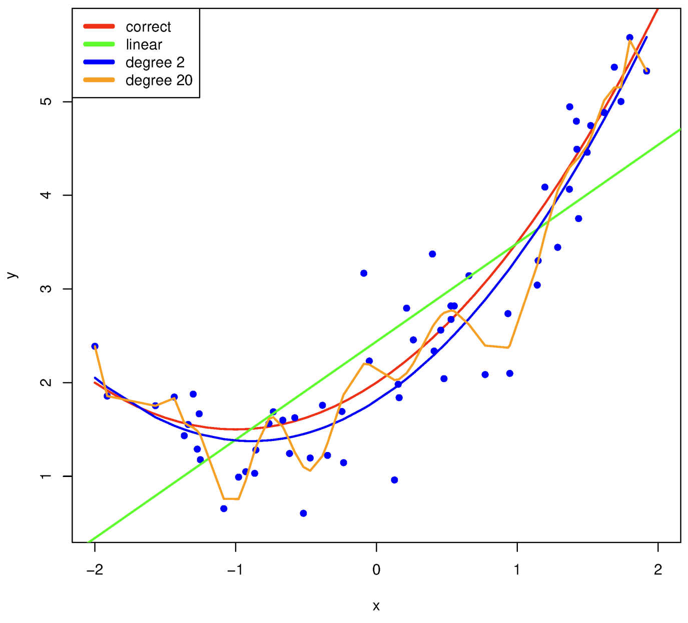
In supervised machine learning, we will typically get a data set consisting of *features*—the variables which are inputs—and the *response* variable, which we want to be able to predict from the features.

We will see many methods ("models") for doing this, some which apply particularly in the classification setting, others which apply particularly in the regression setting, and others which apply in both, possibly with slight modifications.

But the general concepts of underfitting and overfitting are common to all.

Given lots of data with several variables, it is often hard to know whether a linear model is really appropriate. We could extend our models to allow for polynomial terms in some of the variables, but, at least in our example above, this can quickly lead to more terms than observations, and overfitting quickly becomes a problem.

Here's a picture of 60 samples got by adding random noise to the red quadratic function. We then fit a linear model (the green line); clearly there are significant errors here. The blue line is the quadratic function which best approximates the data—not far from the red line, and likely to perform well also for any future data. The orange line is the result of fitting the best degree 20 polynomial to the data points. It is clear that this is picking up the random noise to a significant amount, and this will distort any future predictions.



Let's assume our main goal is to try to make future predictions. There are three possible sources of error:

1. Our model is too simple, and doesn't capture the real structure of the data. Then the model won't work that well on our data (even though this data is used to train the model!), and has little hope of doing well on any future data either.
2. Our model is too complicated, and captures not only the real structure of the data, but also any random noise. By capturing the error as well, similar sets of data with different random errors will behave very differently. Future data is unlikely to be correctly dealt with, since any random errors in future data will be different. In a sense, the model is performing *too* well on the given data set.
3. There is a lot of randomness in the individual observations. So even if we were to guess the "correct" form for a model, there might still be considerable errors owing to the noise in the data.

There's nothing that we can really do about the last of these; guessing the model correctly is clearly the best we can hope for in making predictions, and random noise is going to make predictions for future data uncertain.

However, we might hope to do something about the first two points here. Ideally, we need to choose a model which is simple enough that it reflects the structure of the original data set. It should not be too simple, since that will miss this structure, nor too complicated, since then the contribution from the random errors in the model will be too great.

So there is a trade-off in our choice of model. There are two quantities which define the effectiveness of a model, called the *bias* and the *variance*. The variance describes how predictions vary amongst "similar" sets of data, so models where predictions reflect the noise (overcomplicated models, with overfitting) are likely to have a high variance, whereas simple models will probably not. The bias describes the errors in prediction due to oversimplified models.

It turns out that one can show that the expected mean square error, $\mathbb{E}[y_i - \hat{y}_i]^2$, where y_i is the actual output value, and $\hat{y}_i$ is the prediction of the model, can be decomposed into a contribution from the variance and the bias, together with a contribution measuring the variance of the random noise.

Let's make this a little more formal. We have our feature variables $x_1, \ldots, x_q$ and a response variable y. We are trying to fit a model $y = f(x) + \varepsilon$, where the error term is $\mathrm{N}\big(\varepsilon \,\big|\, 0, \sigma^2\big)$. Unfortunately, f is not given to us—all we can do is to try to find an approximation $\hat{f}$ to f, and this approximation depends on the random noise ε. Let's look at the *expected* square error in $\hat{f}(x)$ for an input x:

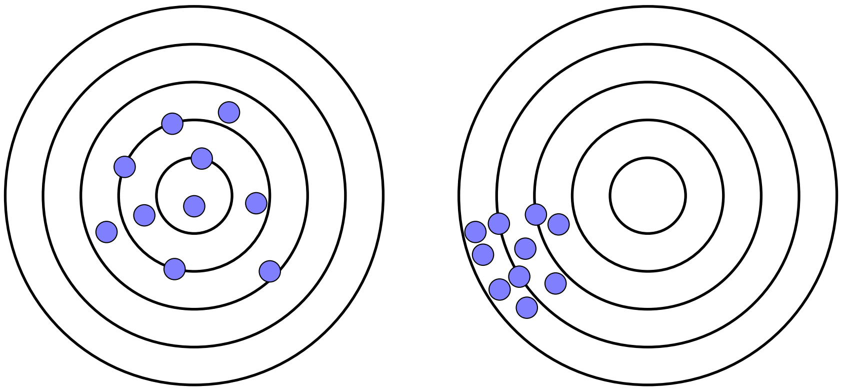
$$\begin{aligned}
\mathbb{E}\Big[(y - \hat{f}(x))^2\Big] &= \mathbb{E}\Big[y^2 - 2y\hat{f}(x) + \hat{f}(x)^2\Big] \\
&= \mathbb{E}\big[y^2\big] + \mathbb{E}\Big[\hat{f}(x)^2\Big] - 2\,\mathbb{E}\Big[y\hat{f}(x)\Big] \\
&= \mathbb{V}[y] + \mathbb{E}[y]^2 + \mathbb{V}[\hat{f}(x)] + \mathbb{E}\Big[\hat{f}(x)\Big]^2 - 2\,\mathbb{E}\Big[y\hat{f}(x)\Big]
\end{aligned}$$

$$
\begin{aligned}
&= \mathbb{V}[y] + f(x)^2 + \mathbb{V}[\hat{f}(x)] + \mathbb{E}\Big[\hat{f}(x)\Big]^2 - 2f(x)\,\mathbb{E}\Big[\hat{f}(x)\Big] \\
&= \mathbb{V}[y] + \mathbb{V}[\hat{f}(x)] + \Big(f(x) - \mathbb{E}\Big[\hat{f}(x)\Big]\Big)^2 \\
&= \sigma^2 + \mathbb{V}[\hat{f}(x)] + \mathrm{bias}[\hat{f}(x)]^2,
\end{aligned}
$$

where $\mathrm{bias}[\hat{f}(x)] = f(x) - \mathbb{E}[\hat{f}(x)]$. If we knew the model exactly, and predicted $\hat{f} = f$, we would get zero bias; if we had large amounts of data, we could ensure that the variance was very small. However, the first is very rare, while the second may or may not hold.

Let's look more at the variance term $\mathbb{V}[\hat{f}(x)]$. This tells us how much the predictive model $\hat{f}$ might change on x as the noise varies randomly. The bias tells how the expected predicted value differs from the real value of the model, so tells us about the deficiency in the predictions $\hat{f}$ compared with f.

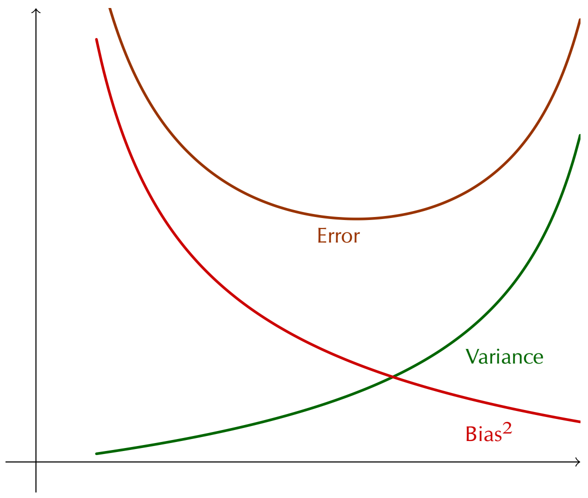
Think of archers firing arrows at the middle of a target. The variance would measure how spread apart the arrows are from each other; the bias would measure whether the arrows were consistently landing some direction from the middle.



In the left-hand target, the bias is very small, since the arrows are fairly well centred, but the variance is quite large; in the right-hand target, the variance is smaller as the arrows are more closely packed, but the bias is large, as the arrows are all off-centre.

By fixing the form of a model, such as a linear model, we increase the possibility of oversimplification, which is likely to raise the bias, but generally the variance will be quite small. The more parameters we allow into our model, adding more and more flexibility, the bias is reduced since we expect the model to reflect the training data more closely, but the variance tends to increase. We will see later models with no prescribed form, where we expect low bias but high variance; these tend to give prediction methods which are less robust to small changes.

Here's a schematic plot of model complexity (along the x-axis) against mean square error (along the y-axis):



There tends to be a trade-off between the bias and the variance. This trade-off is not restricted to linear models, but applies to all choices of model.

6.4 Regularisation

Especially if there are a lot of variables, or not many observations, there is a danger that we may have so many parameters that there is a substantial danger of overfitting.

In the case of linear regression models, we could simply restrict the number of variables used in forming the model. But if we simply ignore some variables, there is a danger that we may be ignoring variables whose inclusion would substantially improve the model. We will select a different method.

Recall that in fitting the linear model, we are trying to minimise some sum of squares error $\sum_{i=1}^{n}(y_i - \widehat{y}_i)^2$, if $\widehat{y}_i$ is the predicted value and y_i is the actual value for the ith response variable. In order to reduce the number of variables, we are going to change this by adding in an extra term, sometimes known as a *cost function*, which penalises models with undesirable properties. If we wish to reduce the number of nonzero coefficients, our penalty term should reflect this.

So we try instead to minimise

$$F = \sum_{i=1}^{n}(y_i - \widehat{y}_i)^2 + \text{cost function},$$

where the cost function is a function of the coefficients $\beta_1, \ldots, \beta_p$ in the linear regression function. If we wish to minimise the number of nonzero coefficients, we might have a cost function

$$\text{cost function}_0 = \text{number of nonzero coefficients.}$$

(This is known as *subset selection*.) We must then go through the derivation of the optimal coefficients in the theory of linear models, but using this new function F, and adapt the results to give a new sort of optimum. It may no longer be possible to adapt the calculations which gave the solution for the coefficients of the linear model in closed form; we may need to use some sort of gradient descent method.

There are other ways to regularise the coefficient selection in the case of linear models: choosing

$$\text{cost function}_1 = \text{sum of absolute values of coefficients}$$

gives *LASSO regression*, and choosing

$$\text{cost function}_2 = \text{sum of squares of values of coefficients}$$

gives *ridge regression*.

If there are more variables than observations, then the technique of linear models will not work (looking at the formula, we invert some $p \times p$ matrix, but if there are n variables, you may be able to persuade yourself that it has rank at most n—and therefore if $n < p$, the matrix will not be invertible). Other methods have similar issues. We will therefore need some method to restrict the number of variables with nonzero coefficients, and there are similar techniques.

6.5 New data

The mean square error is a reasonable measure of how well a model is predicting in regression problems. But this won't tell us how well the model will do for new data, as we have just explained. We could have a model which is completely overfitted, and fits the exact value for every given data point. Then the MSE for the model will be 0, since we'd always have $\hat{y}_i = y_i$. But, in order to do this, we will probably have needed to make the model extremely complicated, as we have had to capture the information in the random noise, as well as the underlying model. This means that the variance of the model will be very high; predictions for new data are very likely to have large errors. So although the model fits the training data perfectly, it will do badly for new test data.

So how can we estimate the performance for unseen data?

It would be nice if our data set were large enough that we could fit our model from a large subset of the data (called the *training set*), and see how well it performs on the remaining part of the data (the *validation set*), perhaps adjusting the model parameters to optimise performance for the validation set. Then this would hopefully give us some idea how it would perform on new data.

Unfortunately, data sets are not always large enough for this to be practical, and we often need to use the full data set to build the best model we can.

Both questions are partly resolved by the following techniques of *resampling*.

Cross-validation

Here, the model (or other method being assessed) is recomputed using some subset of the training set, and testing it on the rest. We could, for example, divide the set into two, and use one half to build the model and see how it performs with the other half. Building a model with half of the data is likely to be less robust, however, and the model may depend substantially on how the set is divided into two. There are a lot of different ways to divide a set of n observations into two, once n is any reasonable size. So a more robust way to do this is to use *leave-one-out cross-validation*, where each observation in the training data is omitted in turn, and the rule is applied to the remaining set. Then we predict the answer for the missing observation.

But with a lot of observations this can be very time-consuming, although using a moderate size random sample rather than the whole set may give a good estimate. As a more practical alternative, we could divide the training data into k approximately equally-sized sets, for some $k < n$, and work out the errors if we apply the rule using any $k - 1$ of the sets as the training data, predicting the result for the kth, and comparing with the actual values. So the leave-one-out method is just a special case of this *k-fold validation*, where $k = n$. This method has other advantages—in the leave-one-out method, there are often outliers whose omission causes the behaviour to change significantly, and this is largely avoided if the groups are big enough. (This sort of thing is another manifestation of the bias-variance trade-off.)

For a linear model, say, we could divide our data into perhaps 5 subsets X_1, X_2, X_3, X_4 and X_5. We would calculate the coefficients in the linear model by fitting it to the data $X_1, \ldots, X_4$, and then apply the model to the input variables in the set X_5, comparing the output from the model with the actual output variable in the data. This gives us an estimate of how the model does on unseen data. To avoid potential issues that X_5 somehow has more outliers, we would then fit the model to X_1, X_2, X_3 and X_5, and compare its predictions with X_4. Then fit the model to X_1, X_2, X_4 and X_5, comparing the results with X_3, and so on, until we have 5 estimates of how well the model performs. Exactly the same method may be used for any model.

Bootstrapping

This is another method for obtaining data sets from the original training data. Rather than splitting the set into subsets, as in cross-validation methods, we form new data sets of the same size as the original by repeatedly random sampling from the original set, *with replacement*—so these new data sets can contain repeated observations. In this way, we can start with a single data set, and obtain from it as many "new" data sets as we like. Then we fit the same kind of model for each, and average them (for regression) or take a majority vote (for classification) to produce an overall classifier which should be more robust, and produce a "wisdom of crowds" classifier. Bootstrapping has another useful feature—we get some test data for free! If there are observations not used in a particular data set (since we allow ourselves to repeat observations from the original set, this is very likely), we can run the trained method on these left-out observations, and see how well the method does. This gives us an "out-of-box" (OOB) error estimation.

We will return to bootstrapping later when we discuss random forests.

7 Supervised learning – classification

In this chapter, we will continue to think about the problems of machine learning, but in the context of classification methods. We will introduce a counterpart to linear regression for classification, known as "logistic" regression (confusingly, it isn't a regression algorithm at all, but a classification one!), but first we will think about a more elementary idea, called "k-nearest neighbours".

One aim of the module is to begin to introduce common problems in contemporary machine learning. Certainly one of the most important is the problem of deciding which of two (or more) known categories to place a new observation. This sort of thing arises in medicine—given a new patient, can the doctor diagnose an illness on the basis of observations? It occurs in computing—can a program automate the decision of whether or not a given email is spam? And you will be able to think of many other situations yourselves.

To fix notation, we suppose that we collect data on "objects" which can be classified into one or other of k known categories ($k \geq 2$). For example, "objects" could be patients, and the $k = 3$ categories are Rheumatoid Arthritis, Psorathic Arthritis and Psoriasis. Or "objects" could be iris flowers, and the $k = 3$ categories could be setosa, versicolor and virginica.

As usual, we measure p variates on each "object" to get a p-dimensional observation $\boldsymbol{x} \in \mathbb{R}^p$.

A *discriminant rule* d can be regarded as a partition of $\mathbb{R}^p$ into k regions $R_1, \ldots, R_k$ (so every point of $\mathbb{R}^p$ lies in precisely one of the regions) such that if $\boldsymbol{x} \in R_j$, then we classify $\boldsymbol{x}$ as in category j. This is a formal way of saying that a discriminant rule d is a way of deciding unambiguously which category an object $\boldsymbol{x}$ belongs to, just on the basis of the measurements $\boldsymbol{x}$.

When you think about it, this is very similar to the linear regression problem we have just considered—the only difference is that rather than predicting a numerical output, we are predicting a *class*.

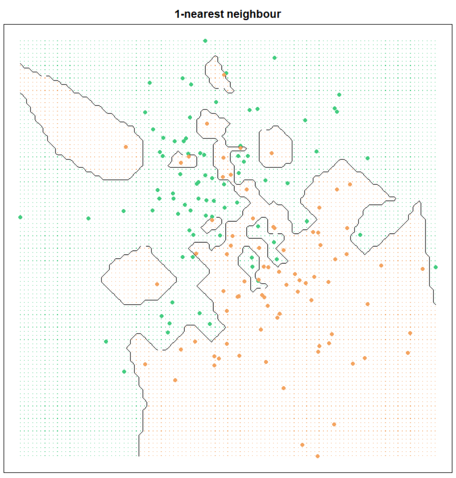
7.1 k-nearest neighbours

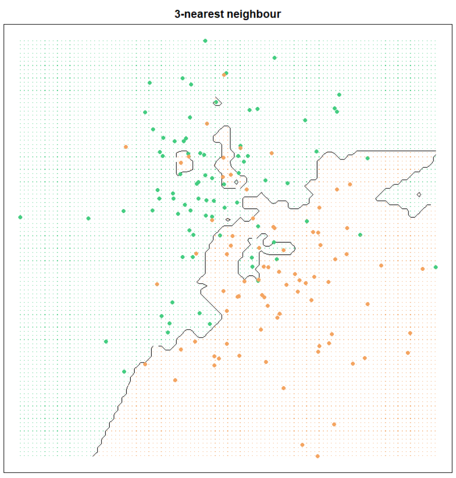
Since we won't discuss this elsewhere, let's introduce the notion of classification by *nearest neighbours*, and illustrate the ideas in practice with this model.

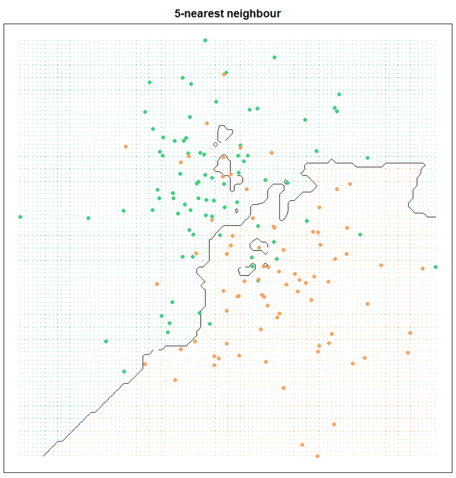
The idea is extremely simple. Let's first consider the case where we have a binary classification problem, so that we have a data set with two classes, say. Then given a new piece of data, we look for the point in the original set which is nearest to it, and allocate the new data to the same class as this point. This is *1-nearest neighbour* classification: notice that *every* point in the original set is correctly classified, which is already an indication that we are picking up all the random noise in the data, and that overfitting is most likely to be taking place.

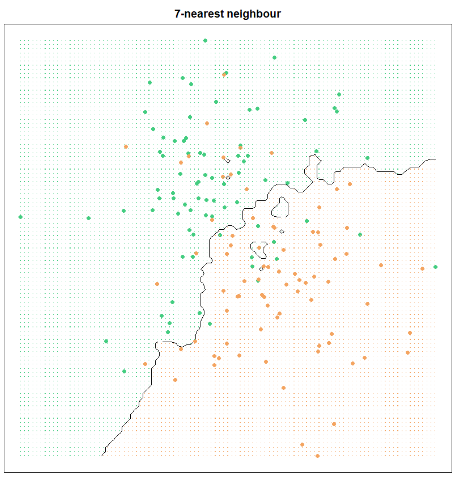
There's a danger, though, that the new data is right by an outlier, so is picking up the random noise from the outlier. It may be preferable to select the three closest neighbours, and allocate the new data to the class which most of the three belong to. This gives *3-nearest neighbours* classification.

We can clearly extend this to k-nearest neighbours classification for any odd number k. For larger k, it should be clear that we are less and less likely to pick up random noise from outliers, but we should be careful not to make k too large, as then we are taking so many points into account that we might be missing some local behaviour. So if k is too small, we are likely to be overfitting the data, whereas if k is too big, we will be underfitting. Here are some images for $k = 1, 3, 5, 7$, 15 and 25:



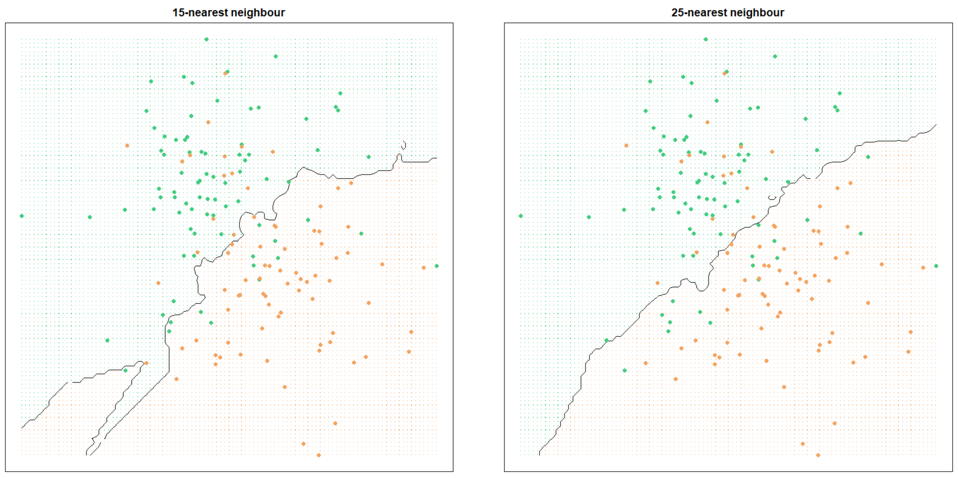






We have also plotted the *decision boundary*, the line dividing those points predicted to go into one class from those predicted to go into the other. It is striking how it changes from being very wiggly for small k to being nearly linear for large k.

One can also use nearest neighbour methods for regression; one predicts a response for a new piece of data which is an average of the responses for the k nearest neighbours. However, we won't discuss nearest neighbour methods any more; they work well for fairly small data sets and for fairly small dimensions. But if the data sets are too big, it takes the algorithm a long time to work out all



the distances involved, although there are ways around this. More seriously, our intuition starts to break down when working with lots of variables (this is the so-called "curse of dimensionality"), and points quickly tend to have very few near neighbours in many dimensions.

7.2 Logistic regression

We would like to consider similar algorithms to linear regression for classification problems also, and logistic regression is one way to do this.

In fact, it is not unreasonable to use linear regression for classification problems also, as long as there are two classes. We can form the linear model with the usual regression methods. Then, with new data, we can compute the value of the linear function, and assign it to the first class if it is positive, and to the second if it is negative.

This approach is less successful with more than two classes. Indeed, if there are three classes, it is unlikely that there is a natural ordering of the classes—the classes may consist of three variants of a disease, for example. Although one can use logistic regression with more than two classes, in practice, one tends to use the methods of discriminant analysis which we will consider in the next chapter. For this reason, we will restrict to the binary situation here.

To explain the ideas, we suppose that we have two categories A and B, and explanatory variables $x_1, \ldots, x_p$. We generalise the idea of a linear model, and assign a new observation to category A if some linear combination $\beta_1 x_1 + \cdots + \beta_p x_p$ is less than a certain threshold, and to category B if it is above it. In other words, there is a function $\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$—a new observation will be classified as category A if it is negative, and category B if it is positive.

There are various nice reasons why we might not want the output just to be simply a binary decision between the two classes:

1. We might want to have some quantification of the probability that the classification has been successful, so values of the function which are only slightly above 0 might give a probability of just over 0.5 of being in category A.

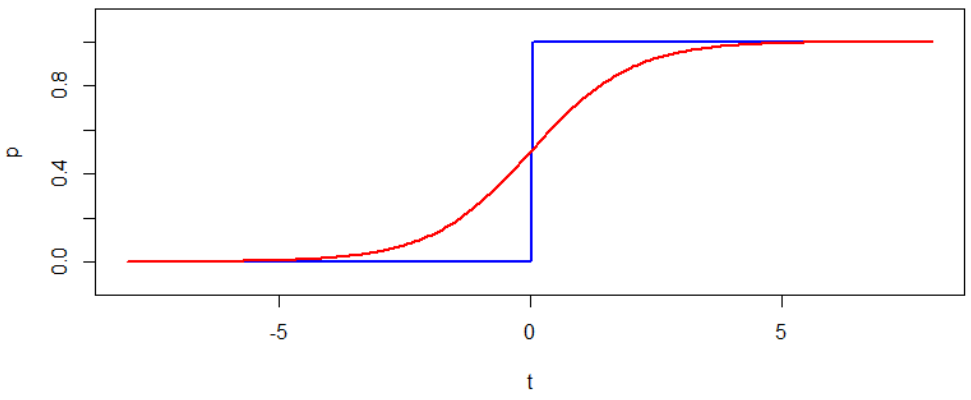
2. As we will see, it will be useful for our iterative procedure for the function to be differentiable. This is because we will use a gradient ascent approach, where we adjust our parameter choices according to the direction of steepest ascent, which we work out using differentiation.

So we look for a function p which has the following properties:

- $p(t)$ should be monotonically increasing, so larger values of the linear model correspond to larger probabilities;
- $p(t)$ should be differentiable;
- $t = 0$ should have $p(t) = 0.5$;
- As $t \to -\infty$, $p(t) \to 0$ and as $t \to \infty$, $p(t) \to 1$.

There will be many functions like this, but perhaps the simplest, and the one very often used in practice, is the *sigmoid* function

$$p(t) = \frac{e^t}{1+e^t}.$$



The function is closely related to the hyperbolic tangent. We'll meet it again when we discuss neural networks.

So if the linear model gives a value of $t_i = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$ for the ith observation, we assign a "probability" of $p(t_i)$ of belonging to category A, and then a probability of $1 - p(t_i) = \frac{1}{1+e^{t_i}}$ of being in category B. It is easy to invert this function: we get $t_i = \log\left(\frac{p(t_i)}{1-p(t_i)}\right)$.

So we model the probability that an object with values of $x_1, \ldots, x_p$ belongs to category A as a *logistic function*

$$P(\text{belongs to A}) = \frac{\exp(\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p)}{1 + \exp(\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p)},$$

and then estimate the unknown parameters β_i from training data on objects with known classifications. New observations would be classified as type A if the estimated probability of belonging to A is greater than 0.5, otherwise they would be classified as type B. The parameters are estimated not using mean square error methods, but by maximum likelihood methods. (Indeed, you should think about how one might try to use mean square error methods, and persuade yourself that it no longer makes real sense.)

So how can we measure the success of the linear model? We can look at all those samples which belong to category A, and see whether the model predicts it correctly, and similarly for those in category B. For the category A samples, we predict that they will be in category A with probability $p(t_i)$, so we regard this as the probability that we have correctly predicted this sample. Similarly, for a sample in category B, the model will predict it correctly if $p(t_i) < 0.5$, and we regard $1 - p(t_i)$ as the probability that it is correctly predicted.

So we aim to choose the parameters $\beta_0, \ldots, \beta_p$ to maximise the likelihood that our predictions are right, i.e., to maximise the function

$$\mathrm{L}\big(\beta_0, \ldots, \beta_p \,;\, \boldsymbol{x}\big) \propto \prod_{\text{category A}} p(t_i) \prod_{\text{category B}} (1 - p(t_i)).$$

In practice, it is easier to work with the logarithm of this:

$$\ell = \log \mathrm{L}\big(\beta_0, \ldots, \beta_p \,;\, \boldsymbol{x}\big) = \sum_{\text{category A}} \log p(t_i) + \sum_{\text{category B}} \log(1 - p(t_i)).$$

We have a formula above for $p(t_i)$ in terms of the parameters, so we can work out $\frac{\partial}{\partial \beta_j} \log p(t_i)$ and hence $\frac{\partial \ell}{\partial \beta_j}$. It is now easy to use methods of gradient descent (or rather, gradient *ascent*, since we want to maximise the log likelihood) to find good choices of the parameters. In fact, this log likelihood function is concave, so the local maximum is actually a global maximum.

The technique is widely used and is very effective; it is known as *logistic regression* or *logistic discrimination*. It can readily handle cases where the $\boldsymbol{x}_i$ are a mixture of continuous and binary variables. If there is an explanatory variable which is categorical with $k > 2$ levels, then it should be replaced by $k - 1$ dummy binary variables.

Of course, measuring model performance for classification problems can no longer be done with sum of squares of differences in actual and predicted values. Instead we can simply count the number of data points whose classes are predicted incorrectly. But we still have the notions of cross-validation and bootstrapping for measuring performance in classification problems as well.

7.3 Multiclass extensions

Logistic regression is naturally a "binary" classification algorithm, so allocates new observations to one or other of two classes. There are a couple of obvious ways to extend binary classification methods to more classes, both of which are used for logistic regression and other binary classification problems (e.g., support vector machines, which we will meet later):

one-versus-one Here we work out all the boundaries between every possible pair of classes. That is, with k classes, we run our method $\binom{k}{2} = k(k-1)/2$ times, once for each pair of classes, with just the observations from that pair each time. This gives us a boundary between each pair of classes, and then we can look at the regions in space where each class is preferred.

one-versus-all or *one-versus-rest*. Here we look at the boundaries between each class and the collection of all the other classes. With k classes, we run the method k times, with the ith run comparing the ith class with the collection of all observations apart from the ith. We choose the class for a new piece of test data to be the one with the largest (positive) distance from the boundary.

7.4 Model performance of classifiers

We can work out the model performance for binary classifiers on existing data by comparing the actual classes with the predicted ones. We have the following notions, where we work with the two classes “positive” and “negative” (as might be the case when testing a hypothesis):

True positives these are the observations correctly predicted to be positive. Thus their actual class and their predicted class are both positive.

False positives these are the observations which are actually negative, but whose predicted class is positive. These are also known as *Type I* errors.

True negatives these are the observations correctly predicted to be negative, so both their actual and predicted classes are negative.

False negatives these are the observations which are actually positive, but whose predicted class is negative. These are also known as *Type II* errors.

We can summarise the model performance with a *confusion matrix*: $\begin{pmatrix} TP & FP \\ FN & TN \end{pmatrix}$, where TP denotes the number of true positives etc.. So the top row consists of positive predictions, and the left-hand column of actual positive observations.

One can do the same for multiclass problems: with c classes, one would get a $c \times c$-matrix.

But it’s nice to have a single number to quantify how well the classifier is performing!

There are various ways to measure the success of a binary classifier. The most natural, perhaps, is *accuracy*, which measures the proportion of all observations which are correctly predicted. That is:

$$\text{accuracy} = \frac{TP + TN}{TP + FP + TN + FN},$$

and this also works well in the multiclass situation, where accuracy would be defined as the proportion of observations correctly defined.

It might seem that this is such a natural measure that it makes no sense to use any other. Let’s explain that accuracy is not always very useful.

Let's consider a medical diagnosis for a comparatively rare disease, perhaps. Probably rather few patients have the condition, and few will be predicted to have the condition. We could predict that no patient has the disease, and then we will have a high accuracy rate, dominated by the true negatives. This isn't very useful!

The moral is that accuracy is a less good measure for these *unbalanced* classification problems, where the proportion of observations in one class is much higher than the proportion of observations in the other. We will use "negative" and "positive" for the majority and minority classes, as would be typical in a medical diagnosis problem. One can certainly imagine minority classes being smaller still than the 1% given above; some medical conditions are extremely rare indeed. As a starting point, it makes sense to regard classification errors with the minority class as more important; one practical issue is that training models focusing on the minority class is awkward, since there are by definition fewer observations involved in training the model.

So we might want to eliminate true negatives from consideration, and only consider whether the diagnostic procedure is working well on those patients with the disease. Or perhaps we want to ensure that there are few incorrect diagnoses, so we would look at those predicted to be negative, and work out the proportion of true negatives. This leads to further measures, or metrics, for model performance:

$$\text{sensitivity} = \frac{TP}{TP + FN},$$

the proportion of those who are positive who are predicted to be (this measures how sensitive the test is in detecting positives, and is also known as *recall*—see below);

$$\text{specificity} = \frac{TN}{TN + FP},$$

the proportion of those who are negative who are correctly predicted.

So specificity and sensitivity are complementary in that they compute the same measure, but for the two classes.

Sometimes these are combined to give further metrics. A measure that combines specificity and sensitivity to give a measure that treats the two classes equally is their geometric mean, often abbreviated as *G-mean*:

$$G\text{-mean} = \sqrt{\text{sensitivity} \times \text{specificity}}.$$

It is easy to imagine how to extend this to weight one measure more than the other, by taking something like

$$\sqrt[\alpha+\beta]{\text{sensitivity}^{\alpha} \times \text{specificity}^{\beta}}.$$

There is another pair of commonly used measures:

$$\text{precision} = \frac{TP}{TP + FP},$$

the proportion of examples assigned to the positive class that are correctly predicted,

$$\text{recall} = \frac{TP}{TP + FN},$$

the proportion of positive examples correctly predicted to be positive.

Notice that recall has exactly the same definition as sensitivity above; "recall" tends to be used in conjunction with precision, and "sensitivity" with specificity.

The *F-score* combines the precision and recall measures, using the harmonic mean.

Recall that the harmonic mean of a and b is the number h such that

$$\frac{1}{h} = \frac{1/a + 1/b}{2},$$

i.e., the reciprocal of h is the arithmetic mean of the reciprocals of a and b. Rearranging, we get

$$h = \frac{2ab}{a+b}.$$

We define

$$F\text{-score} = \frac{2\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}.$$

7.5 Threshold measures

Many decision procedures, like logistic regression, give more than a simple classification; they provide us with an estimate of a "probability" of the classification.

Ideally, the classifier should separate the classes well, so that positive predictions have probabilities near 1 and negative predictions have probabilities near 0. Then we would hope that varying the threshold between, say, 0.3 and 0.7 would not change the predictions much; we might use this variation of the threshold as a measure of confidence of the classifier.

It may seem natural to predict positive or negative in the setting above if the predicted probability of being positive is greater than, or less than, 0.5.

There are other reasons why we might consider varying the threshold. There may be no reason to expect that the training set and test set should have the same proportion of positive and negative classes, and we might want to adjust the threshold so that the proportion of positive and negative predictions from the test set is under our control. Or we may have a medical example, where we might want a probability of at least 0.95 that a patient doesn't have a disease before deciding not to progress with further tests.

We might choose to predict that an observation is positive if the output of logistic regression was greater than 0.4, for example. So every threshold has an associated list of predictions, and an associated list of true positives, true negatives, false positives and false negatives. At one extreme, with a threshold of 0, everything would be predicted to be positive; this would mean that the sensitivity would be 1 and the specificity would be 0. Conversely, with a threshold of 1, everything would be predicted to be negative, and then the sensitivity would be 0 and the specificity would be 1. You should check the formulas above to see that observations coming from a random guess (uniformly taken from $[0, 1]$) at a threshold of p would have sensitivity $1 - p$ and specificity p.

Other classifiers exhibiting no skill, such as always predicting the majority class, will have the same property.

Notice that

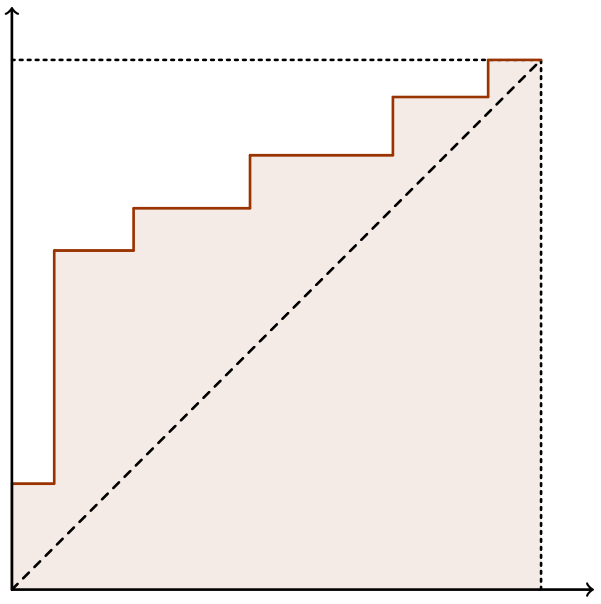
$$1 - \text{specificity} = \frac{FP}{TN + FP},$$

which we can think of as the *false positive rate*, while

$$\text{sensitivity} = \frac{TP}{TP + FN}$$

is the *true positive rate*.

Plotting the false positive rate ($1 - \text{specificity}$) on the x-axis and the true positive rate (sensitivity) on the y-axis for every possible threshold, gives a curve known as the *receiver operating characteristic (ROC)* curve. For a random guess, as above, we would get a diagonal line between $(0, 1)$ and $(1, 0)$ (the line $y = 1 - x$). We would hope that most thresholds give correct predictions, so that the true graph will be higher. In practice, we usually reflect the graph so that the diagonal line goes from $(0, 0)$ to $(1, 1)$ by plotting sensitivity against $1 - \text{specificity}$.



The *area under the curve* (AUC) is a good measure of how well the model is performing; we want every positive to have a probability near 1, and every negative to have a probability near 0 – if this were exactly true, the ROC curve would rise vertically from the origin up to $(0, 1)$, and then move horizontally from there to $(1, 1)$, and the AUC would be 1. A random guess would have AUC equal to 0.5; the higher the AUC, the better the model predicts correctly.

We have used sensitivity and specificity here, but sometimes precision and recall are used. So the x-axis will use

$$\text{recall} = \frac{TP}{TP + FN}$$

while the y-axis will use

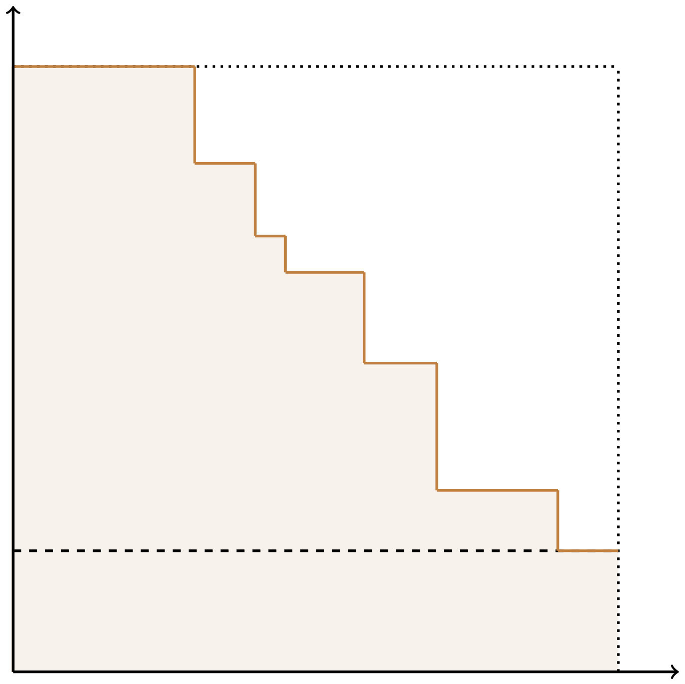
$$\text{precision} = \frac{TP}{TP + FP}.$$

With a threshold of 0, so that every observation is classified as positive, we get $FN = 0$, and $\text{recall} = 1$ and precision equal to the proportion of positive samples in the data set.

With a threshold of 1, we see $\text{recall} = 0$.

A classifier with no skill will have precision always equal to the proportion of positive samples in the data set, so the precision-recall curve would be a horizontal line at height equal to this proportion.

A perfect classifier will have no false predictions, so $\text{precision} = 1$ for all thresholds; a good classifier will have curve above the horizontal line:



and we can use the area under the precision-recall curve (PR-AUC) as another measure of the performance of the classifier. This PR-AUC will be 1 for a classifier that measures the accuracy perfectly at all thresholds.

8 Discriminant Analysis

8.1 Introduction

Linear discriminant analysis (LDA) is a classification method derived in a similar way to PCA.

We recall our notation from the previous chapter: we suppose that we collect data on "objects" which can be classified into one or other of k known categories ($k \geq 2$). As usual, we measure p variables on each "object" to get a p-dimensional observation $\boldsymbol{x} \in \mathbb{R}^p$.

Given this data, we would like to find a way to classify new data: given values of the input variables, we want to predict the class of the new observation.

In fact, we will come up with an alternative derivation later in the chapter, but for now, we give a derivation that looks like what we used for PCA.

While PCA finds the direction in which the data looks most spread out, LDA gives the direction in which the groups are as separated as possible.

If there are k different groups, we will find $k-1$ separate linear combinations (which we can regard as projections of the data set to 1 dimension), which partition the *between groups variance* into decreasing order, similarly to the way that PCA partitions the *total variance* into decreasing order.

- Like PCA, this will come down to an eigenvector calculation.
- The method requires more observations than variables (i.e., $n > p$), since it requires the invertibility of a $p \times p$ matrix whose rank is at most n.
- Boundaries between classes will be linear.

Suppose that there are k groups, and that the ith group has n_i observations of dimension p. Write $n = \sum n_i$.

Write X_i for the data matrix for the ith group (so X_i is a $n_i \times p$ matrix), $\bar{\boldsymbol{x}}$ for the p-vector which is the overall mean of the n observations, and $\bar{\boldsymbol{x}}_i$ for the p-vector which is the mean of the ith group.

Define

$$S_i = \frac{1}{n_i - 1}(X_i - \bar{\boldsymbol{x}}_i)(X_i - \bar{\boldsymbol{x}}_i)' = \frac{1}{n_i - 1}\sum_{j=1}^{n_i}(x_j^{(i)} - \bar{\boldsymbol{x}}_i)(x_j^{(i)} - \bar{\boldsymbol{x}}_i)',$$

the group i variance matrix.

Write

$$W = \frac{1}{n-k}\sum_{i=1}^{k}(n_i - 1)S_i.$$

This is the *within groups* variance matrix, and gives the contribution to the total variance from each group—it only involves the spread of the data within each individual group.

Also define

$$B = \frac{1}{k-1}\sum_{i=1}^{k} n_i(\bar{\boldsymbol{x}}_i - \bar{\boldsymbol{x}})(\bar{\boldsymbol{x}}_i - \bar{\boldsymbol{x}})',$$

the *between groups* variance matrix—it measures how the class means are separated, and therefore how far apart the groups are, but nothing about the spread within each group. Both are symmetric $p \times p$-matrices.

If S denotes the overall variance matrix for the whole data set, it turns out that

$$(n-1)S = (n-k)W + (k-1)B.$$

8.2 Linear discriminant analysis

W and B are analogous to the within and between groups mean squares in a univariate one-way analysis of variance, and if we set $p = 1$ then the usual formulae for these are retrieved. We want to compare W and B; since these are matrices, we can't take a quotient, but we will give a method below where the product $W^{-1}B$ comes out naturally, and when $p = 1$, this is indeed just the quotient. We should regard $W^{-1}B$ as a measure of comparison between W and B; we need some measure of how "large" $W^{-1}B$ is, and there are many possibilities—determinant, largest eigenvalue, sum of eigenvalues (trace), etc., and none really stands out as a natural choice.

If we projected all the data into one dimension, then we could perform a 1-way analysis of variance and compare the between groups mean square with the within groups mean square: we would calculate the F-statistic, which is the ratio of between to within groups mean squares. If there are large differences between the groups, then the between group mean square will be relatively large and so the F-statistic will be large.

As we take different projections, this ratio will vary; sometimes it may be very small, when the differences between the groups are hidden by the projection, and at other times it may be very significant, when the projection reveals differences between the groups.

So we project all the p-dimensional data onto vector $\boldsymbol{a}$ (a column p-vector), and the projected data matrix for the ith group becomes $\boldsymbol{a}'X_i$. Then the sample variance of the projected data of the ith group is $\boldsymbol{a}'S_i\boldsymbol{a}$ (note that this is a scalar!).

Similarly, the within group variance of the projected data is $\boldsymbol{a}'W\boldsymbol{a}$, and the between group variance becomes $\boldsymbol{a}'B\boldsymbol{a}$.

To highlight the distinction between the groups, we want to choose $\boldsymbol{a}$ to maximise $F = \frac{\boldsymbol{a}'B\boldsymbol{a}}{\boldsymbol{a}'W\boldsymbol{a}}$ (i.e., just the usual F-ratio in a classical analysis of variance).

So the problem is to maximise $F = \frac{\boldsymbol{a}'B\boldsymbol{a}}{\boldsymbol{a}'W\boldsymbol{a}}$ with respect to $\boldsymbol{a}$. Note that F is a ratio of quadratic forms in $\boldsymbol{a}$, so if $\boldsymbol{a}$ is multiplied by any scalar, then the value of F is unaltered.

This means that an equivalent problem is: "maximise $\boldsymbol{a}'B\boldsymbol{a}$ subject to $\boldsymbol{a}'W\boldsymbol{a} = 1$".

This allows us to convert the original problem to a constrained maximisation problem which we solve with a Lagrange multiplier.

Introduce a Lagrange multiplier λ, and let $\Omega = \boldsymbol{a}'B\boldsymbol{a} - \lambda(\boldsymbol{a}'W\boldsymbol{a} - 1)$. Then $\frac{\partial\Omega}{\partial\boldsymbol{a}} = 2B\boldsymbol{a} - 2\lambda W\boldsymbol{a} = 0$, i.e., $W^{-1}B\boldsymbol{a} - \lambda\boldsymbol{a} = 0$, i.e., $\boldsymbol{a}$ is an eigenvector of $W^{-1}B$ corresponding to the eigenvalue λ.

Notice that this is the point where we use the invertibility of the $p \times p$ matrix W, which requires that it has sufficiently large rank.

To see which eigenvector is needed to maximise F, note that $B\boldsymbol{a} = \lambda W\boldsymbol{a}$ implies that $\boldsymbol{a}'B\boldsymbol{a} = \lambda\boldsymbol{a}'W\boldsymbol{a}$, so $\boldsymbol{a}'B\boldsymbol{a} = \lambda$, and we take λ as the largest eigenvalue of $W^{-1}B$ and $\boldsymbol{a}$ as the corresponding eigenvector.

The first eigenvector $\boldsymbol{a}_1$ is the *first discriminant coordinate*.

By analogy with PCA, subsequent eigenvectors $\boldsymbol{a}_2, \boldsymbol{a}_3, \ldots$, are the second, third,..., *discriminant coordinates* (or *crimcoords*), and data plotted on these axes.

Note that:

- unlike PCA, later eigenvectors don't come from a Lagrange multiplier calculation.
- although B and W are symmetric, $W^{-1}B$ is not, so the eigenvectors are not generally orthogonal, so when we plot data on these axes, there is generally distortion.
- B has rank at most $\min(k-1,\ p)$, so $W^{-1}B$ does also, and so there are at most $k-1$ eigenvectors with nonzero eigenvalues.

The function $h(x) = \boldsymbol{a}'x = \boldsymbol{a}_1'x$ is known as *Fisher's linear discriminant function*. This choice of $\boldsymbol{a}_1$ is the direction that makes the separation between the groups most clear.

The functions $\boldsymbol{a}_i'x$ are called the *discriminant coordinates*, and the space they span is called the *discriminant space*.

Notice that the decision boundaries are determined by the linear combinations $\boldsymbol{a}_i'x$, so are linear in the entries of x, and so is a hyperplane—a line in 2 dimensions, or a plane in 3 dimensions, etc.

8.3 Remarks

- Note that the eigenvectors of $W^{-1}B$ are not orthogonal—although both W and B are symmetric, $W^{-1}B$ need not be. Although we don't have $\boldsymbol{a}_i'\boldsymbol{a}_j = 0$, we do have $\boldsymbol{a}_i'W\boldsymbol{a}_j = \boldsymbol{a}_i'B\boldsymbol{a}_j' = 0$, so we get a sort of twisted orthogonality, and if the variables are independent with the same variance σ^2, then $W = \sigma^2 \mathrm{I}_p$, and the condition really is an genuine orthogonality condition.
- Although the crimcoords are not orthogonal, it is traditional to plot them on orthogonal axes; this distorts the original space of points, but in a sensible way. Transforming to discriminant

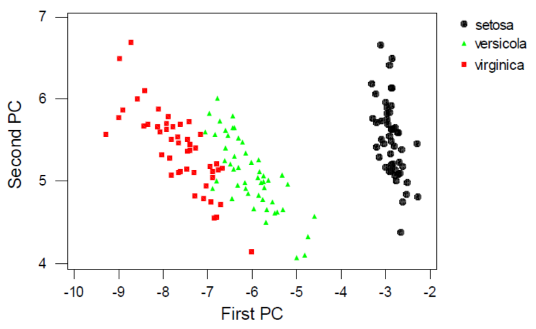
space is a good way to display relationships and distinctions between groups, and we hope that the groups will be better discriminated.

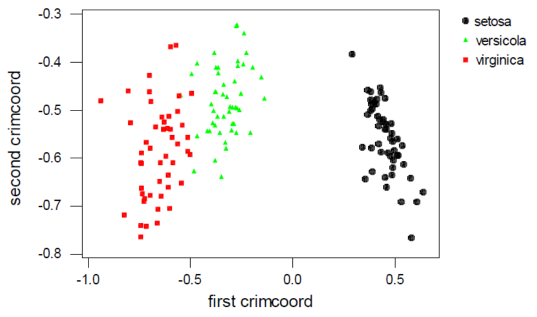
- In practice, one chooses a suitable number t of co-ordinates (just like PCA), and views the data with these crimcoords. Sometimes these functions are also referred to as *canonical variates*.
- We can choose a suitable t by examination of the sequence $\lambda_1, \lambda_2, \ldots$ using a scree plot (by analogy with PCA), but the non-orthogonality of the eigenvectors means that the amount of discrimination is not partitioned into separate bits, unlike principal components and classical scaling.
- One could imagine interpreting successive eigenvalues (and cumulative proportions etc.) as "successive amounts of discrimination achieved" by each axis, but this should be treated with caution:
  - Firstly, only $\boldsymbol{a}_1$ and λ_1 came out of the calculation, and we work with the others only because it might be suggested by PCA (and that they sometimes give nice results in practice!).
  - Secondly, it might be more appropriate to normalise each variable to have variance 1 at the start, otherwise the variables with larger variance will again dominate the calculations.

8.4 Example (iris data)

Let's have a quick look at how PCA and LDA differ for the iris data. I'll use plots from R here, but you might like to replicate this in `Python`.

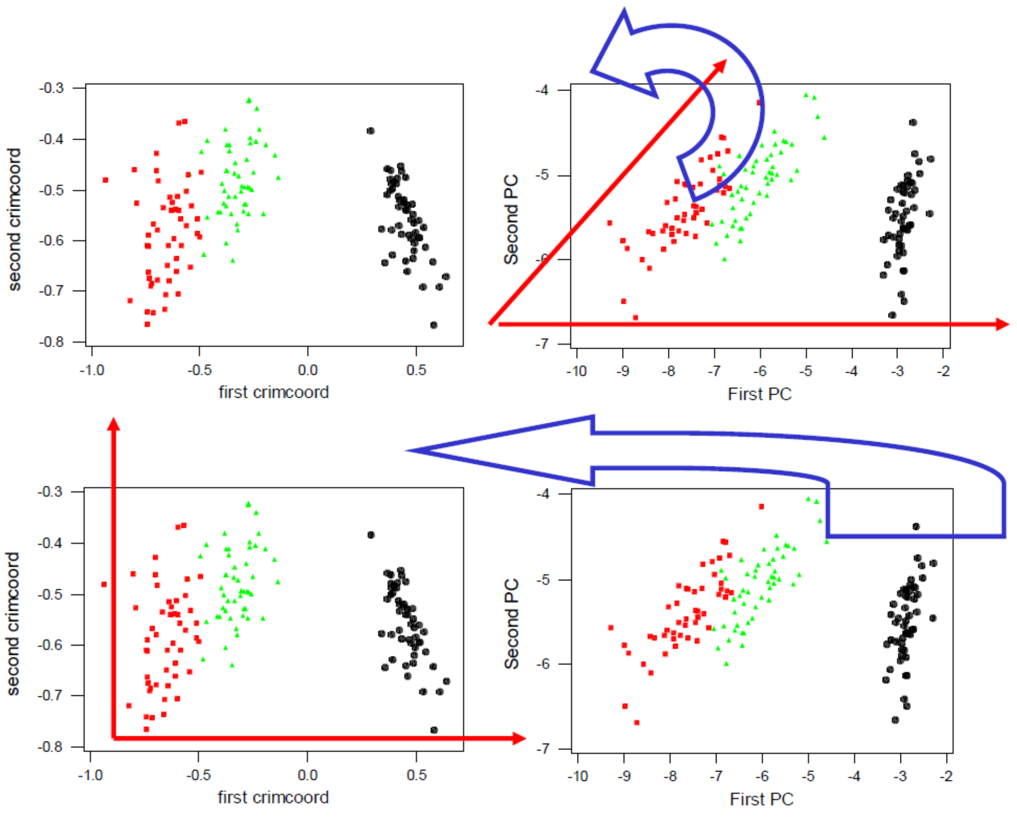
Let's plot the iris data on principal components:





Note the distortion of the data on the crimcoords—it is as if the true axes have been stretched from an acute angle: and made into a right angle by pulling out one axis to stretch the data space:

On the Lab Sheet, there is a nice example of a set with 4 groups; you should find that PCA separates out one of the main variables, but LDA is more successful at separating all of the groups.



8.5 Special case: $k = 2$

Let's restrict to the case where $k = 2$. In this case, B has rank 1, and so

$$B = \frac{n_1 n_2}{n}(\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_2)(\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_2)' = \frac{n_1 n_2}{n} d d',$$

say. Then $W^{-1}B$ has only one non-zero eigenvalue (as it has rank 1) which is $\frac{n_1 n_2}{n} d' W^{-1} d$, and the corresponding eigenvector is $W^{-1}d$.

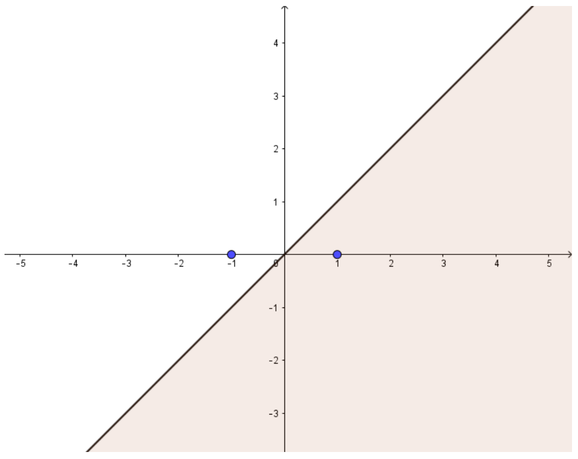
The rule is to allocate to class 1 if

$$(\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_2)' W^{-1}(\boldsymbol{x} - (\bar{\boldsymbol{x}}_1 + \bar{\boldsymbol{x}}_2)/2) > 0.$$

In fact, "discriminant analysis" also refers to another method, which we shall look at next. For the binary classification problem, we will see that we get exactly the same result as the rule we have just found.

8.6 The decision boundary between two Gaussian distributions

As it will influence later discussions, let's consider the following problem. We have two multivariate Gaussian distributions, $\boldsymbol{x}_1 \sim \mathrm{N}_p(\boldsymbol{x}_1 \mid \boldsymbol{\mu}_1, \Sigma_1)$ and $\boldsymbol{x}_1 \sim \mathrm{N}_p(\boldsymbol{x}_2 \mid \boldsymbol{\mu}_2, \Sigma_2)$. We see observations which are generated *either* from population 1 *or* from population 2. The question is simply: Given



an observation $x \in \mathbb{R}^p$, we want to know when we should guess that x comes from population 1 or from population 2.

In other words, we want to know the region where x_1 has a larger pdf than x_2 (and if x lies in this region, we guess x comes from population 1), and where x_2 has a larger pdf than x_1 (when we'll guess x comes from population 2).

The region where the pdfs of x_1 and x_2 are equal is going to divide the space of all possible observations into two parts, and this is called the *decision boundary*.

It will be instructive to work out the decision boundaries for a couple of sample cases.

For simplicity, let's assume $p = 2$, and we'll take $\mu_1 = (-1, 0)$ and $\mu_2 = (1, 0)$. We suppose first that $\Sigma_1 = \Sigma_2 = \Sigma = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}$. Then $\Sigma^{-1} = \begin{pmatrix} 1 & -1 \\ -1 & 2 \end{pmatrix}$, and the two pdfs are equal for points x satisfying

$$\frac{|\Sigma|^{-1/2}}{(2\pi)^{p/2}} \exp\left[-\frac{1}{2}(x-\mu_1)'\Sigma^{-1}(x-\mu_1)\right] = \frac{|\Sigma|^{-1/2}}{(2\pi)^{p/2}} \exp\left[-\frac{1}{2}(x-\mu_2)'\Sigma^{-1}(x-\mu_2)\right].$$

So we simply need

$$(x-\mu_1)'\Sigma^{-1}(x-\mu_1) = (x-\mu_2)'\Sigma^{-1}(x-\mu_2).$$

If $x = (x_1, x_2)$, we simply need

$$(x_1+1\ x_2)\begin{pmatrix} 1 & -1 \\ -1 & 2 \end{pmatrix}\begin{pmatrix} x_1+1 \\ x_2 \end{pmatrix} = (x_1-1\ x_2)\begin{pmatrix} 1 & -1 \\ -1 & 2 \end{pmatrix}\begin{pmatrix} x_1-1 \\ x_2 \end{pmatrix}.$$

Expanding, we get

$$x_1^2 + 2x_1 + 1 - 2x_1x_2 - 2x_2 + 2x_2^2 = x_1^2 - 2x_1 + 1 - 2x_1x_2 + 2x_2 + 2x_2^2,$$

or $x_1 = x_2$. So the decision boundary is a straight line, consisting of all points (x_1, x_2) where the two coordinates are equal, i.e., all multiples of $(1, 1)$.

Try this yourself with any two Gaussian distributions of the same dimension, with equal variance matrices. You will always find that the decision boundary is a straight line (or a plane in three dimensions, etc.).

But now let's try a situation with different variance matrices. We'll keep $\boldsymbol{\mu}_1 = (-1,\ 0)$ and $\boldsymbol{\mu}_2 = (1,\ 0)$. Let's take $\Sigma_1 = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}$, and $\Sigma_2 = I$, the identity. (Both have the same determinant, so the parts before the exponential terms agree.) Note that $\Sigma_1^{-1} = \begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix}$.

In the same way as before, we need to solve

$$(x_1+1\ x_2)\begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix}\begin{pmatrix} x_1+1 \\ x_2 \end{pmatrix} = (x_1-1\ x_2)I\begin{pmatrix} x_1-1 \\ x_2 \end{pmatrix}.$$

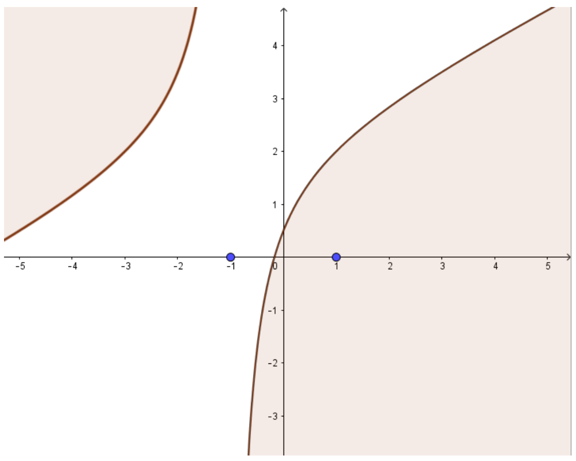
This expands to

$$2x_1^2+4x_1+2-2x_1x_2-2x_2+x_2^2 = x_1^2-2x_1+1+x_2^2,$$

or

$$x_1^2+6x_1+1-2x_1x_2-2x_2 = 0,$$

which is the equation of a hyperbola.



In general, if the two variance matrices are different, then the decision boundary is quadratic (i.e., a circle, ellipse, parabola or hyperbola, if $p = 2$), and is never something linear.

Example 8.1 We consider the case of two univariate normal populations. Here $p = 1$, $k = 2$ and $x \sim \mathrm{N}(x_i \mid \mu_i,\ \sigma_i^2)$ if x is in category i. We allocate x to category 1 if $f_1(x) > f_2(x)$, i.e., if

$$\frac{\sigma_2}{\sigma_1}\exp\left[-\frac{1}{2}\left(\frac{x-\mu_1}{\sigma_1}\right)^2+\frac{1}{2}\left(\frac{x-\mu_2}{\sigma_2}\right)^2\right] > 1,$$

i.e., if

$$Q(x) = x^2\left(\frac{1}{\sigma_1^2}-\frac{1}{\sigma_2^2}\right)-2x\left(\frac{\mu_1}{\sigma_1^2}-\frac{\mu_2}{\sigma_2^2}\right)+\left(\left(\frac{\mu_1^2}{\sigma_1^2}-\frac{\mu_2^2}{\sigma_2^2}\right)-2\log\left(\frac{\sigma_2}{\sigma_1}\right)\right) > 0.$$

Suppose that $\sigma_1 > \sigma_2$, and the coefficient of x^2 in $Q(x)$ is negative. Then $Q(x) < 0$ if x is sufficiently small or sufficiently big.

If $\sigma_1 = \sigma_2$, then $\log(\sigma_2/\sigma_1) = 0$, so the rule becomes allocate x to category 1 if $|x - \mu_1| < |x - \mu_2|$.

Here's an example in more dimensions:

Example 8.2 p dimensions, k normal populations, means μ_i and common variance Σ. Then

$$f_i(\boldsymbol{x}) = (2\pi)^{-p/2}|\Sigma|^{-1/2} \exp\big[-(\boldsymbol{x} - \boldsymbol{\mu}_i)'\Sigma^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_i)/2\big],$$

which is maximised when $(\boldsymbol{x} - \boldsymbol{\mu}_i)'\Sigma^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_i)$ is minimised, i.e., allocate $\boldsymbol{x}$ to the category whose mean has the smallest Mahalanobis distance from $\boldsymbol{x}$.

8.7 Discriminant analysis

We use this idea in building a classification method. We assume each group in the training data is generated as random samples from a multivariate Gaussian distribution. So we model each group by a Gaussian distribution, making the choices of means and variances which maximise the likelihood of the data.

As we pointed out above, the discrimination boundary is a quadratic surface in general, and is a linear space if the two variances coincide. So the moral is:

> If you have a reason to think that the two groups are normally distributed, and have a similar variance, you should expect a linear boundary, but otherwise expect a quadratic boundary.

We usually apply this without even checking that the normality assumption is justified—we look at the sample variance matrix for each group informally, and decide whether to model the groups with the same variance matrix, or different ones. So:

Linear discriminant analysis (LDA) is appropriate when the variance matrices of each group are similar; in this case, we model each group by a (multivariate) Gaussian distribution with a common variance matrix, but differing means. Essentially, the method will be to make the optimal choice of means and a common variance matrix for each group, and then to allocate new observations to the group where the pdf is greatest. As we have already seen, the decision boundaries will be linear.

Quadratic discriminant analysis (QDA) is appropriate when the variance matrices of each group are clearly different; in this case, we model each group by a (multivariate) Gaussian distribution with different means and variance matrices. Again, the idea is to make the optimal choices for all the parameters (note that there are more parameters to estimate in this case), and then to allocate new observations as in LDA. This time the decision boundaries will be quadratic.

Note that there are more parameters available to us in QDA than in LDA, so there is more danger of overfitting in QDA.

A more formal approach for deciding whether to use LDA or QDA would be to try both methods, and use cross-validation or bootstrap techniques to decide which is more likely to give good predictions for your data.

For LDA, we model each group by a Gaussian distribution with a common variance matrix. It turns out that the correct choices to make are:

- $\hat{\mu}_k = \bar{x}_k$, the group mean,
- $\hat{\Sigma} = W$, the within groups variance matrix (sometimes known as the *pooled* variance matrix).

Then we allocate new test data $\boldsymbol{x}$ to that category which gives the greatest likelihood to $\boldsymbol{x}$.

Exercise 8.1 Show that when $k = 2$, Example 8.2 reduces to the rule that we should allocate $\boldsymbol{x}$ to category 1 if

$$(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)'\Sigma^{-1}(\boldsymbol{x} - \boldsymbol{\mu}) > 0,$$

where $\boldsymbol{\mu} = (\boldsymbol{\mu}_1 + \boldsymbol{\mu}_2)/2$, i.e., the dividing point/line/plane/hyperplane between two populations with the same variances is linear (it is a hyperplane passing through $\boldsymbol{\mu}$, though not necessarily perpendicular to this line).

Here's an example of this in practice, considering only the sepals for two of the three iris species:

Example 8.3 ***Iris Setosa and Versicolor*** Taking just sepal length and width gives

$$\bar{\boldsymbol{x}}_1 = \begin{pmatrix} 5.01 \\ 3.43 \end{pmatrix}, \quad \bar{\boldsymbol{x}}_2 = \begin{pmatrix} 5.94 \\ 2.77 \end{pmatrix}, \quad S = \begin{pmatrix} 0.195 & 0.092 \\ 0.092 & 0.121 \end{pmatrix},$$

so $(\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_2)'S^{-1} = (-11.44, 14.14)$, so the rule is to allocate $\boldsymbol{x} = (\boldsymbol{x}_1\ \boldsymbol{x}_2)'$ if

$$(-11.44\ 14.14)\begin{pmatrix} x_1 - \frac{1}{2}(5.01 + 5.94) \\ x_2 - \frac{1}{2}(3.43 + 2.77) \end{pmatrix} = -11.44x_1 + 14.14x_2 + 18.74 > 0.$$

A Bayesian version

Finally, there's a *Bayesian* version, where we take account of the prior probabilities of the classes.

In some circumstances, it is sensible to recognise that there are differing a priori probabilities of membership of categories. For example, in medical diagnosis some conditions may be very rare and others very common, although the symptoms (i.e., measurements available) may be very similar for the differing conditions (e.g., flu and polio). In these cases, it is reasonable to make some allowance for this, and shift the balance of classifying towards the more common category.

If the k categories have prior probabilities $\pi_1, \ldots, \pi_k$, then the Bayes Discriminant Rule is to allocate $\boldsymbol{x}$ to that category for which $\pi_i f_i(x)$ is greatest. The Maximum Likelihood Rule is equivalent to a Bayes Discriminant Rule when the probabilities are equal. While we don't generally know the probabilities, we can estimate them from the training data, and take $\hat{\pi}_i = n_i/n$, the proportion of the training data in group i.

It is an easy exercise to show that if $k = 2$, the decision boundaries are still linear for equal variance matrices and quadratic for differing variance matrices. (Indeed, all that happens is that a constant gets added to the formula of the boundary.)

In a similar way, we might try to quantify the costs involved in making errors in classification. In the example above, misdiagnosing a case of polio as flu is likely to have serious consequences, whereas misdiagnosing a case of flu as polio, and treating it more seriously, is likely to have only a small cost. One can adjust the algorithm to deal with this issue also, and again one finds that it simply shifts the decision boundary by a constant.

8.8 Probabilities of miscalculation

When we know (or believe) the distribution of the groups to be normal, we can estimate the probabilities of misclassification. (This is one advantage of this derivation of the method.)

Let $p_{ij} = \mathrm{P}[\text{a type } j \text{ object is classified as type } i]$; the *performance* of a discriminant rule is described by the p_{ij}. Good rules have p_{ij} small for $i \neq j$ and big for $i = j$.

Example 8.4 ***Two populations*** Consider two Gaussian populations $\mathrm{N}_p(\boldsymbol{x} \mid \boldsymbol{\mu}_i, \Sigma)$. Let $\boldsymbol{\mu} = \frac{1}{2}(\boldsymbol{\mu}_1 + \boldsymbol{\mu}_2)$. Then when $\boldsymbol{x}$ is in the first population, we have $y = \boldsymbol{\alpha}'(\boldsymbol{x} - \boldsymbol{\mu}) \sim \mathrm{N}\big(y \bigm| \frac{1}{2}\boldsymbol{\alpha}'(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2), \boldsymbol{\alpha}'\Sigma\boldsymbol{\alpha}\big)$ for any vector $\boldsymbol{\alpha} \in \mathbb{R}^p$.

When we classify $\boldsymbol{x}$ on the value of the discriminant function $h(\boldsymbol{x}) = \boldsymbol{\alpha}'(\boldsymbol{x} - \boldsymbol{\mu})$, with $\boldsymbol{\alpha} = \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$ (classifying as population 1 if $h(\boldsymbol{x}) > 0$), then $h(\boldsymbol{x}) \sim \mathrm{N}\big(h \bigm| \frac{1}{2}\Delta^2, \Delta^2\big))$, where $\Delta^2 = (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)'\Sigma^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$.

Similarly, if $\boldsymbol{x}$ is in population 2, then $h(\boldsymbol{x}) \sim \mathrm{N}\big(h \bigm| -\frac{1}{2}\Delta^2, \Delta^2\big)$.

So $p_{12} = P(h(\boldsymbol{x}) > 0 \mid \boldsymbol{x} \text{ is in population 2}) = \Phi = \Phi\Big(\frac{-\Delta^2/2}{\sqrt{\Delta^2}}\Big) = \Phi\big(-\frac{1}{2}\Delta\big)$, and similarly $p_{21} = \Phi\big(-\frac{1}{2}\Delta\big)$. Thus for two Gaussian populations with a common variance, the misclassification probabilities are equal.

To demonstrate the behaviour, here are a couple of remarks for $p = 2$:

Remark 8.1 (Correlation in the bivariate Gaussian) Consider the case $p = 2$, with Gaussian populations $\mathrm{N}_2(\boldsymbol{x} \mid \boldsymbol{0}, \Sigma)$ and $\mathrm{N}_2(\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma)$, where $\boldsymbol{\mu} = (\mu_1 \ \mu_2)'$, and suppose $\mu_1, \mu_2 > 0$, and $\Sigma = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix} \sigma^2$.

1. First, we examine how the correlation may affect the power of the discriminant function.

   We have $\Delta^2 = \boldsymbol{\mu}'\Sigma^{-1}\boldsymbol{\mu} = \frac{\mu_1^2 + \mu_2^2 - 2\rho\mu_1\mu_2}{\sigma^2(1-\rho^2)}$. If the variables were uncorrelated, then we would have $\Delta^2 = \frac{\mu_1^2 + \mu_2^2}{\sigma^2} = \Delta_0^2$.

   Now the correlation will "improve" the discrimination (i.e., reduce p_{12}) if $\Delta^2 > \Delta_0^2$, which rearranges to

$$\rho\left(\left(1+\left(\frac{\mu_2}{\mu_1}\right)^2\right)\rho-2\left(\frac{\mu_2}{\mu_1}\right)\right)>0,$$

i.e., if $\rho<0$ or $\rho>\frac{2\mu_1\mu_2}{\mu_1^2+\mu_2^2}$. In particular, if $\mu_1=\mu_2$, then any positive correlation reduces the power of discrimination.

2. Now we compare the rule with that based just on the first variable. So if the rule were based just on measurements of the first variable, then $p_{12}=\Phi(-\frac{1}{2}\Delta^*)$, where $\Delta^{*2}=\frac{\mu_1^2}{\sigma^2}$, and so using both variables instead of just one improves the discrimination if $\frac{\mu_1^2+\mu_2^2-2\rho\mu_1\mu_2}{1-\rho^2}>\mu_1^2$, which rearranges to $(\rho\mu_1-\mu_2)^2>0$, which is always the case.

8.9 Estimation

If parameters are estimated, then these can be used to estimate the p_{ij}. For example, in Example 8.4 we can obtain $\hat{p}_{12}=\Phi\left(-\frac{1}{2}\hat{\Delta}\right)$, where

$$\hat{\Delta}^2=(\bar{\boldsymbol{x}}_1-\bar{\boldsymbol{x}}_2)'S^{-1}(\bar{\boldsymbol{x}}_1-\bar{\boldsymbol{x}}_2).$$

In Example 8.3 of discriminating between *iris setosa* and *iris versicola*, this gives $p_{12}=0.013$. We can see how our rule works for the 100 flowers in the set, and see how many are misclassified.

But this is the misclassification rate for the training data, and is likely to be an over-optimistic estimate if we were to use this rule for new test data. Indeed, our rule is formed using the training data, so is optimised for it, and it shouldn't be surprising that new data may not perform as well.

Again, some form of cross-validation, or bootstrapping, should give a more accurate estimate of how the process is working on the data.

8.10 Quadratic discriminant analysis (QDA)

We motivated LDA by the idea of viewing the data from a particular direction. While this was a nice motivation, directions are necessarily linear, and we already know situations where decision boundaries are not linear.

Indeed, we referred to such a situation earlier! If the two groups are generated from multivariate Gaussian distributions, then we would expect a linear boundary *only* if the variance matrices of the two groups were the same. In general, we would expect a quadratic boundary. This motivates quadratic discriminant analysis, where we model our groups as normally distributed, but without any assumption of similar variances.

Then we estimate the group i variance $\hat{\Sigma}_i=\left(\frac{n_i-1}{n_i}\right)S_i$, and classify new observations $\boldsymbol{x}$ by allocating to the group with the largest value of the density function.

Generally discrimination and classification on the training data improves with the number of dimensions and with the complexity of the model—quadratic should generally be better than linear.

Having said that, one expects that the more parameters one has, the better one can do on the training data, but it may be that the classification procedure works less well on future data, owing to the problem of overfitting.

8.11 Summary

We have seen two derivations of what turned out to be the same rule.

We were first motivated by the problem of finding the projection that maximised the separation between two groups. This method gave Fisher's linear discriminant coordinate as the first eigenvector of a certain matrix; one advantage of this method is that (by analogy with PCA) we were led to considering the later eigenvectors of this matrix, and developing a way to potentially separate several groups. Having several coordinates allows us to plot the data with respect to these coordinates, allowing us to get:

- Effective data display using this extra information.
- Dimensionality reduction whilst retaining information on differences between groups.
- Informal interpretation by examination of loadings of the nature of differences between groups.

The second way involved making an additional modelling assumption, that the groups were sampled from a multivariate normal distribution. This method gave the same rule, but doesn't naturally suggest further discriminant functions after the first. On the other hand, the additional assumption allowed us to quantify a probability of misclassification, and also suggested the notion of quadratic discriminant analysis.

9 Decision trees and related methods

Logistic regression and LDA are classification methods with linear boundaries, and QDA was an extension. However, if we have no reason to expect particular shapes of decision boundaries, other methods may well do better.

The other similarity between the methods previously considered is that regression and LDA are parametrised methods, in the sense that we are hypothesising some model for the data (linear, in the simplest cases) that depends on some unknown parameters. We then use an optimisation method (least squares, or maximum likelihood) to construct the most likely model, and use that for future predictions. The methods in this chapter will be purely data analytic; we don't make any assumption of the form of a model where parameters need to be optimised. This means that the model will not suffer from high bias, but it may be prone to overfitting, so we should take some care over this.

Classification and regression trees are similar supervised techniques which are used to analyse problems in a step-by-step approach. These are just generalisations of methods you probably met at school, when it came to recognising things like leaves of trees. You would start by choosing some property—is the leaf rounded? is it convex?—and make decisions about the classification process based on the answers. Generally (at least the examples I did at school!) the answers were yes/no answers based on some characteristics. But exactly the same process can be made to work for more general data—we just need to work out what the right rules are.

Algorithms for generating these trees have been around for a while, but received most prominence from a book "Classification and Regression Trees" (Breiman et al.), and this area of decision trees is often abbreviated CART after the book. Strictly speaking, "CART" refers to a particular technique within the theory. While the principal use is for classification, it is simpler to introduce in the regression case, and the classification situation is a mild refinement, as we shall see.

The CART procedure is a method to produce binary trees given the training data, so at each node, there are two branches, one to the left and one to the right, until we reach the bottom of the tree, and the final nodes ("terminal nodes") have no descendants. The final node will give us the prediction.

9.1 Introduction

The idea is to consider the training data and build a model in the form of a tree. We'll start by going through the terminology for the trees we will be considering.

There are plenty of formal definitions of these on the internet, and they might look rather complicated at first glance. However, the idea is rather simple. But we'll use some formal definitions too!

Definition 9.1 A *graph* is a collection V of points, called *nodes* or *vertices*, and a set E of *edges* going from one vertex to another(so we can regard E as a subset of the Cartesian product $V \times V$). All our graphs will have at most one edge between any given pair (graphs with more than one edge are known as *multigraphs*).

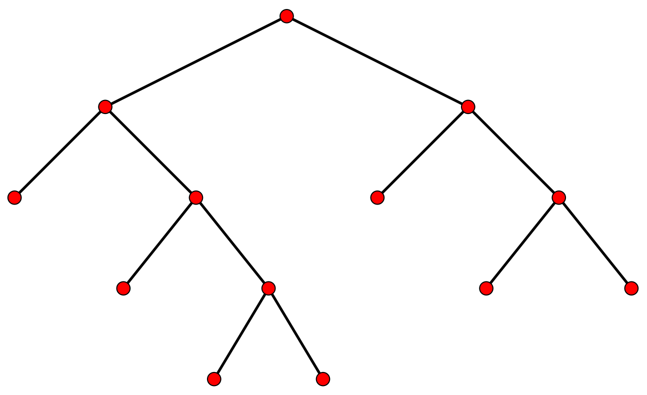
(Actually, what we have defined is often called a *directed graph*, since our edges are really arrows from one vertex to another, with some specified direction, rather than merely a line joining a pair of edges.)

We will be interested in trees. There are many definitions in the literature, but we will use "tree" where pure mathematicians and computer scientists might use "rooted binary tree".

Definition 9.2 A *tree* is a graph where:

- There is a single *root* node, and there are no edges going to this root;
- Every other vertex has exactly one edge going to it;
- Every vertex has either 0 or 2 edges going from it. Nodes with no edges going from it are called *terminal* (or sometimes *leaves*), while those with 2 edges are *internal* (except for the root node). Edges go from the *parent node* to a *child node*.

We will always draw our trees with the root at the top, and all arrows will go downwards. Because of this convention, we won't bother putting arrowheads on our edges to indicate the direction of the line. Here's an example of a tree:



You should be able to see that such trees are in *layers*, where the children of the ith layer are in the $i + 1$st layer; the layer of a node is the number of edges in the (unique) path from the root to the node.

We start with the method for forming the full decision tree for a classification problem.

The whole data set is represented by the root node on the 0th layer of the tree. At every layer, each data point will be at one of the nodes, so each layer is a partition of the whole data set.

Suppose that we have split the data as far as the ith layer. For every node in the ith layer, we define its children in the $i + 1$st layer as follows:

- If all the elements in the node have the same class, stop.
- Otherwise, choose a variable, and a threshold, and form two "child" nodes: the left child, those data points with the value of the variable less than the threshold, and the right child, consisting of the data points with the value of the variable greater than the threshold.

This gives a "recursive partitioning" of the whole data set, and we end when all data points at a given node all have the same class amongst the target features, or there are no more ways to distinguish between the data at that node.

Example 9.1 The sinking of the Titanic in 1912 was one of the most famous in history. Of the 2224 passengers and crew, 1502 were killed.

There is a data set, `titanic.csv`, consisting of the passenger data for 891 passengers on the trip. It has headings:

- `Class`: class of passenger (`1`, `2` or `3`)
- `Sex`: sex of passenger (`male` or `female`)
- `Age`: age of passenger (not all known)
- `SibSp`: number of siblings or spouse aboard
- `ParCh`: number of parents or children aboard
- `Embarked`: port of embarkation (`S` for Southampton, `C` for Cherbourg and `Q` for Queenstown, Ireland)
- `Survived`: `0` for died, `1` for survived

For simplicity, we will remove the observations where `Age` is unknown, which leaves 714 passengers.

We'll also remove the column `Embarked`, since dealing with factor variables where there are more than 2 classes is slightly awkward—we would want to replace this with two binary variables, `EmbarkedS` and `EmbarkedC`, say, which would indicate whether or not the passenger embarked at Southampton or at Cherbourg—but in any case we might guess that this isn't such an important variable.

It would perhaps be sensible to do the same with the `Class` variable, but there is a more natural sense in which a second class passenger is between a first and third class, so we shall retain this as a single variable. (But it isn't really completely true that the second class is exactly half way between the first and third—does this even make sense? As an exercise, you might like to investigate how the analysis below would change if we were to replace the `Class` variable by two binary variables, `Class1` and `Class2`, say.)

Thus we are left with a data set with 6 columns, `Class`, `Age`, `Sex`, `SibSp`, `ParCh` and `Survived`.

We would like to explore whether or not passengers survived, and how survival rates depend on the values of the other variables. We will explain the CART process to form such trees later, but for now will treat this more informally.

Of the 714 passengers in this test set, it turns out that 290 survived, but 424 died. So, without any further information, the best guess at this first stage is that the passenger died. We would be correct with probability $424/714 = 0.594$. But we can do better with more information.

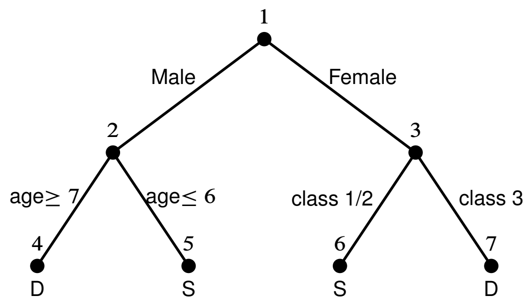
One natural first question is whether a given passenger is male or female. Indeed, it turns out that females had a substantially greater chance of surviving. There were 453 male passengers out of the 714, and only 93 survived, whereas there were 261 female passengers of whom 197 survived. So it looks as if a good first question is the sex of the passenger; it would be reasonable to then predict that males would die and females would survive. This is correct for 360 males and 197 females, so we would be correct with probability $557/714 = 0.780$, a substantial improvement.

Now we think about all the male passengers. Children were quite likely to survive, and it turns out that we might split the male passengers into those aged 7 or above (there were 429 of those, but only 77 survived) and those aged 6 or below (there were 24 of those, but 16 survived).

It turns out that for the female passengers, age is a less good splitting criterion than the class of ticket they held. Of the 159 females in first or second class, 150 survived, whereas only 47 of the 102 in the third class survived.

So if we predict that males die, unless they are aged 6 or below, and that females survive, except for those with third-class tickets, this improves our probability of being correct on the test set to $573/714 = 0.803$.

Then we get a picture whose first two layers will look something like the following:



Here the bottom line indicates our prediction of whether the passenger died or survived.

We could continue this process until each node is “pure”, that is, every passenger corresponding to the node dies, or every passenger corresponding to the node lives, i.e., all the passengers in the node have the same class. This is likely to be an overfitted tree, as it will predict the survival of all 714 passengers in the test data set completely correctly (unless there are two passengers with identical data with different survival outcomes), and we have seen in other contexts that this might imply that test data will be less well predicted. We will explain later how to deal with this.

To summarise the picture above:

- Node 1 is the root node, and contains all 714 passengers. At Node 1, we ask whether the sex of the passenger is male or female. Node 2 will contain the male passengers, and Node 3 will contain the female passengers.
- At Node 2, we ask about the age of the passenger to determine whether to place them in Node 4 or Node 5. Node 4 has the male passengers aged at least 7, and Node 5 has those aged under 7.
- Node 3 consists of the female passengers, and look at their ticket class to decide whether to put them in Node 6 or Node 7. Node 6 contains those in the first or second class; the third class female passengers are in Node 7.

Just using the first two layers of this tree has improved our prediction rate of survival from 59% to 80%, and we could go much further still. As before, we caution against using this as a guide to how the model would perform on unseen test data!

Notice that if we label the root node as 1, we can label the child nodes of the ith node (if it has children) as $2i$ and $2i+1$.

There are many uses of such trees, such as medical diagnosis; marketing studies; approval of bank loans etc.

Trees are determined by three decisions:

1. At each node, we need a procedure to determine the feature, or combination of features, which will divide the data;
2. At each node, we need a method to give the break point, or threshold, that tells us where to divide the data;
3. We need to have a procedure to tell us when not to bother splitting classes further.

One might also insist that trees have a certain maximum depth, or a maximum number of terminal nodes, or a minimum size of data represented by each terminal leaf, etc. There are a lot of ways to tune the parameters, and it is not always obvious that one method is going to be better than another.

Statistical methods for doing this were given by Breiman et al., in the book mentioned above, although there are many others (see the Wikipedia article on *Decision Tree learning*, for example). We will go through the CART algorithm, since it is easiest to discuss, but there are alternative methods, and the most widely used alternatives in practice are the "C4.5 algorithm" and "C5.0 algorithm", due to Ross Quinlan.

We can use these *decision tree* methods for both regression and classification problems. Although the example above is concerned with classification, we shall see that the method works well also for regression, and, in fact, is easier to explain.

9.2 Regression trees

As usual, we suppose that we have a numerical response variable y which depends on a number of input variables $x_1, \dots, x_p$. The aim is to predict the value of y given the values of the inputs; we suppose that we have a lot of training data which will enable us to train a prediction model.

The idea is that at each step, we will select precisely one of the input variables x_i and split the model at a certain point. That is, there will be some value s such that if $x_i < s$, we will go down the left branch of the tree, while if $x_i > s$, we will go down the right branch.

At the first step, we consider all the training data. We consider each input in turn, and for the ith, we want to choose the best point to split. So, for each i and s, we divide the data into two sets, $L_i(s)$ and $R_i(s)$, where $L_i(s) = \{x : x_i < s\}$, and $R_i(s) = \{x : x_i > s\}$.

How do we measure how successful this split is? The method used by CART is to try to find values of c_L and c_R so that the total sum of squares error

$$\sum_{x \in L_i(s)} (y(x) - c_L)^2 + \sum_{x \in R_i(s)} (y(x) - c_R)^2$$

is minimised—so we are minimising the sum of squares of the differences between the predictions c_L and c_R, and the actual values of the response variables. It is easy to see that if i and s are given, then the best choices are to take c_L to be the mean y_L of the response variable for the observations in $L_i(s)$, and c_R to be the mean y_R of the response for the observations in $R_i(s)$. Thus, for any i and s, it is easy to get the minimum value of the squared error.

Next, for any given i, we want to choose s to minimise this minimum squared error. This is also easy (on a computer); we just have to arrange the data in variable i in order, and look at the possible splits between them, and see how it changes as we move s from before one data point to after it.

Finally, we choose the variable i to minimise the minimum squared error.

This procedure explains how to choose i and s to minimise

$$\sum_{x \in L_i(s)} (y(x) - y_L)^2 + \sum_{x \in R_i(s)} (y(x) - y_R)^2$$

where y_L is the mean of responses for the observations in $L_i(s)$, and y_R is the corresponding value for $R_i(s)$.

This provides the split at the top of the tree, and since the process is recursive, we can repeat this at every subsequent node.

Example 9.2 Let's take a very small toy example with just 3 observations on 2 input variables, with 1 response variable. (Of course, one would never work with such a small data set in practice—this is why it is known as a *toy* example!)

```
     X1  X2  Y
Ob1   0   1  1
Ob2   1   0  3
Ob3   2   1  6
```

At the top node, we have all the data. With no other information, our best prediction for any new data is simply the mean $\bar{y}$ of the response variable, which is $\frac{10}{3}$. Just as in Example 9.1, we hope to do better by using extra information from splitting the data.

Then we have 2 choices of variable. For `X1`, we can split between 0 and 1, or between 1 and 2. In the first case, we could split at 0.5, say, and get $L_1(0.5) = \{\texttt{Ob1}\}$ and $R_1(0.5) = \{\texttt{Ob2, Ob3}\}$, and the total sum of squares error is $(1-1)^2 + [(3-4.5)^2 + (6-4.5)^2] = 4.5$. If we split at 1.5, say, we would get $L_1(1.5) = \{\texttt{Ob1, Ob2}\}$ and $R_1(1.5) = \{\texttt{Ob3}\}$, with sum of squares error being $[(1-2)^2 + (3-2)^2] + (6-6)^2 = 2$. But we could also split `X2`, at 0.5, say, and get $L_2(0.5) = \{\texttt{Ob1, Ob3}\}$ and $R_2(0.5) = \{\texttt{Ob2}\}$ with sum of squares error being $[(1-3.5)^2 + (6-3.5)^2] + (3-3)^2 = 12.5$.

So the best first split is to split on `X1` with a value of 1.5. To the left, we get `Ob1` and `Ob2` and to the right we get `Ob3`.

For future data, we ask about `X1`; if it is less than 1.5, we move down the left-hand node, and predict its response as the mean response of the variables in the left-hand node. Since these are `Ob1` and `Ob2`, the mean response here is $\frac{1+3}{2} = 2$, and this is our prediction for the new data. On the other hand, if `X1` is greater than 1.5, our predicted response will be the mean response of the variables in the right-hand node, which just consists of `Ob3`, so our prediction is 6.

We could now progress down the tree and repeat the process to try to improve our prediction. On the right-hand side, there is only one object, but on the left-hand side we might split these two observations with another variable.

Let's do a real-life example. **Example 9.3** There are a number of good sources for data sets online. The data set `airfoil.csv` is taken from https://archive.ics.uci.edu/ml/datasets/Airfoil+Self-Noise, the UCI Machine Learning Repository.

The data set consists of 1503 tests on certain airfoil blade sections conducted in wind tunnels; the aim is to study the scaled sound pressure level `Sound` (a number of decibels) in terms of the other variables. These are:

- `Freq`: Frequency, in Hertz
- `Angle`: Angle of attack, in degrees
- `CLen`: Chord length, in metres
- `FSVel`: Free-stream velocity, in metres per second

- SSDT: Suction side displacement thickness, in metres

The top node of the tree consists of all 1503 observations. The mean Sound of all observations is 124.84 decibels, but the mean square error of Sound is 47.56, so there is quite a wide spread in the data. Each of the 5 variables is tested in turn; for each, the algorithm finds the best choice of threshold which splits the data into two sets so as to reduce the total square error as much as possible.

In fact, we find that the choice which improves our prediction most is whether or not Freq is at least 3575. The 424 observations with Freq greater than 3575 are placed at Node 2, and have mean Sound of 120.42 decibels; the 1079 observations with Freq less than 3575 are placed at Node 3, and have mean Sound of 126.57 decibels. With this prediction, the mean square error reduces to 39.91.

Actually, both Nodes 2 and 3 now have a splitting condition on SSDT. Node 2 splits into those observations with SSDT less than 0.00156 (Node 4), and the observations with SSDT greater than 0.00156 (Node 5). Node 3 splits into those observations with SSDT less than 0.01558 (Node 6) and those with SSDT greater than 0.01558 (Node 7). For each of these nodes, we can make a prediction for the mean, and it turns out that the mean square error is reduced to 28.81.

We can continue (the full details are left for the reader): each of these nodes again splits, and it turns out that Nodes 4 and 6 split again on SSDT, while Node 5 splits on Freq, and Node 7 on CLen. At this point, we have Nodes 8–15, and can compute the mean square error as 24.30.

At each level, we make different predictions for the Sound, and as the mean square error is decreasing, we get more and more confident in our prediction.

If we now have test data with

```
Freq=1000, Angle=3.2, CLen=0.1, FSVel=50, SSDT=0.01,
```

we can trace this element down the tree. At Node 1, we see that Freq<3575 and so we put it into Node 3 in the first layer. At Node 3, we see SSDT< 0.01558, so we put it into Node 6 in the second layer. An examination of SSDT again puts into into Node 12. Since the mean value of Sound in Node 12 is 115.27, this is our prediction for the sound level for our new test data if we stop at this layer.

But we could continue as far as we like, even until each of the 1503 observations in the original training set was in a node by itself. Clearly this is not efficient, and it will also overfit the data. Any noise in the data may affect classification of test data; better would be to truncate the tree so that it stops earlier. So we grow the tree as above, and *prune* the tree so that it is more manageable. There is a trade-off between the amount we grow the tree, and the goodness of fit to the training data.

Again we need a similar regularisation process to that we used for linear models.

To do this, we penalise large trees by making a cost function which we need to minimise. As already mentioned, we measure how well the tree fits the data by the function

$$\sum_t \sum_{N_t} (y - \hat{y}_t)^2,$$

where t runs over the terminal nodes, N_t denotes the observations in node t, and $\hat{y}_t$ is the prediction for the data in N_t (as above, this is just the mean of the response variables over the observations in t). But we introduce a cost for trees which are too large, by adding another term:

$$C = \sum_t \sum_{N_t} (y - \hat{y}_t)^2 + \alpha T,$$

where T is the number of terminal nodes in the tree, and α is a number at our disposal. If we have a lot of terminal nodes in our tree, so that the tree fits the data well, there will be a cost coming from the other term. Conversely, if we have few nodes, the second term will be small, but we may not fit the data well, and the total square error will be large. There will be a trade-off between the contributions from the two terms, and we will want to choose α so that we get a good balance between the terms, with a tree which is not too large fitting the data reasonably well. We can predict the performance of the tree using a cross-validated estimate for the total square error, say, by using k-fold cross validation for k around 5 or 10. Using this with various levels of tree will suggest a good level at which to prune the tree (of course, this is all done automatically by standard computer implementations).

We remark that the alternatives to CART use different strategies to choose the variables and split points, minimising different quantities to the sum of squares given here.

9.3 Classification trees

Classification trees are very similar. However, it is no longer appropriate to measure the success of the fit using a sum of squares, as in regression trees; we need some other measure.

For a node t with data N_t, we write $\hat{p}_{tj}$ for the proportion of observations at node t belonging to class j. We are trying to make each node as "pure" as possible, so that each node correctly predicts one class. There are measures available, and the most commonly used are the *Gini index*

$$\sum_{j=1}^{k} \hat{p}_{tj}(1 - \hat{p}_{tj}),$$

and the *cross entropy*

$$-\sum_{j=1}^{k} \hat{p}_{tj} \log \hat{p}_{tj}.$$

Both vanish for nodes which contain elements all of the same class, and get more and more positive the more evenly split the nodes are.

If $k = 2$, so that there are two classes, and a proportion of p in one class and $1 - p$ in the other, then the Gini index of that node is $2p(1 - p)$, and the cross entropy is $-p \log p - (1 - p) \log(1 - p)$. We'll use the Gini index, but it doesn't really matter (the cross entropy gives similar results; indeed, a plot suggests that they differ very little for $0 \leq p \leq 1$, at least if they are both scaled to have the same range).

We measure how well the tree fits the data by using the function

$$\sum_t \sum_{N_t} |N_t| G_t,$$

where as above, t runs over the terminal nodes, N_t is the set of data descending to t, and G_t is the Gini index of t, and as for regression trees, we add a cost function to allow us to prune fully grown trees back to a more reasonable size, and avoid the problems that come with overfitting.

Example 9.4 Let's do a similar toy example to the one earlier, with two input variables `X1` and `X2` and a class `Cl` which we suppose depends on the input variables, and we want to make a tree to predict the class depending on the input variables:

	X1	X2	Cl
Ob1	0	1	A
Ob2	1	0	A
Ob3	1	1	B
Ob4	2	0	A
Ob5	2	1	B

Now splitting on `X1` at a value of 0.5 gives a division $\{$`Ob1`$\}$ and $\{$`Ob2`, `Ob3`, `Ob4`, `Ob5`$\}$. The left-hand node is clearly pure, just with observations in class A, so has Gini index $1(1-1)+0(1-0) = 0$; the right-hand node has 2 elements in each class, so has Gini index $\frac{1}{2}(1-\frac{1}{2}) + \frac{1}{2}(1-\frac{1}{2}) = \frac{1}{2}$.

Splitting `X1` at 1.5 gives a division $\{$`Ob1`, `Ob2`, `Ob3`$\}$ and $\{$`Ob4`, `Ob5`$\}$. The left-hand node has 2 out of 3 in class A, and the right-hand node has 1 out of 2 in class A. So the Gini index is $\left[\frac{2}{3}(1-\frac{2}{3}) + \frac{1}{3}(1-\frac{1}{3})\right] + \left[\frac{1}{2}(1-\frac{1}{2}) + \frac{1}{2}(1-\frac{1}{2})\right] = \frac{17}{18}$.

Splitting `X2` at 0.5 gives a division $\{$`Ob2`, `Ob4`$\}$ and $\{$`Ob1`, `Ob3`, `Ob5`$\}$. The left-hand node has both in class A, and the right-hand node has 1 out of 3 in class A, so the Gini index of the split is $\left[1(1-1) + 0(1-0)\right] + \left[\frac{2}{3}(1-\frac{2}{3}) + \frac{1}{3}(1-\frac{1}{3})\right] = \frac{4}{9}$. This is the smallest Gini index, and so this is our first split.

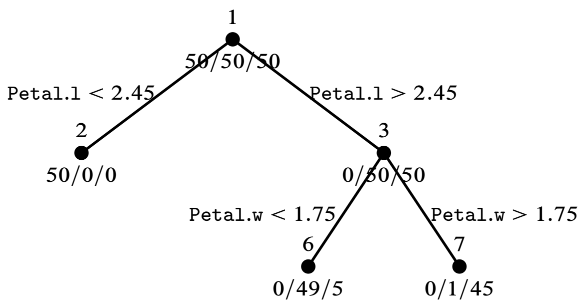
We could now continue and use `X1` to split the left-hand node, but since both elements have the same class, this is not necessary. The right-hand node could split on `X1` at either 0.5 or 1.5; the first of these results in two pure nodes, whereas the second doesn't, so the first would be preferred.

The splittings in Example 9.1 were found using these methods, andso form a good example of how they work in practice.

Example 9.5 For another example, we can consider the iris data `irisnf.csv`, where the objective is to find a set of rules, based on the four measurements we have, of classifying the flowers into one of the four species. It turns out that `Petal.l` being less than 2.45 already splits off the *setosa* variety perfectly (Node 2, say) with *versicolor* and *virginica* forming Node 3. Then Node 6 consists of the observations with `Petal.w` being less than 1.75, and this contains all but one of the *versicolor*, and 5 of the *virginica*, with the remaining *versicolor* and 45 *virginica* being in Node 7.

At this point, we have a reasonably good classification with a small tree, and it turns out that this is a suitable point to stop building the tree.

So we end up with a nice pictorial representation of our data, just like the earlier one for the `titanic.csv` data set, and the numbers of observations in each class have been added to the nodes:



In particular, if we had new test data with

```
Petal.l=2.7, Petal.w=1.5, Sepal.l=4.6, Sepal.w=1.4
```

we would use the decision procedure at Node 1 to place it in Node 3, because the `Petal.l` is more than the threshold; then we would look at `Petal.w`, and decide that it should be in Node 6. Since most of the elements of Node 6 are *versicolors*, we would predict that our new observation was also a *versicolor*.

You might wonder whether we could take advantage of the principal components or linear discriminant coordinates, and use linear combinations (or other functions) of the variables instead, but in practice this is used less than one might think. (In fact, Hastie et al. explain why this is, and suggest alternatives which do make use of linear combinations.)

9.4 Ensemble methods

Trees are extremely quick to produce, as already remarked. They are also very easy to interpret and understand. Indeed, they are widely used in medicine; it is believed that this may be because it is how doctors think when trying to diagnose patients, looking for one variable at a time to try to isolate the particular variables which are anomalous. They can cope with mixtures of numerical and categorical variables.

Because of all these, they are very popular in machine learning.

On the other hand, they are very unstable. Small perturbations in the data set can make a different split early on in the process, and then it is likely that this difference will propagate all the way down the tree, so that there will be no relation between the trees, possibly giving rather different predictions. This means that although individual trees are easy to interpret, we need to be cautious about the interpretation, since small changes to the data might lead to major changes in the interpretation.

Partly for this reason, however, they are not generally very accurate compared with other methods. In the remaining sections of this chapter, we will explain some techniques to improve accuracy which are frequently used in practice. These tend to involve many trees, which leads to a decrease in speed, and a loss of interpretability, but the models produced are much more accurate.

You will all have heard of the idea of the "wisdom of crowds". There's a famous (statistical) example of this, when Francis Galton was surprised at how close the average (both the mean and the median) of a crowd's guesses of the weight of an ox at a county fair was to the true value.

The idea of ensemble methods is to exploit this idea. We use multiple learning methods to get better predictions than from any of the individual methods. Taking averages should reduce the variance of errors in the individual methods. Indeed, we know that if we have a set of n i.i.d. observations with variance σ^2, then the mean would have variance σ^2/n.

Ensemble methods are much more widely applicable than to trees, but trees are an excellent place to introduce the ideas, and, in practice, ensemble methods are much more widely used with trees than with other methods.

9.5 Bagging and Random Forests

Bagging

Given a collection of training data, we can find new sets of training data by sampling from the original set uniformly and with replacement (so that observations may be repeated in the new training set). We can fit the model to each of these new sets of training data, and take the average result (for a regression problem) or a majority vote (for a classification problem).

The collection of new sets of training data arising in this way from old sets is known as a *bootstrap sample*. The method of averaging gives rise to *bootstrap aggregating*, which gets abbreviated to *bagging*.

Tree models are very simple and fast to train, and so we can afford to spend extra time improving our classifier by running many examples on a bootstrap collection. We can fit our tree models to each data set in the bootstrap. Then, for a new piece of test data, we can apply each of the trees to the new data. Each bootstrap sample gives rise to one model, and hence one prediction for the response variable for the new observation.

For a classification problem, each bootstrap sample produces a model which we can use to make a prediction for the class of the new observation. We define the bagged prediction to be the class which is predicted most often by all the models. Sometimes this is also called the *majority vote* prediction.

Regression problems are similar, except that now each bootstrap sample gives a prediction for the numerical value of the response variable for the new observation. Then we take the bagged prediction to be the average of the values of the response variables.

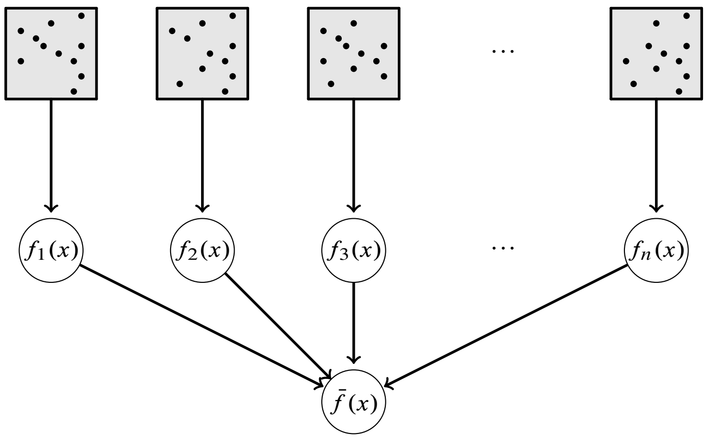
So, for either form of problem, we get a bagged classifier, which obviously depends on the individual randomly chosen bootstrap samples. But we hope that by using a large bootstrap sample, we can reduce the variability of the prediction, thus making our classifier more robust and reliable.

Slightly more formally, we start with a data set X consisting of pairs (x_i, y_i) as in the previous section. We form new (bootstrap) training sets, by sampling with replacement to get $\{X_1, \ldots, X_n\}$. For each of these training sets X_i, we form a classification or regression tree; a new sample x would be classified to be $f_i(x)$, say.

Given an unseen test sample, we would average the predictions

$$\bar{f}(x) = \frac{1}{n}\sum_{i=1}^{n} f_i(x)$$

for a regression problem, and simply take the majority vote in a classification problem.



Random forests

Unfortunately, this is still not quite satisfactory. Using bagging on trees is an improvement on using a single tree, but there is still an issue.

We are hoping that using many classifiers will reduce the variability of our classifier. It is well known that if σ^2 denotes the variance in a single sample, then the variance of the mean of n samples should be σ^2/n, but this is only valid if the samples are independent. Indeed, if we had n samples which were as far from independent as possible—let's imagine they are *identical*—then the mean will just correspond to the value of any of the individual values, and will still have variance σ^2.

Slightly more generally, if samples are correlated with a correlation of $\rho > 0$, then the expected variance is actually $(\rho + \frac{1-\rho}{n})\sigma^2$.

Indeed, the variance of the mean is given by

$$\frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\mathrm{Cov}[X_i,\,X_j].$$

Separating the terms where $i = j$ from the rest, and using the symmetry of the covariance, we get:

$$\mathbb{V}[\bar{X}] = \frac{1}{n^2}(n\,\mathbb{V}[X_1]) + \frac{1}{n^2}(n^2-n)\,\mathrm{Cov}[X_1,\,X_2] = \frac{\sigma^2}{n} + \frac{(n-1)\rho\sigma^2}{n}.$$

Even if we take more and more samples, we see that the variance is tending to $\rho\sigma^2$ as $n \to \infty$. We would certainly hope to reduce the variance of our classifier to be as small as possible.

It is quite likely that many of the trees coming from the bootstrap sample might start with a split on the same first variable. This would mean that the features involved in the early splits will tend to dominate, and the resulting predictions won't be uncorrelated, so that the variance of the mean will not be reduced as much as we might hope.

So the random forest algorithm differs from simple bagging on trees by modifying the procedure at each split, to ensure a greater mixture of variables where the splits take place. Rather than consider all possible variables on which to split at each node, only a random subset is considered. This increases the variability of the trees produced (and also speeds up the method). It may well be that any given tree is considerably worse than the best trees produced by the CART algorithm, but the trees become much more independent, and the calculation above suggests that this might be preferable if it is just our aim to take averages in some sense.

Aside from this difference, random forests work in the same way as bagging: we get many different tree models for each bootstrap data set. Each model will make some prediction, and we can then take the average (for regression) or majority vote (for classification).

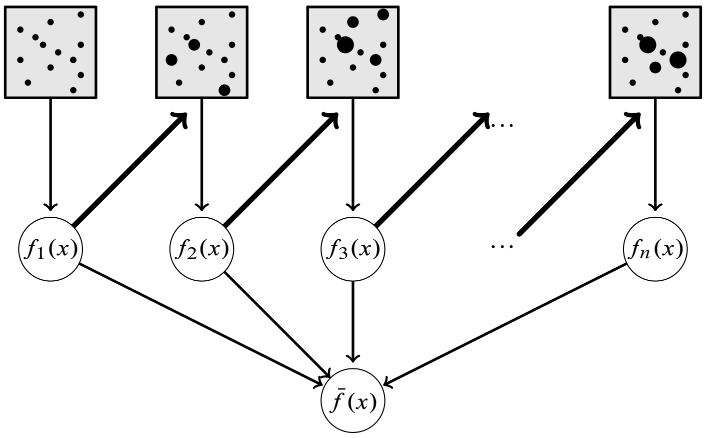
Random forests certainly improve on using a single tree, or on bagging, for many data sets, in that the variance of the method is reduced. When there are just a few features which dominate the classification, however, it may be the case that random forests work less well, as these features may not appear in our list of possible splits until far down many trees.

9.6 Boosting

Random forests are a very good classifying method for general data sets (as opposed to pattern recognition, say, where suitable neural networks may prove more successful), but boosting is another excellent alternative, which often performs even better. Boosting works in a slightly similar way to random forests, in that we combine a large number of less good models to form a combined classifier which is more effective.

Whereas random forests use many individual decision trees, which are independent of each other, boosting uses many individual decision trees but formed sequentially, where each tree depends on the previous one. You can liken the procedure to electronic circuits working in parallel (bagging and random forests) or in series (boosting).

Boosting takes a number of forms, but the common feature of all of them is that we consider a model, look at what it is doing badly at, and improve the model. For example, if we want to consider a classification problem with *adaptive* boosting, we look at the observations which aren't correctly classified, and make them more highly weighted, so that the next classifier looks more carefully at them, penalising future models more for making the same errors.



Boosting can work with other learning methods than trees, although in practice, trees are by far the most common use.

Adaptive Boosting (AdaBoost)

Whereas bagging and random forests work by taking lots of bootstrap sample data, and working out models independently for each, boosting always uses the original data set, but at each stage the elements are weighted depending on whether they have been correctly or incorrectly predicted by the previous stages. We will outline the ideas in the classification situation, and mention the regression situation briefly at the end.

Starting from the original data set, we fit the decision tree model to it. This needn't be a large tree, and in fact, the most common tree used has a single layer, so that there is only one split. Such a tree is often called a (decision) *stump*. We take note of which observations are incorrectly classified by the model. We form the second version of the data set by picking randomly from the original set, but rather than having a uniform probability, we weight the probabilities so that data which was incorrectly classified by the first model is more likely to be picked for the second data set. A model is then fitted, and the procedure repeated. The idea is that the models pay increasing attention to the observations which are often misclassified, and try to correct them.

We will consider the case where we have two classes, and the response variable is given by a class $y \in \{-1, +1\}$. We begin by finding a predictor P_1 for the whole dataset which predicts the class $P_1(x) \in \{-1, +1\}$ for any values of the variables $\boldsymbol{x} = (x_1, \ldots, x_p)$. Let M_1 denote those observations which are misclassified by P_1.

Now we re-weight the data set, so that elements misclassified by P_1 are weighted more highly, and those correctly classified have their weights decreased. We will then form another predictor (again a stump) which best fits the weighted data. Then we let M_2 denote the observations misclassified by P_2.

We can repeat this as often as necessary, and get a series of predictors $P_1, \dots, P_m$, each of which gives a class $P_i(x) \in \{-1, +1\}$ for a variable $\boldsymbol{x} = (x_1, \dots, x_p)$. The predictor P_i will have some confidence level α_i, and we form the overall prediction $\sum_{j=1}^{m} \alpha_j P_j(x)$. If this is positive, we assign $\boldsymbol{x}$ to class $+1$, and if it is negative, we assign $\boldsymbol{x}$ to class -1. That is, the overall prediction is given by

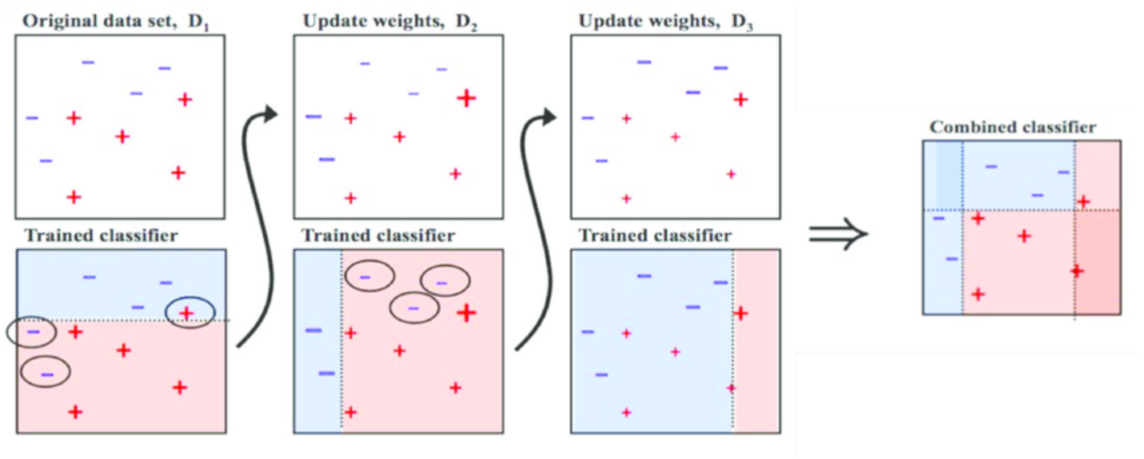
$$P(x) = \operatorname{sign}\left(\sum_{j=1}^{m} \alpha_j P_j(x)\right).$$

Let's describe the algorithm more precisely, and explain the choices of the weights:

1. Initialise the algorithm by making each of the n observations in the training set have weight $w_i^{(1)} = 1/n$. (We will update the weights at each stage so that misclassified observations have their weights increased, so that they become more important at future stages.)
2. For each classifier $P_1, \dots, P_m$ (often a decision stump) in turn:
   (a) Fit the classifier to the (weighted) data, and let M_j denote the set of the observations which are misclassified by P_j;
   (b) Compute the error
   $$\text{error}_j = \sum_{i \in M_j} w_i^{(j)},$$
   the weighted proportion of misclassified observations;
   (c) Define the confidence of the jth prediction by
   $$\alpha_j = \log\left(\frac{1 - \text{error}_j}{\text{error}_j}\right).$$
   (d) Define
   $$w_i^{(j+1)} = \begin{cases} \frac{(1-\text{error}_j)}{\text{error}_j} w_i^{(j)} & \text{if the } i\text{th observation is misclassified by } P_j \\ w_i^{(j)} & \text{otherwise} \end{cases}$$
3. Finally, output the classifier
$$P(x) = \operatorname{sign}\left(\sum_{j=1}^{m} \alpha_j P_j(x)\right).$$

Note that the formulas in 2(c) and 2(d) imply that if there is a low error rate, so that α_j is large, then the re-weighting in (d) is much more pronounced.

Experiments suggest that as m gets larger, so that we have more and more trees contributing to the final classifier, the error on the test data continues to decrease, or at least to stabilise. This contrasts with many other algorithms where, if we make it more and more complex, there comes a point where overfitting starts to occur. Thus AdaBoost seems to be resistant to overfitting, and it seems to be a generally successful algorithm in practice.



We'll give the algorithm in the regression situation for completeness, but without much further comment. In this situation, rather than saying that an object is misclassified, we need to measure how much the predicted value differs from the actual value. So we want to change the measurement of the error from a simple count of the misclassified elements.

More precisely, if P_j predicts a response value $\widehat{y}_i^{(j)}$ for the ith observation when the actual response is y_i, we need to involve the differences $\left|\widehat{y}_i^{(j)} - y_i\right|$ in our algorithm:

1. Initialise the algorithm by making each of the n observations in the training set have weight $w_i^{(1)} = 1/n$, as in the classification situation.
2. For each classifier $P_1, \ldots, P_m$ (often a decision stump) in turn:

   (a) Fit the predictor to the (weighted) data, and let $\widehat{y}_i^{(j)}$ be the prediction given by P_j for the ith observation, with actual response y_i. Define the maximum error for P_j by

   $$E_j = \max_i |\widehat{y}_i^{(j)} - y_i|,$$

   and define the relative error for the ith observation as

   $$L_i^{(j)} = \frac{(\widehat{y}_i^{(j)} - y_i)^2}{E_j^2}.$$

   (there are other possibilities for this function). We define

   $$\text{error}_j = \sum_{i=1}^{n} L_i^{(j)} w_i^{(j)},$$

(b) Define the confidence of the jth prediction by

$$\alpha_j = \frac{1 - \mathrm{error}_j}{\mathrm{error}_j}.$$

(c) Update the weights by

$$w_i^{(j+1)} = \alpha_j^{L_i^{(j)}} w_i^{(j)},$$

and normalise them by scaling them all so that $\sum_{i=1}^{n} w_i^{(j+1)} = 1$.

3. Finally, output the classifier

$$P(x) = \mathrm{sign}\left(\sum_{j=1}^{m} \alpha_j P_j(x)\right).$$

Gradient boosting

Gradient boosting is another form of boosting, this time easier to explain in the regression case. Since gradient boosting, and especially in its implementations `XGBoost` and `lightGBM`, seems to lie at the heart of most current prize-winning solutions to machine learning competitions, it should certainly be included in these notes. (`XGBoost` has several computational enhancements, but otherwise the only difference between it and gradient boosting as given below is that there is an extra regularisation term in the loss function to penalise more complex models.)

Let's first imagine that we have a regression problem with data consisting of n observations, each with 1 response variable y_i depending on the input variables. We fit some model p_0, giving predicted values $y_i^{(0)} = p_0(\boldsymbol{x}_i)$ for the ith observation. Of course, our model p_0 will not be quite right, and the errors are the *residuals* $\varepsilon_i^{(0)} = y_i - y_i^{(0)}$.

Now these error terms can themselves be modelled. That is, we can try to model the data with inputs $\boldsymbol{x}_i$ and output $\varepsilon_i^{(0)}$. We find a model p_1 which approximates the residuals, so that $p_1(\boldsymbol{x}_i) = \varepsilon_i^{(0)} + \varepsilon_i^{(1)}$, for some error $\varepsilon_i^{(1)}$. Recalling that $\varepsilon_i^{(0)} = y_i - p_0(\boldsymbol{x}_i)$, we see that $p_1(\boldsymbol{x}_i) = y_i - p_0(\boldsymbol{x}_i) + \varepsilon_i^{(1)}$, i.e., that

$$y_i = p_0(\boldsymbol{x}_i) + p_1(\boldsymbol{x}_i) + \varepsilon_i^{(1)}.$$

Since we hope that the errors in approximating $\varepsilon_i^{(1)}$ are less than $\varepsilon_i^{(0)}$ itself (in some average sense), we then expect that $p_0(\boldsymbol{x}_i) + p_1(\boldsymbol{x}_i)$ is a better approximation to y_i than our original model $p_0(\boldsymbol{x}_i)$.

And we can repeat this! We try to fit the error $\varepsilon_i^{(1)}$ to the inputs $\boldsymbol{x}_i$, finding a model p_2 with $\varepsilon_i^{(1)} = p_2(\boldsymbol{x}_i) + \varepsilon_i^{(2)}$. This means that

$$y_i = p_0(\boldsymbol{x}_i) + p_1(\boldsymbol{x}_i) + p_2(\boldsymbol{x}_i) + \varepsilon_i^{(2)}.$$

Eventually we would hope that with enough models p_j, the sum $p^{(M)}(\boldsymbol{x}) = \sum_{j=1}^{M} p_j(\boldsymbol{x})$ is a good model for predicting future data.

Obviously there may be an issue here with overfitting if we take too many terms in our sum.

This idea is at the heart of *gradient boosting*. It resembles adaptive boosting, in that we are using the deficiencies of the model at one stage to form the model for the next. But rather than try to fit the points which are misclassified, or which are furthest away from being correct (in a regression situation), we are trying to fit the residual at each stage.

We ought to say what it has to do with the gradient. Generally, the idea of all modelling is to minimise some cost function which describes how well our model is performing. A natural common choice is

$$L(y,\ p) = \sum_{i=1}^{n} (y_i - p(\boldsymbol{x}_i))^2$$

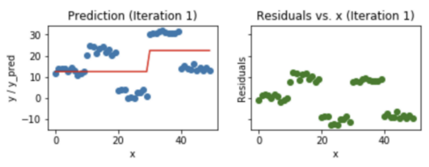
so that $L(y,\ p) = \sum_{i=1}^{n} L(y_i,\ p(\boldsymbol{x}_i))$, with

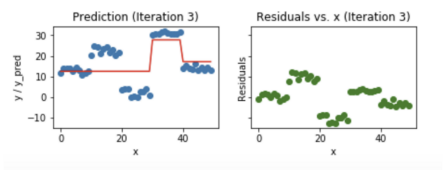
$$L(y_i,\ p(\boldsymbol{x}_i)) = (y_i - p(\boldsymbol{x}_i))^2.$$

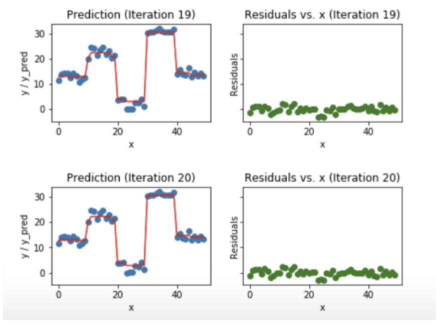
How would we try to minimise this by gradient descent? We would begin with a random function p_0, and then move in the direction which descended most steeply. Since

$$\frac{\partial L(y_i,\ p(\boldsymbol{x}_i))}{\partial p(\boldsymbol{x}_i)} = 2(p(\boldsymbol{x}_i) - y_i),$$

the gradient of the loss function is (up to a constant) exactly the residual. So gradient descent tells us that we should be looking to move in exactly the direction given by the residuals.







Then we can see that we will be increasing or decreasing our predicted model by the *mean* of the under- or overestimates multiplied by the chosen learning rate.

When we looked at gradient descent, we didn't always want to move the full amount in the direction of the gradient, but perhaps some small multiple. We do the same here, and choose some *learning rate* $0 < \eta_j < 1$. Then, as above, we fit the residuals $\epsilon_i^{(j)} = y_i - p^{(j)}(\boldsymbol{x}_i)$ at each stage with a model p_{j+1}, say, but then define $p^{(j+1)} = p^{(j)} + \eta_j p_{j+1}$. The parameters η_j are at our disposal, but it is not uncommon to choose small values such as 0.1 for the learning rates (and I have seen values as low as 0.01 proposed). It seems to be the case, heuristically at least, that models which learn more slowly tend to learn more accurately.

Of course, there are other loss functions, such as

$$L(y_i,\ p(\boldsymbol{x}_i)) = |y_i - p(\boldsymbol{x}_i)|.$$

The corresponding derivative is less interesting here, but as far as gradient descent goes, we will use $\operatorname{sign}(y_i - p(\boldsymbol{x}_i)) = \pm 1$ as our "direction". Then we can see that we will be increasing or decreasing our predicted model by the *median* of the under- or overestimates multiplied by the chosen learning rate.

In fact, it can be shown that AdaBoost is a special case of gradient boosting, if we use the exponential loss function

$$L(y,\ p) = \frac{1}{n}\sum_{i=1}^{n} \exp(-y_i p(\boldsymbol{x}_i)).$$

Let's state the algorithm more fully. Recall that we are trying to minimise some loss function $L(y,\ p)$ which measures the success of our model.

1. We initialise the procedure by letting $P_0(\boldsymbol{x})$ be the constant c that minimises $\sum_{i=1}^{n} L(y_i,\ c)$. (For many common loss functions, this will just be the mean $\bar{y}$ of the responses.)
2. For the predictors $P_1, \ldots$:
   (a) For each element $\boldsymbol{x}_i$, compute the *pseudoresidual*
   $$r_i^{(j)} = -\left(\frac{\partial L(y_i,\ f(\boldsymbol{x}_i))}{\partial f(\boldsymbol{x}_i)}\right),$$
   where $f = P_{j-1}$. (As noted above, when L is the mean square error, this just works out as the residual.)
   (b) Fit a regression tree to the targets $r_i^{(j)}$.
   (c) For every terminal node $t^{(j)}$ of this tree, compute the best value of c to minimise
   $$\gamma_{t^{(j)}} = \sum_{\boldsymbol{x}_i \in t^{(j)}} L\big(y_i,\ P_{j-1}(\boldsymbol{x}_i) + c\big).$$
   (d) Put $P_j(\boldsymbol{x}) = P_{j-1}(\boldsymbol{x}) + \sum_{t^{(j)}} \gamma_{t^{(j)}} I(x \in t^{(j)})$.
3. Put $P(\boldsymbol{x}) = P_M(\boldsymbol{x})$ when M is large enough.

Let's turn to the case of a binary classifier, with classes -1 and $+1$. So now we have a classification problem. Then we need to alter the function we use to measure how well the model is performing. A common loss function is the binomial negative log likelihood:

$$L(y_i,\ p(\boldsymbol{x}_i)) = \log(1 + \exp(-y_i p(\boldsymbol{x}_i))),$$

and we will simply give the algorithm in this case. Then the algorithm becomes:

1. We initialise the procedure by letting $P_0(\boldsymbol{x})$ be $\log(\frac{1+\bar{y}}{1-\bar{y}})$.
2. For the predictors $P_1, \ldots$:
   (a) For each element $\boldsymbol{x}_i$, compute the pseudoresidual
   $$r_i^{(j)} = \frac{y_i}{1 + \exp(y_i P_{j-1}(\boldsymbol{x}_i))}.$$
   (b) Fit a regression tree to the targets $r_i^{(j)}$.
   (c) For every terminal node $t^{(j)}$ of this tree, put
   $$\gamma_{t^{(j)}} = \frac{\sum_{\boldsymbol{x}_i \in t^{(j)}} r_i^{(j)}}{\sum_{\boldsymbol{x}_i \in t^{(j)}} |r_i^{(j)}|(2 - |r_i^{(j)}|)}.$$
   (d) Put $P_j(\boldsymbol{x}_i) = P_{j-1}(\boldsymbol{x}_i) + \sum_{t^{(j)}} \gamma_{t^{(j)}} I(\boldsymbol{x}_i \in t^{(j)})$.
3. Put $P(\boldsymbol{x}) = P_M(\boldsymbol{x})$ when M is large enough.

10 Support Vector Machines

10.1 Introduction

Support Vector Machines (often abbreviated SVMs) are yet another classification tool. If only there was a "best" classifier that worked for every data set! But we will continue to develop a range of classification tools that work well for different data sets. As we shall see at the end of the chapter, though, there is also a version of SVMs which applies to regression problems. For now, we will focus on the classification problem.

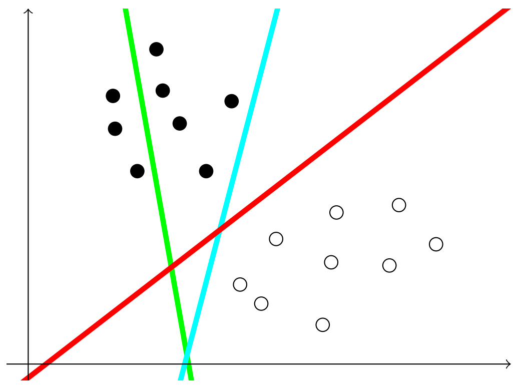
As usual, we have inputs given by vectors $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$, and each has a corresponding class. For simplicity, we'll take the class y to be ± 1. So category A might be assigned $y = -1$, whereas category B gets $y = +1$. The aim is to find a hyperplane that best separates the data.

We'll start with the simplest situation, then make it more complicated, and finally move to the most general situation.

10.2 Separating hyperplanes and margins

In the simplest case, the data points can be plotted in space so that they are *linearly separable*, that is, there are lines (or hyperplanes) that can be drawn so that one side contains all elements in one category, and the other side contains all elements of the second category. By a *hyperplane*, we mean the solution set to a single equation of the form $\beta_0 + \beta_1 \boldsymbol{x}_1 + \cdots + \beta_p \boldsymbol{x}_p = 0$. So in the case where $p = 2$, we simply get a line; when $p = 3$, we get a plane, and when $p > 3$, we get some more general version of a line or plane.

Here's a picture of a linearly separable set when $p = 2$:



In this picture, the green line doesn't separate the data, since some black points are on both sides of the line, but the blue and red ones do.

However, in some sense, the red line looks like it separates the data better, in that the distance of all the points from the red line is quite big—there's quite a margin between the data points and the red line, while for the blue line, it wouldn't take a big change in the data set to change it so that it no longer separates the two classes.

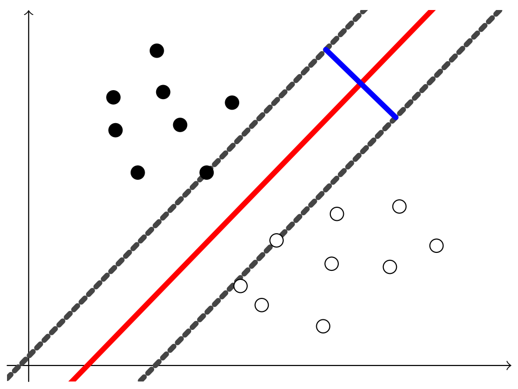
We try to find the *maximum-margin hyperplane*, which is the hyperplane that separates the two groups as much as possible, i.e., that which has all the data points in one group on one side, and all those from the other point on the other side, and whose distance from the data points is as much as possible.

Mathematically, the equation of a hyperplane is $\boldsymbol{w}'\boldsymbol{x} = b$, where b is some number, and $\boldsymbol{w}$ is the vector of coefficients, i.e., $\boldsymbol{w}'\boldsymbol{x} = b = 0$. Translating this up and down, we try to ensure that $\boldsymbol{w}'\boldsymbol{x} - b \geq 1$ if the observation $\boldsymbol{x}$ corresponds to the class $y = 1$ (category B), and $\boldsymbol{w}'\boldsymbol{x} - b \leq -1$ if it corresponds to the class with $y = -1$ (category A). Conveniently, we can combine these into a single equation

$$y(\boldsymbol{w}'\boldsymbol{x} - b) \geq 1$$

for all our observations of a vector $\boldsymbol{x}$ and corresponding class y (this was why we chose ± 1 for the class labels!).

It turns out that the distance between the hyperplanes $\boldsymbol{w}'b - \boldsymbol{x} = 1$ and $\boldsymbol{w}'b - \boldsymbol{x} = -1$ (the length of the blue line below) is $2/\|\boldsymbol{w}\|$, so the *margin* between the separating plane and these hyperplanes is $M = 1/\|\boldsymbol{w}\|$.



So we get a minimisation problem:

Minimise $\|\boldsymbol{w}\|$ (or equivalently $\|\boldsymbol{w}\|^2 = \boldsymbol{w}'\boldsymbol{w}$) subject to the condition $y(\boldsymbol{w}'\boldsymbol{x} - b) \geq 1$ for all observations.

This will be solved by some particular $\boldsymbol{w}$ and b. (We'll go into some of the details of this solution—it uses Lagrange multipliers again—but this is more for completeness; I doubt many people using these ideas are aware of all the mathematical background.)

We've already said that the distance between the hyperplanes is $2/\|\boldsymbol{w}\|$. So we are trying to minimise a function subject to a constraint, and we know that we should use Lagrange multipliers for this. (Actually, it is a little more subtle, since the constraint is given by *inequalities*, not equalities, so it is actually a case where the KKT multipliers are used.)

Notice that the function we are trying to minimise is a quadratic function, as $\|\boldsymbol{w}\|^2 = w_1^2 + \cdots + w_p^2$ if $\boldsymbol{w} = \big(w_1, \ldots, w_p\big)'$. The constraint is a linear constraint on $\boldsymbol{w}$.

10.3 The dual formalism

Let's briefly outline how to solve this. This is a convex optimisation problem, and there are several ways to do this numerically, including some using gradient ascent and variants. For future reference, however, we will use the theory of convex optimisation to translate the problem into the *dual* formulation.

We solve this by using Lagrange multiplier methods (or KKT multipliers). So we define

$$\Omega = \|\boldsymbol{w}\|^2 - \sum_{i=1}^{n} \alpha_i \big[y_i(\boldsymbol{w}'\boldsymbol{x}_i - b) - 1\big],$$

and recall that we need to minimise Ω as $\boldsymbol{w}$ and b vary, subject to the conditions:

- $\frac{\partial \Omega}{\partial \alpha_i} = 0$ for all i, and
- all $\alpha_i \geq 0$.

The latter condition arises because our constraints are *inequalities*, not equalities.

Let's take the partial differentials with respect to $\boldsymbol{w}$ (note that $\|\boldsymbol{w}\|^2 = \boldsymbol{w}'\boldsymbol{w} = \boldsymbol{w}'\mathrm{I}\boldsymbol{w}$, so we can use the earlier results) and b:

$$\frac{\partial \Omega}{\partial \boldsymbol{w}} = 2\boldsymbol{w} - \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i,$$

$$\frac{\partial \Omega}{\partial b} = \sum_{i=1}^{n} \alpha_i y_i.$$

Setting both to 0 gives

$$\boldsymbol{w} = \frac{1}{2} \sum_{i=1}^{n} \alpha_i y_i x_i,$$

$$\sum_{i=1}^{n} \alpha_i y_i = 0.$$

There is an equivalence in convex optimisation problems (like ours) between the solution of the primal problem, as above, and the dual problem, defined by *maximising* the objective function Ω as the α_i vary, subject to the constraints given by the primal problem. So the dual problem is:

Maximise

$$\Omega = \|\boldsymbol{w}\|^2 - \sum_{i=1}^{n} \alpha_i \big[y_i(\boldsymbol{w}'\boldsymbol{x}_i - b) - 1 \big]$$

but where we impose the constraints:

- $\boldsymbol{w} = \frac{1}{2}\sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i$,
- $\sum_{i=1}^{n} \alpha_i y_i = 0$.

Let's substitute these constraints into Ω. First,

$$\|\boldsymbol{w}\|^2 = \boldsymbol{w}'\boldsymbol{w} = \frac{1}{4}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i\alpha_j y_i y_j \boldsymbol{x}_i'\boldsymbol{x}_j.$$

Next, as $\sum_{i=1}^{n} \alpha_i y_i = 0$, we notice that $(\sum_{i=1}^{n} \alpha_i y_i)b = 0$. Finally,

$$\begin{aligned}\sum_{i=1}^{n} \alpha_i y_i \boldsymbol{w}'\boldsymbol{x} &= \sum_{i=1}^{n} \alpha_i y_i \left(\frac{1}{2} \sum_{j=1}^{n} \alpha_j y_j \boldsymbol{x}_j' \right) \boldsymbol{x}_i \\ &= \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i\alpha_j y_i y_j \boldsymbol{x}_j'\boldsymbol{x}_i.\end{aligned}$$

So now we get the dual problem: Maximise

$$\Omega_D = -\frac{1}{4}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i\alpha_j y_i y_j \boldsymbol{x}_i'\boldsymbol{x}_j - \sum_{i=1}^{n} \alpha_i,$$

subject to the constraints that $\sum_{i=1}^{n} \alpha_i y_i = 0$ and $\alpha_i \geq 0$ for all i.

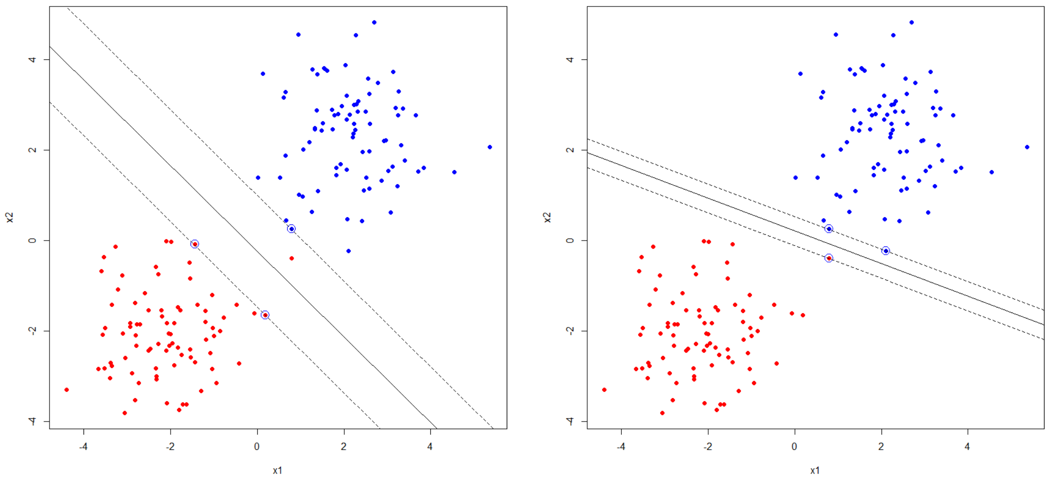
Let's remark that two orthogonal vectors $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ don't contribute to this objective function Ω_D, since their dot product is $\boldsymbol{x}_i'\boldsymbol{x}_j = 0$. Two vectors in the same direction will have a non-zero dot product; let's suppose they point in the same direction so that $\boldsymbol{x}_i'\boldsymbol{x}_j > 0$. If they have the same class, so $y_i = y_j = \pm 1$, then $y_i y_j = +1$, and $\alpha_i\alpha_j y_i y_j \boldsymbol{x}_i'\boldsymbol{x}_j > 0$, and this reduces the value of Ω_D, so these pairs are basically redundant. But if $y_i \neq y_j$, we have $y_i y_j = -1$, so we have opposite classes, and this is contributing in a positive direction to Ω_D. So the pairs contributing positively to Ω_D are precisely those which distinguish the two classes.

It is unusual that either the primal or dual problem can be solved analytically, except in rare situations such as when the data is separable and you can see in advance from the data which vectors will be the support vectors. But there are numerical methods, some based around gradient ascent (although there are other methods), for solving this maximisation problem. Further, since the function we are trying to maximise is convex, there is only one local maximum, and this must be at a global maximum.

10.4 The non-separable case

In the linearly separable case. then, we get a particular separating hyperplane, and we note that it is determined by just a few data points, known as the *support vectors*—these are the observations in the training set which, if removed, would change the position of the separating hyperplane, and are the points lying on the hyperplanes at the margin. In the above picture, there is one black point and one white point which are support vectors (and one white point which looks very close).

There are a couple of issues here. Firstly, the separating hyperplane is not robust; the addition of a new data point can hugely alter the separating hyperplane and the margin. In the diagrams below, compare the two hyperplanes if we regard the red point in the middle of the picture as an outlier, and not to be included in the classification, or not:



But much more seriously, most data sets are not linearly separable! It is very unusual that there is some straight line/hyperplane that separates two sets. We'll explain later how to do something even in this case.

It may be that there is no hyperplane that nicely separates the data. But perhaps only a few vectors lie on the wrong side. We will enhance the problem above by adding a penalty term to vectors lying on the wrong side (this process is another example of regularisation, and we've already seen the idea in other contexts). We fix a value C, which we can think of as a *cost* to having a point on the wrong side of the hyperplane.

Let's put $\xi_i = \max(0, 1 - y_i(\boldsymbol{w}'\boldsymbol{x}_i - b))$, so that ξ_i represents the amount that the ith data point encroaches into the separating band. An error in classification will occur only if $\xi_i \geq 1$, so that the point not only goes into the separating band, but goes over the central hyperplane. So $\sum_i \xi_i$ is an (over-)estimate of the number of errors in classification.

Then we adjust our minimisation problem to:

Minimise $\|w\|^2 + C \sum_i^n \xi_i$.

Remark 10.1 It turns out that in the dual problem, this adjustment to the primal optimisation problem simply becomes a bound $0 \leq \alpha_i \leq C$ for each i, and the problem is little harder numerically than the separable case.

Just like the earlier case, $\boldsymbol{w} = \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i$, and $\alpha_i = 0$ except at support vectors.

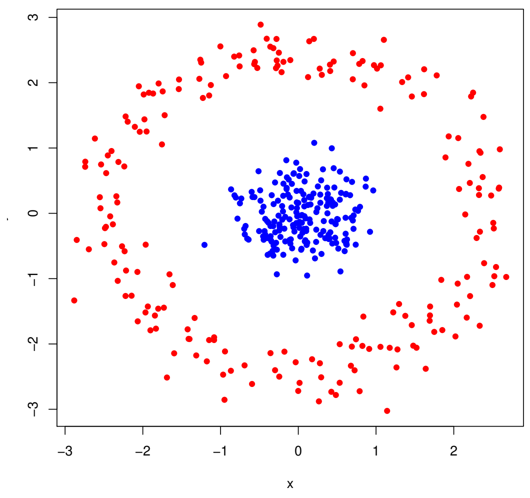
Even in the separable case, it makes sense to use this regularised version, since it allows outliers over the hyperplane with a little penalty, but the hyperplane then separates the bulk of the data in a more optimal way.

Here, C, the penalty for vectors on the wrong side, is at our disposal; we can find the value of C which gives the best prediction for future data using cross validation techniques.

The procedure for finding these separating hyperplanes is called the Support Vector Machine (SVM).

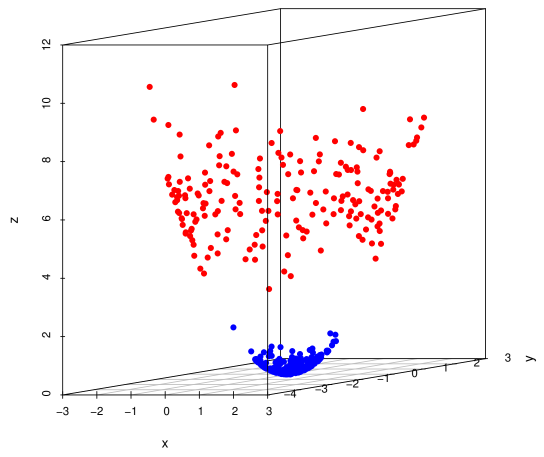
10.5 Kernels

But what if the data is obviously separable, but by something nonlinear? We've already given the following plot:

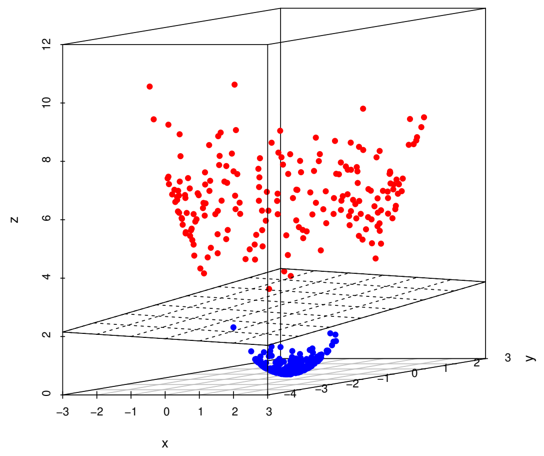


The data is clearly not linearly separable—no straight line will separate the classes, nor do any sort of good job of separating the two classes! And yet it's clear that we can draw some sort of circular boundary between the two classes.

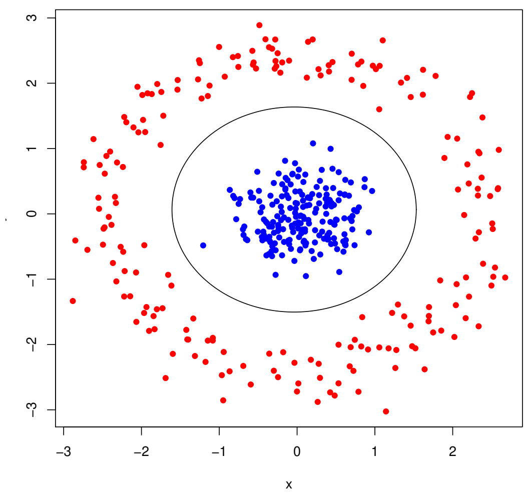
However, although the data naturally lives in $\mathbb{R}^2$, we can map it to $\mathbb{R}^3$ via $(x, y) \mapsto (x, y, x^2+y^2)$, so that it lies on the paraboloid $z = x^2 + y^2$. This gives a set of data points like the following:



where the data clearly are linearly separable:



Then we intersect the separating hyperplane with the paraboloid (which gives a circle), and project back onto $\mathbb{R}^2$ by restricting to the first two coordinates:



This procedure looks promising in this case, and we shall briefly explain how to make it work more generally.

In this example, it looks as if we are mapping the data into 3 dimensions, and doing the SVM there. Effectively, we add to the variables x and y in our data frame a new variable $z = x^2 + y^2$. We might want to add all sorts of functions, such as xy, or just x^2 by itself. It might be more natural to map (x, y) to $(1, x, y, x^2, xy, y^2)$ and use all the monomials of degree at most 2. And perhaps degree 3 would work better still? There are 10 terms of degree at most 3. If we started off with p variables, and use all monomials of degree at most d in these variables, we quickly move into a very high dimensional space, and you might imagine that computing the SVM becomes very time-consuming very quickly.

However, the situation turns out not to be nearly as bad as this, either in our mapping above, or in general. If we look closely at the dual problem (see Section 10.3), we see that everything is written in terms of the dot products between the data points. If we work in higher dimensions, our dual problem there will again be written in terms of the dot products, but in the higher dimension. But if we choose a good mapping, we may be able to get this with no additional computation! This is known as the *kernel trick*. Here's a simple example:

Example 10.1 Let's map 2-dimensional data to degree 2 with the map $(x, y) \mapsto (x^2, \sqrt{2}xy, y^2)$ (the reason for the $\sqrt{2}$ factor will become clear in a moment; it makes the formulae much simpler, but just has the effect of stretching one of the coordinate axes). Let's take the dot product of two elements like this:

$$\begin{pmatrix} x_1^2 \\ \sqrt{2}x_1 y_1 \\ y_1^2 \end{pmatrix} \begin{pmatrix} x_2^2 & \sqrt{2}x_2 y_2 & y_2^2 \end{pmatrix} = x_1^2 x_2^2 + 2x_1 x_2 y_1 y_2 + y_1^2 y_2^2$$

$$= (x_1 x_2 + y_1 y_2)^2 = \left(\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \begin{pmatrix} x_2 & y_2 \end{pmatrix} \right)^2.$$

So we can compute the dot product after taking this degree 2 mapping simply by computing the dot product before, and squaring it. So we don't really need to do the new arithmetic in 3 dimensions—we do it in 2 dimensions, and square the result. We would replace the dot products occurring in the dual SVM optimisation problem by their squares, and we're essentially solving the SVM problem after this degree 2 projection. The hyperplanes in this setting are those of the form $w_1x^2 + w_2\sqrt{2}xy + w_3y^2 = b$, so these are ellipses (or parabolas, hyperbolas etc.). So this trick of transforming coordinates can (a) give quadratic boundaries, and (b) needs hardly any more computation than the original linear SVM problem.

The *kernel* is a sort of generalised dot product. In our dual optimisation problem, we replace all occurrences of the dot product $\boldsymbol{x}_i'\boldsymbol{x}_j$ of $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ with a function $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ which is easy to compute in terms of the original dot products, and which still has some appropriate properties. So basically, we are choosing some mapping ϕ of our original variables into a space where the images are more separable; there, the dot products would be between $\phi(\boldsymbol{x}_i)$ and $\phi(\boldsymbol{x}_j)$, and we hope that there is an easy-to-compute kernel function K such that $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)'\phi(\boldsymbol{x}_j)$. Then our dual problem involves

$$\Omega_D = -\frac{1}{4}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) - \sum_{i=1}^{n}\alpha_i.$$

The *polynomial kernel* is defined by

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i'\boldsymbol{x}_j + 1)^d,$$

where d is the degree. Computing $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ requires hardly any more work than computing $\boldsymbol{x}_i'\boldsymbol{x}_j$. However, with this kernel, we are essentially allowing ourselves to have any decision boundary which looks like a degree d function, which is much more general than simply a line. For example, with $d = 2$, this kernel function is equivalent to expanding the original variables to 2nd degree polynomials; in the case of a variable (x_1, x_2), we are mapping it to 5-dimensional feature space of $(x_1, x_2, x_1^2, x_2^2, x_1x_2)$. We get quadratic decision boundaries for very little extra work.

The *radial basis kernel* is another sort of kernel, which can be thought of as having a Gaussian at each point, and measuring the interaction between two points. For this, we define the function to be

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = e^{-\gamma\|\boldsymbol{x}_i-\boldsymbol{x}_j\|^2}.$$

(Sometimes you may see this written with $\gamma = 1/2\sigma^2$, so that $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = e^{-\|\boldsymbol{x}_i-\boldsymbol{x}_j\|^2/2\sigma^2}$, which emphasises the relation with the Gaussian.) It's hard even to imagine exactly what the corresponding map ϕ does in this case—it's a map from $\mathbb{R}^p$ to some infinite dimensional space.

This gives boundaries which are much more local in nature. The parameter γ can be tuned to affect how quickly the Gaussians die away. One could allow different parameters at each point, chosen so that some separating margin is maximised, for example, and many other refinements are possible.

The radial basis kernel is the one most frequently used in practice; the polynomial kernel is also used—while other kernels exist, they are more rarely found in practice.

Remark 10.2 There are other methods we have seen which also can be rewritten in terms of the dot products between data points. So we can use the kernel trick also for other algorithms, and kernels are a very useful tool throughout many parts of statistical learning. In fact, PCA is an example, and kernel PCA is widely used.

10.6 Support vector regression

There is a version of support vector machines for regression, known, unsurprisingly, as support vector regression (SVR). As we remarked above, the model given by the SVM depends only on the *support vectors*, a subset of all the points. We do something similar for the regression problem, and try to produce a model which depends only on a subset of the points.

If we are given, as usual, a response variable y which we want to relate to input variables $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_p$, we try to minimise $\|w\|^2$, subject to $|y_i - \boldsymbol{w}'\boldsymbol{x}_i - b| \le \epsilon$ for all i. So we predict a linear model $\boldsymbol{w}'\boldsymbol{x}_i + b$, and compare it with the actual value y.

So we are hoping to fix ϵ, which is a margin either side of the line $y_i = \boldsymbol{w}'\boldsymbol{x}_i + b$, and hope that all data points lie inside this margin. Again, we can impose a cost C for points which go outside the boundary to get a more regularised version of the optimisation problem. Further, there is a kernel version also; we can imagine that we map the data point $\boldsymbol{x}_i$ via some map ϕ to get nonlinear functions. You can explore these ideas in the Lab sheet on support vector machines.

10.7 Multiclass extensions

Just as for logistic regression, we have the two obvious ways to extend the theory to more than two classes: *one-versus-one* and *one-versus-all*.

11 Neural networks

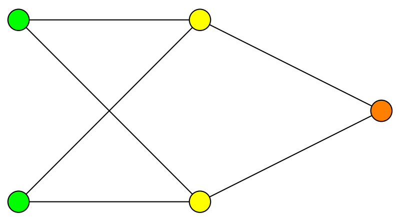
11.1 Introduction

Neural networks are a relatively new invention—although they first appeared in the literature in the 1960s, it is only really in the last few years that computing power has developed enough to make them a valuable addition to the machine learning field. They were originally postulated as a way to mimic the operation of neurons in the brain—I don't believe this is completely understood even now, but we'll focus on the method of operation and the mathematics involved.

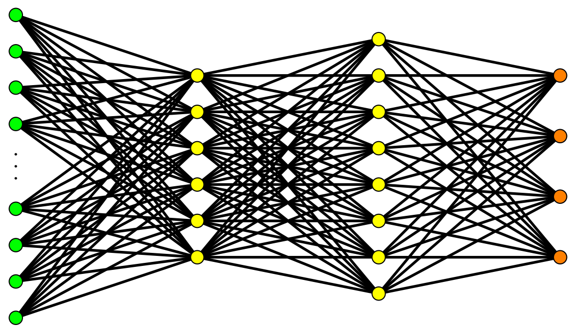
Again, we have some input variables $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_p$, and we want to use these to predict an output class, based on a load of training data.

The idea is that we form a graph with some nodes corresponding to inputs and another node or nodes corresponding to outputs, and with some extra nodes between them. These extra nodes are organised into layers, and the output of each layer is the input to the next layer.

Here is a very simple network architecture, with two input nodes (in green), one "hidden" layer with two nodes (yellow), and one output node (orange):



Of course, graphs might be much more complicated (and we might have more hidden layers still, with many nodes each):



We imagine that the data is input to the left-hand side, and travels through the network left to right, ending up with the output. We can see the inputs, and we get the output out; the other layers will be hidden in the sense that we don't see them directly. Each node is meant to play the role of a neuron in the brain; these neurons either fire or not at a given time, and they fire in response to the firing of the neurons connected to them.

At each node in the hidden and output layers, we imagine that the output is a function of the inputs, and we use a linear model: in particular, if the inputs to a given node are $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k$, our model is to output 1 if some linear combination $w_1\boldsymbol{a}_1 + \cdots + w_k\boldsymbol{a}_k$ is above some threshold b, usually known as the *bias*, and 0 if not. So:

$$\text{output} = \begin{cases} 1 & \text{if } w_1\boldsymbol{a}_1 + \cdots + w_k\boldsymbol{a}_k - b > 0, \\ 0 & \text{if } w_1\boldsymbol{a}_1 + \cdots + w_k\boldsymbol{a}_k - b < 0. \end{cases}$$

(The output value of 1 corresponds to the neuron firing, whereas an output of 0 means that the neuron does not fire.)

These parameters w_i, known as *weights*, and the bias b will be different for each node in the graph. Already in the simplest example above, the two yellow nodes will have two weights each and one bias each, and the orange node will also have two weights and a bias, so there are 9 parameters in this simple model. With more nodes, and more layers, the number of parameters increases very quickly; all of them are at our disposal, and we will need to be aware of the danger of overfitting.

Let's suppose that we want to classify an input into one of two classes. Then we would choose our network to have a single output node, and we assign one category if the output node is 0, and the other if the output node is 1.

Initially, we might make random choices for the parameters $w_1, \ldots, w_k$ and threshold b at each node. Using the training data, we want to learn the best values of these parameters. So the input data is sent into the model, and we get an output of 0 or 1 at the end. This is compared with the actual known category to give a score for the parameters.

Since we are going to be using (stochastic) gradient descent, we will need to be able to differentiate the score function with respect to the parameters in order to find the best direction to move. Unfortunately, with our simple 0/1 output model, this is not differentiable, and so we replace it (for the moment) with the logistic function, or sigmoid, we saw earlier: $\sigma(t) = \frac{e^t}{1+e^t}$. Thus the output becomes

$$\text{output} = \sigma(w_1\boldsymbol{a}_1 + \cdots + w_k\boldsymbol{a}_k - b).$$

You can also convince yourselves that if every node is essentially a linear model, then the final output is also a linear function of the inputs, so the hidden layers don't really contribute anything; the nonlinearity introduced by the sigmoid function permits a much wider variety of relations between the outputs and inputs. Indeed, it is known that any "nice" function can be modelled using an appropriate neural network and the sigmoid function.

It is sensible to use values between 0 and 1 throughout the network, and in particular, we can do this for the inputs (for images, colour levels are generally specified between 0 and 255; we scale these to be between 0 and 1 by dividing through by 255, so we could deal with image recognition problems using inputs of $0/255, 1/255, \ldots, 254/255, 1$). Similarly to logistic regression, the final output layer can now return a probability for each class ("cat", "dog", etc.) rather than either 0 or 1. The number stored at a node is sometimes known as its *activation*.

So the idea is that activations in one layer determine the activations in the next layer, similarly to the behaviour of neurons. In this way, the original data, which is fed into the input layer, activates nodes in the successive layers, ending with the nodes in the output layer.

Remark 11.1 One could view logistic regression as a neural network with one output node, and no hidden layers, and then think of neural networks as being some sort of nonlinear version of logistic regression, where a logistic regression is taking place at each node of a graph. Some other classification techniques could be viewed as special cases of neural networks also.

Let's turn to a more mathematical formulation.

11.2 Notation

As you can imagine, it's hard to find good notation for neural networks, since there are an awful lot of variables and parameters to deal with. Let's write $a_1^{(0)}, \dots, a_{n_0}^{(0)}$ for the activations at the input layer—i.e., the input variables, so that there are n_0 nodes in the input layer. In the first layer, we will write the activations as $a_1^{(1)}, \dots, a_{n_1}^{(1)}$. The formula above tells us that the activation of the ith node in the first hidden layer is

$$a_i^{(1)} = \sigma(w_{i,1}^{(0)} a_1^{(0)} + \dots + w_{i,n_0}^{(0)} a_{n_0}^{(0)} - b_i^{(0)}), \tag{11.1}$$

where $w_{i,j}^{(0)}$ is the weight corresponding to the edge from the jth node in the input layer to the ith node in the first layer, and $b_i^{(0)}$ is the bias at the ith node in the first layer.

As usual, it is more convenient to write this as a matrix calculation. Let's write the activations at each layer as a vector:

$$\boldsymbol{a}^{(0)} = \begin{pmatrix} a_1^{(0)} \\ \vdots \\ a_{n_0}^{(0)} \end{pmatrix}, \qquad a^{(1)} = \begin{pmatrix} a_1^{(1)} \\ \vdots \\ a_{n_1}^{(1)} \end{pmatrix},$$

the weights at the first hidden layer as

$$W^{(0)} = \begin{pmatrix} w_{1,1}^{(0)} & \dots & w_{1,n_0}^{(0)} \\ \vdots & \ddots & \vdots \\ w_{n_1,1}^{(0)} & \dots & w_{n_1,n_0}^{(0)} \end{pmatrix},$$

and the bias at this layer as

$$\boldsymbol{b}^{(0)} = \begin{pmatrix} b_1^{(0)} \\ \vdots \\ b_{n_1}^{(0)} \end{pmatrix}.$$

Then the quantity in parentheses in (11.1) is the ith row of $W^{(0)}\boldsymbol{a}^{(0)} - \boldsymbol{b}^{(0)}$. If we write $\sigma(v)$ for a vector v for the vector with σ applied to each entry of the vector, we can abbreviate (11.1) by

$$\boldsymbol{a}^{(1)} = \sigma(W^{(0)}\boldsymbol{a}^{(0)} - \boldsymbol{b}^{(0)}).$$

Note that:

- $\boldsymbol{a}^{(0)}$ is a vector of length n_0;
- $W^{(0)}$ is an $n_1 \times n_0$-matrix, so $W^{(0)}\boldsymbol{a}^{(0)}$ is a vector of length n_1;
- $\boldsymbol{b}^{(0)}$ and $\boldsymbol{a}^{(1)}$ are vectors of length n_1.

Exactly the same will happen at the next layer also: we will have activations, a weight matrix, and biases given by

$$\boldsymbol{a}^{(2)} = \begin{pmatrix} a_1^{(2)} \\ \vdots \\ a_{n_2}^{(2)} \end{pmatrix}, \qquad W^{(1)} = \begin{pmatrix} w_{1,1}^{(1)} & \dots & w_{1,n_1}^{(1)} \\ \vdots & \ddots & \vdots \\ w_{n_2,1}^{(1)} & \dots & w_{n_2,n_1}^{(1)} \end{pmatrix}, \qquad b^{(1)} = \begin{pmatrix} b_1^{(1)} \\ \vdots \\ b_{n_2}^{(1)} \end{pmatrix}.$$

These will combine to give the activations at the second layer:

$$\boldsymbol{a}^{(2)} = \sigma(W^{(1)}\boldsymbol{a}^{(1)} - b^{(1)}).$$

Clearly this extends to any number of layers.

Suppose that the layers are numbered $0, \dots, L$, so that the terms involving the input layer have a superscript $*^{(0)}$, and similarly the terms in the output layer have a superscript $*^{(L)}$. We let n_k denote the number of nodes in the kth layer, and label them with the numbers $1, \dots, n_k$.

Although the matrix formulation above is nicest to write down, we will prove some results in the next section for which the individual entries are required, so we will still need (11.1).

11.3 Backpropagation

Next, we need to see how far the results are from the correct classes, and update the weights and biases to improve the classification.

We will do this with gradient descent, so we need to choose a cost function, to measure how far the results are from the correct classes.

A common choice, of course, is the *mean square error*,

$$C = \frac{1}{n} \sum_{\text{inputs}} (\hat{y}_k - y_k)^2,$$

where $\hat{y}_k$ is the predicted class for the input x_k ($k = 1, \dots, n$) and y_k is the actual class. We need to see how C changes when we adjust the weights and biases. When we introduced (stochastic) gradient descent, we indicated that one way to do this is to adjust each parameter a little in turn, and see how the predictions change, and therefore how the cost changes. Unfortunately, neural networks may have many thousands, if not millions, of parameters, and this approach is impracticable, even with very small batches in the stochastic descent.

But this cost function has some nice features:

- C is differentiable;
- C is an average over the costs of each input;
- C depends on the parameters,

which will allow us to perform *backpropagation*, which is a clever way to compute all the partial derivatives we need without any further training. The equations are slightly intricate, but no mathematics beyond the Chain Rule is needed.

For simplicity, we will define the input to the jth node in the ℓ-th layer as

$$z_j^{(\ell)} = \left(\sum_{k=1}^{n_\ell} w_{jk}^{(\ell-1)} a_k^{(\ell-1)}\right) - b_j^{(\ell-1)}.$$

So we can rewrite (11.1) as

$$\boldsymbol{a}_j^{(\ell)} = \sigma(z_j^{(\ell)}).$$

We want to work out the partial derivatives of C with respect to all the weights and biases. It turns out that a convenient definition is:

$$\delta_j^{(\ell)} = \frac{\partial C}{\partial z_j^{(\ell)}}.$$

First, we see how to compute the values of $\delta_j^{(L)}$—recall that the Lth layer is the output layer. After that, we will explain how to compute the $\delta_j^{(L-1)}$, $\delta_j^{(L-2)}$, … recursively, and finally write the partial derivatives we want to compute in terms of these values.

Proposition 11.1 We have

$$\delta_j^{(L)} = \frac{\partial C}{\partial \boldsymbol{a}_j^{(L)}}.\sigma'(z_j^{(L)}).$$

Proof We use the Chain Rule to see that

$$\delta_j^{(L)} = \sum_{k=1}^{n_L} \frac{\partial C}{\partial a_k^{(L)}}.\frac{\partial a_k^{(L)}}{\partial z_j^{(L)}} = \frac{\partial C}{\partial a_j^{(L)}}.\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}}$$

as the only output $\hat{y}_k = a_k^{(L)}$ with any dependence on $z_j^{(L)}$ is $a_j^{(L)}$. Now we note simply that $a_j^{(L)} = \sigma(z_j^{(L)})$, and the result follows.

Before the next step, let's remark that $z_j^{(L)}$ will have been computed in the training stage for the network. We know σ, and can therefore work out $\sigma'(z_j^{(L)})$. We will have an explicit cost function

C, so that we can work out $\frac{\partial C}{\partial \boldsymbol{a}_j^{(L)}}$—recall that $\boldsymbol{a}_j^{(L)}$ is the activation in the last layer, which is just the output $\hat{y}_j$, so that we can compute the partial derivative easily. For example, if a particular input $\boldsymbol{x}$ should have output $(y_1, \ldots, y_{n_L})$, whereas the model predicts $(\hat{y}_1, \ldots, \hat{y}_{n_L}) = (a_1^{(L)}, \ldots, a_{n_L}^{(L)})$, then its cost is $C_x = \sum_{k=1}^{n_L} \left(y_k - \boldsymbol{a}_k^{(L)}\right)^2$, and so we have $\frac{\partial C}{\partial \boldsymbol{a}_j^{(L)}} = 2(\boldsymbol{a}_j^{(L)} - y_j)$.

All this means that with no extra training, we can compute the values of $\delta_j^{(L)}$ $(j = 1, \ldots, n_L)$ at the output layer.

Now let's give a formula to relate the values of $\delta_j^{(\ell)}$ on the ℓth layer with the corresponding values on the $(\ell + 1)$-th layer:

Proposition 11.2 We have:

$$\delta_j^{(\ell)} = \sum_{k=1}^{n_\ell} \delta_k^{(\ell+1)} w_{kj}^{(\ell)} \sigma'(z_j^{(\ell)}).$$

Proof We argue again with the Chain Rule, this time at the lth layer:

$$\delta_j^{(\ell)} = \frac{\partial C}{\partial z_j^{(\ell)}} = \sum_{k=1}^{n_\ell} \frac{\partial C}{\partial z_k^{(\ell+1)}} . \frac{\partial z_k^{(\ell+1)}}{\partial z_j^{(\ell)}} = \sum_{k=1}^{n_\ell} \delta_k^{(\ell+1)} \frac{\partial z_k^{(\ell+1)}}{\partial z_j^{(\ell)}}.$$

Now we recall that

$$z_k^{(\ell+1)} = \left(\sum_{i=1}^{n_\ell} w_{ki}^{(\ell)} a_i^{(\ell)}\right) - b_k^{(\ell)} = \left(\sum_{i=1}^{n_\ell} w_{ki}^{(\ell)} \sigma(z_i^{(\ell)})\right) - b_k^{(\ell)},$$

so that

$$\frac{\partial z_k^{(\ell+1)}}{\partial z_j^{(\ell)}} = w_{kj}^{(\ell)} \sigma'(z_j^{(\ell)}).$$

Combining these, we see that

$$\delta_j^{(\ell)} = \sum_{k=1}^{n_\ell} \delta_k^{(\ell+1)} w_{kj}^{(\ell)} \sigma'(z_j^{(\ell)}),$$

as required.

Since we know all the $\delta_j^{(L)}$, we can use the formula in the proposition to work out all the $\delta_j^{(L-1)}$, and then all the $\delta_j^{(L-2)}$, etc., recursively. Again, notice that all the terms we need are already computed as part of the training.

The next two propositions will tell us how to compute our desired partial derivatives in terms of all these $\delta_j^{(\ell)}$. First we consider the biases:

Proposition 11.3 We have

$$\frac{\partial C}{\partial b_j^{(\ell-1)}} = -\delta_j^{(\ell)}.$$

Proof This is simply because

$$z_j^{(\ell)} = \left(\sum_{k=1}^{n_\ell} w_{jk}^{(\ell-1)} a_k^{(\ell-1)}\right) - b_j^{(\ell-1)},$$

so a small change in $b_j^{(\ell-1)}$ gives an opposite small change in $z_j^{(\ell)}$, and so

$$\frac{\partial C}{\partial b_j^{(\ell-1)}} = -\frac{\partial C}{\partial z_j^{(\ell)}} = -\delta_j^{(\ell)}.$$

Finally we consider the partial derivatives with respect to the weights.

Proposition 11.4

$$\frac{\partial C}{\partial w_{jk}^{(\ell-1)}} = a_k^{(\ell-1)} \delta_j^{(\ell)}.$$

Proof Again,

$$z_j^{(\ell)} = \left(\sum_{k=1}^{n_\ell} w_{jk}^{(\ell-1)} a_k^{(\ell-1)}\right) - b_j^{(\ell-1)}$$

and so a small change in $w_{jk}^{(\ell-1)}$ makes $z_j^{(\ell)}$ change by $a_k^{(\ell-1)}$ times as much. Recalling the definition of partial differentiation, this means that

$$\frac{\partial C}{\partial w_{jk}^{(\ell-1)}} = a_k^{(\ell-1)} \frac{\partial C}{\partial z_j^{(\ell)}},$$

and the result follows.

So using these formulae, we can easily recover all the values of the partial derivative of the cost function with respect to the weights and biases.

For computer implementation, one should remark that it is fairly straightforward to write all of these in a way which vectorises easily, and which can therefore be computed quickly.

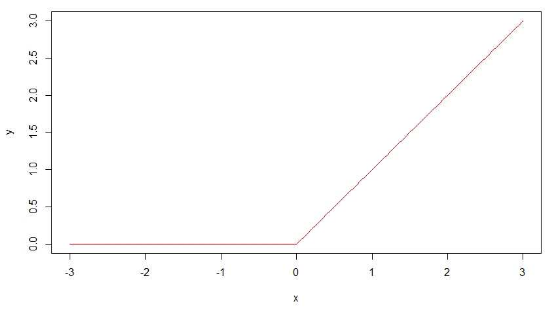
11.4 Variants

In practice, we can choose not only the architecture of the neural network, but also other features. For example, we can choose the procedure for the stochastic gradient descent; in practice, methods known as `adam` or `rmsprop` are often used. These are variants of SGD in which the learning rate changes as the gradient flattens out.

Another alteration we can make is in the use of the sigmoid function. Although it arises from logistic regression, actually we don't really require the activations to stay between 0 and 1 here, just not to vary too far from them. It turns out that if we replace the sigmoid with the *rectified linear unit* (or "relu" for short), given by

$$\mathrm{relu}(x) = \max(0, x),$$

and with graph



then we tend to get better convergence in practice.

(Indeed, it is easy to compute that the maximum value of the derivative of the sigmoid function occurs at 0, when the derivative is $\frac{1}{4}$. This implies that if two inputs to the sigmoid function differ by a certain amount, then the corresponding outputs differ by at most a quarter of this amount. Since we may want to have several layers in our neural networks, and each reduces the differences by a factor of at least 4, it doesn't take many layers before all differences are flattened out and hidden within noise. This is the *vanishing gradient problem*, and the relu function avoids this.)

Finally, the mean square error cost function is often replaced with a different cost; for example, in binary classification, with classes 0 or 1, we might use the (binary) cross-entropy function

$$C = -\frac{1}{n}\sum_x [y \log \hat{y} + (1-y)\ln(1-\hat{y})],$$

where n denotes the number of inputs x, and y is the actual class with $\hat{y}$ being the class predicted by the neural network. Notice first that this C shares the properties listed for the mean square error in the previous section—it is differentiable, an average over costs for each input, and depends on the parameters. This is all that we needed for backpropagation to hold.

Notice that since $0 \le y \le 1$, and $0 \le \hat{y} \le 1$, each logarithm is of a value between 0 and 1, so produces something non-positive; this means that $C \ge 0$. We can also see that the closer $\hat{y}$ is to y for each input, the closer that C is to 0. Indeed, if the correct class for an input x is $y = 1$, then $1 - y = 0$, so the contribution to C is just $-\frac{1}{n}\ln\hat{y}$, so the nearer $\hat{y}$ is to 1, the smaller the contribution. Similarly, if the correct class for an input x is $y = 0$, then the contribution to C is $-\frac{1}{n}\ln(1-\hat{y})$, and again, the nearer $\hat{y}$ is to 0, the smaller the contribution. So this is a plausible choice for a cost function.

In fact, it has further properties that make it more suitable than the mean square error in the binary classification case; one can show that for a single neuron with inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, weights

$w_1, \ldots, w_n$, desired output y and actual output $\sigma(z)$ with $z = w_1x_1 + \cdots + w_nx_n - b$, that

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_{\boldsymbol{x}} x_j(\sigma(z) - y),$$

so that the gradient of C is proportional to the error $\sigma(z) - y$, which is a reasonable property to impose (indeed, this is how one might derive the cross-entropy as a suitable cost function). The larger the error, the faster the cost function should decrease, and the faster the neuron should learn.

11.5 Regularisation

We have already noted that overfitting is a danger with neural networks. In other methods earlier, we used a notion of "regularisation" to penalise models which appeared to show signs of overfitting, and we can do the same here.

There are various ways to do this:

- We can impose a penalty to encourage the weights in the neural network to be small. We can use L_2-*regularisation*, and define a cost by

  $$C = \text{cost} + \alpha \sum \boldsymbol{w}^2,$$

  where α is a parameter we can choose (to optimise the prediction on a validation set, for example), and where we take the sum over all weights in the network. This encourages the network to learn small weights. Although it may not be obvious that this discourages overfitting, it does; the idea is that small weights won't respond too much to changes in the inputs, and in particular, won't respond too much to random noise; however, if some weights are large, certain directions of random noise will combine with these weights, and produce big differences in outputs, and this is a symptom of overfit data.

  Alternatively, we can use L_1-*regularisation*, and change our cost with a penalty depending on the first power of the weights:

  $$C = \text{cost} + \alpha \sum |\boldsymbol{w}|.$$

  Whereas L_2-regularisation tends to reduce weights of large size quickly, L_1-regularisation tends to reduce the weights evenly, so that all of them are pushed towards 0.

  There are plenty of other alterations one can make to the cost function in order to regularise the network.

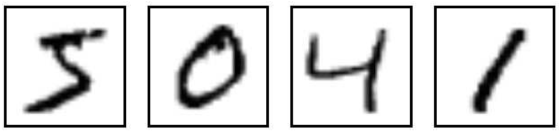
- *Dropout* is another technique for regularisation, of a rather different nature. Here, we don't modify the cost function, but instead we modify the network. Suppose we want to try and train a particular network, and want to do this in such a way that the weights are small. Temporarily, we deactivate a fraction of the hidden neurons in the model, and train the remaining parts of the network, with one step of mini-batch stochastic gradient descent in the forward direction, updating the weights and biases with backpropagation. We repeat this several times, each time dropping a different randomly chosen fraction of the neurons. Once the whole network is trained, we expect to have more of the hidden neurons active, since they have had to compensate for the temporary deactivation of other neurons. So we expect that in the full network, the weights will be too large—if we dropped half of the hidden neurons at each step, for example, we would expect that the weights in the network would all be about twice as large as they should be. So we reduce the values of all the weights to compensate.

So dropout is a way of ensuring that you don't end up with a situation where a few weights are large, and most are very small; instead, the weights are spread out more over the full network.

11.6 MNIST and CNNs

Let's turn to the specific example of the MNIST data set, considered also in the Lab sheet.

The data consists of 70000 handwritten digits, split into a training set of 60000 digits, and a test set of 10000 digits. Here are a few:



Each image is stored as a $28 \times 28 \times 1$ grid, with each pixel being given a value between 0 (white) and 255 (black). The goal is, unsurprisingly, to train a computer to recognise the digits, outputting a digit. So the input layer will contain 784 nodes, with each node corresponding to a pixel, and the input value would be between 0 (white) and 1 (black). After passing through the network, we want the output layer to tell us what the computer thinks the digit is. We therefore have 10 nodes in the output layer, corresponding to the digits 0 to 9. We would classify an image by selecting the output with the largest probability (this is "softmax" classification).

Remark 11.2 You may be wondering why images are stored with structure of $28 \times 28 \times 1$ rather than 28×28; the reason is simply that a colour image would need 3 values to specify the colour at each pixel location, so a colour image of the same size would be stored as $28 \times 28 \times 3$. For consistency, images are always stored with a third dimension. But we will see that we can make use of this in the convolutional layers below.

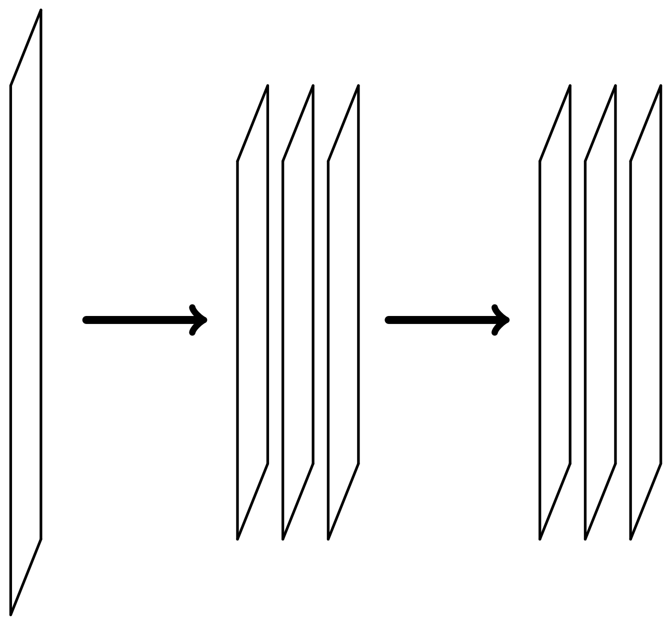
We have a lot of freedom in deciding the network architecture—how many hidden layers to use; how many nodes in each etc., as well as the function we use for the measurement of accuracy, method of (stochastic) gradient descent etc.

For the basic neural network introduced earlier, we want to convert the data set to an input vector. Since we have a 28×28 pixel image, this is converted to a vector of length 784. The Lab sheet considers how to train a network using `keras` (available in both `Python` and `R`), and it leads to a pretty good accuracy of around 98%. But a moment's thought will surely convince you that the process of taking the image and converting it to a vector is surely going to be losing useful information. For example, if pixel 1 is in the top left, then pixel 2 is adjacent to its right, so that pixel 28 is the final pixel in the top row, then pixel 29 will be the first pixel on the second row and so on—and our conversion to a vector is losing the information that pixels 1 and 29 are adjacent, and similarly for pixels 2 and 30 etc. In fact, if every input node is connected to every hidden node in the next layer, we are losing all information about the adjacency of the pixels in the image set.

Let's try to design a method which takes this additional information into account. Actually, our system will resemble the arrangement of neurons in the back of the eye, with these neurons connected only to a restricted number of the inputs. The thought is that the first hidden layer should pick up this highly localised information from the image; the next hidden layer should merge some of this information, again locally, patching together the local information from the first hidden layer into slightly larger patches, and so on until the whole image is processed.

Another advantage of this restriction will be that there are fewer weights in these networks, since it will not be the case that every node in one layer connects to every node in the next layer.

So rather than imagine the layers of the network as vectors, we try to retain the layers in the shape of 2-dimensional images (with a third dimension, as mentioned above):



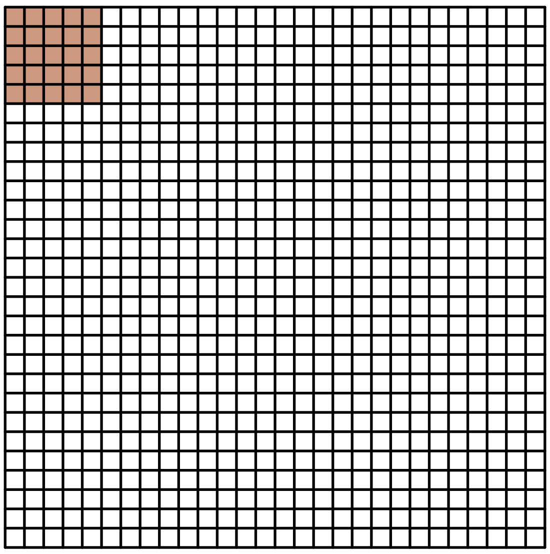
with each sheet in the diagram being a 2-dimensional arrangement of nodes, and the arrows between layers map patches of one layer to individual nodes in each sheet in the next layer.

Remark 11.3 Video processing involves a further dimension on top of images, corresponding to the frame number (essentially a time variable), so here all our layers would be processed with a 3-dimensional convolutional neural network.

Let’s summarise the layers which we use:

Convolutional layers

The first layer in a CNN is always a convolutional layer. It reads in successively small patches from the image and feeds the neurons in the patch to the first hidden layer. For example, the top left 5×5 patch might feed into the first neuron in the first hidden layer:



Then the second 5×5 patch would feed into the second neuron, and so on. With an input image of 28×28, and patches of 5×5, the first hidden layer will be 24×24, since this is the number of ways that a 5×5 patch can fit onto a 28×28 layer.

The precise map from the original layer to the next is the same for each location for the patch: with the 5×5 patch, the “filter” might simply return the average brightness for the 25 pixels in the patch as the output to the next layer. As the patch slides (or “convolves”) across the image, we get one output number for each location, which provides the content of the next layer. There are many alternative ways to taking simple averages of pixel values; filters can detect particular features by choosing particular values. For example, a 5×5 filter of the form

0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	1	0	0	0
1	0	0	0	0

would be more activated by inputs containing a diagonal line from bottom left to top right. So each convolutional layer will employ a number of filters of particular sizes. (In practice, we don't choose the filters, but the parameters are part of the training, using stochastic gradient descent, so we don't know at the start exactly which features are going to be used to identify the image.)

For example, we could use 32 filters of size 3×3 on the original image, to produce 32 numbers going to the first hidden node location. So the first hidden layer wouldn't be 26×26, but is instead $26 \times 26 \times 32$, with the 32 values produced by the filters on the top left 3×3 patch being stored in positions $(1, 1, 1)$ to $(1, 1, 32)$.

Pooling layers

We hope that our first hidden layer of size $26 \times 26 \times 32$ is detecting local features. We'd like to be piecing together local features to form more extended features, and we should be able to do this with further convolutional layers. But you can see that the sizes of the layers are growing quickly! So pooling layers are there partly to reduce the size of the data. For example, a 2×2 pooling layer will reduce the $26 \times 26 \times 32$ data set to a $13 \times 13 \times 32$ structure, by replacing each 2×2 square by a single number which is generally taken to be the maximum of the numbers in the 2×2 grid ("max pooling").

The idea here is that once we know that a feature is detected in approximately one place, the exact precise location doesn't matter, merely its presence at approximately this location. The further advantage of reducing the number of parameters will be valuable to avoid overfitting also, so this can be thought of as a form of regularisation. (One can still use dropout and other regularisation techniques mentioned earlier too, of course!)

Flattening

At some point, once the convolutional and pooling layers have done their job, the features in the layers have to be identified as an image. The output from the output layer will be a single vector of length 10, relating to the probability that the initial image can be classified as a particular digit. But what will the input to this final layer be? We need to "flatten" the output from the earlier layers, which will be 3-dimensional, into something 1-dimensional in order to use the earlier methods of classification, and the flattening layer is used to convert 3-dimensional data values into a single 1-dimensional feature vector with the information of which features are present at approximately which location. Then the network may have some further layers as in the earlier description of neural networks, to predict the class of the image from the feature vector.

Remark 11.4 Finally, we should mention another way to improve classification for the MNIST problem (or any problem in image recognition): we can expand our data set, slightly artificially. For example, we can take an image, and make a mild adjustment to it—perhaps a small rotation, or stretch in the vertical or horizontal direction, or (for certain digits) a reflection. All these will represent the same digit as the original image, but this provides us with many new images, and thus can enlarge the training set enormously. With more data on which to train, the point at which overfitting starts occurs after more training epochs, and the network learns more accurately. This has great potential to improve the classification accuracy of image recognition networks.

11.7 Other architectures

Above we have described the simplest sequential neural network, as well as those used for image recognition. Other specific tasks are often dealt with by other architectures. Just as image recognition is often done with a convolutional neural network (CNN), speech recognition is often done with a *long short-term memory network* (LSTM), and text-based problems are often addressed with a *recurrent neural network* (RNN). Just like our various clever tricks in Remark 11.4 for enlarging the data set for images, there may be techniques available here also; for example, we could distort speech samples by speeding up or slowing down, or adding random noise etc.

"Deep learning" problems may involve networks with several hidden layers. We have assumed that the information goes from left to right (a "feed-forward" network), but one could feed to a node on the same level, or even earlier, or more than one layer ahead. All these are considered in practice, and the choice of architecture is often quite complicated. It is a very active area of research in the machine learning field.

Neural networks are very interesting, and I recommend a lovely online textbook, Neural Networks and Deep Learning, by Michael Nielsen. (The computer code for the book is in `Python`, however, but it might be a nice exercise to convert it to `R`.)

12 Clustering

Aside from the earlier chapter on PCA, the previous chapters have concerned supervised learning, where the data has some response, either numerical in the case of regression problems, or categorical in the case of classification problems, and we are trying to understand how the response depends on the input data. In particular, can we predict the response of new observations, given their input variables?

Recall that unsupervised learning consists of the problems where there is no response, and we are trying instead to find relations between the data. We've already seen PCA, which is an example where we are trying to match the data with a lower dimensional space; another main example is "clustering", where we try to split the data into groups. So the machine is given a load of data, and attempts to separate the observations into groups. The statistical/data analytical terminology for this sort of problem is *cluster analysis*.

12.1 Hierarchical clustering

Hierarchical cluster analysis takes the n observations in p dimensions, and starts by trying to merge two points into a single cluster—the natural first choice here are the two points which are closest (using an appropriate definition of closest—Euclidean distance is a natural choice).

Having started with n distinct observations, we now have $n-1$ clusters; $n-2$ consist of single observations, but there is also 1 cluster consisting of a pair of observations.

We repeat this—and now we need to find which two of the clusters are closest.

It is easy to measure the distance between two observations once we decide on a distance function, but how do we measure the distance between an observation, and the cluster consisting of two observations?

More generally, as more and more clusters are formed by the merging process, we need to measure the distance between two clusters of observations.

There are many methods for doing this:

Single linkage the distance between two clusters is defined as the shortest distance between a point in one cluster and a point in the second cluster.

Complete linkage the distance between two clusters is defined as the furthest distance between a point in one cluster and a point in the second cluster.

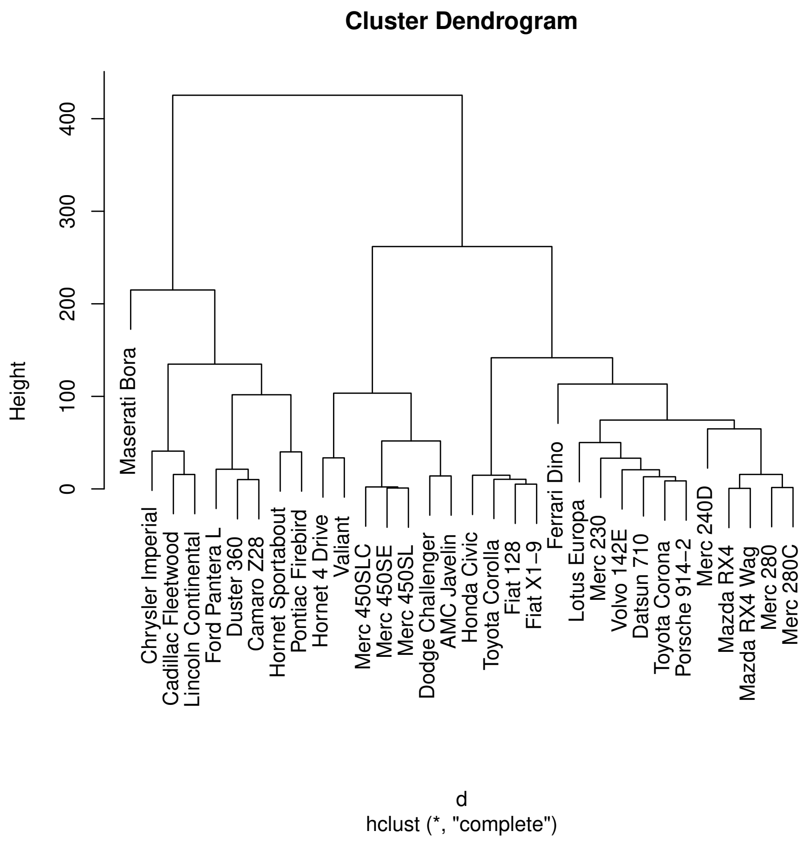
Average linkage the distance between two clusters is defined as the distance between their centroids (i.e., means).

We can try to minimise the sums of squares between centroids of clusters.

etc.

This method of merging clusters is known as *agglomerative* hierarchical clustering, and is the most common method. There is an alternative, where one can start with a single cluster, and split clusters into two where appropriate using these linkages. This is known as *divisive* hierarchical clustering.

We'll use the `mtcars` data set from `R` (see `mtcars.csv` on MOLE). Results are generally displayed on a *dendrogram*:



Going down the height axis, and drawing a horizontal line at any desired point, shows the clusters at that level. In the example, a height of 200 has 4 clusters, one of which is a single outlier while the other three are of similar sizes.

Hierarchical clustering is very unstable; small changes in the inputs can result in wildly differing dendrograms. This makes interpretation slightly tricky.

12.2 Non-hierarchical methods

Nowadays, hierarchical methods are regarded as less useful than the alternative *non-hierarchical* approaches to clustering by the machine learning community.

Centroid-based clustering specifies a number k of clusters in advance. Then the idea is to choose k cluster centres, and assign observations to the cluster with the nearest centre. This gives a clustering of the n objects into k clusters, which depends on the initial choice of cluster centres.

The problem comes in how to determine the best choices of cluster centres. We need to decide some way to score a choice of cluster centres—the most common algorithm here is *k-means clustering*, where the score is the sum of the squared distances between the data points and the nearest cluster centre—and try to choose the cluster centres so that this score is minimised. There are other measures of scoring choices of cluster centres, but k-means clustering is by far the most commonly used method in practice.

This is a hard optimisation problem, but there are iterative methods that converge reasonably fast to local optimum configurations:

1. Choose k cluster centres randomly.
2. Assign each data point to the cluster centre to which it is closest, and compute the total squared distances.
3. Replace the cluster centres by the centroids (averages) of the clusters to which they belong.
4. Repeat the last two steps (note that sometimes data points will now be assigned to different clusters).

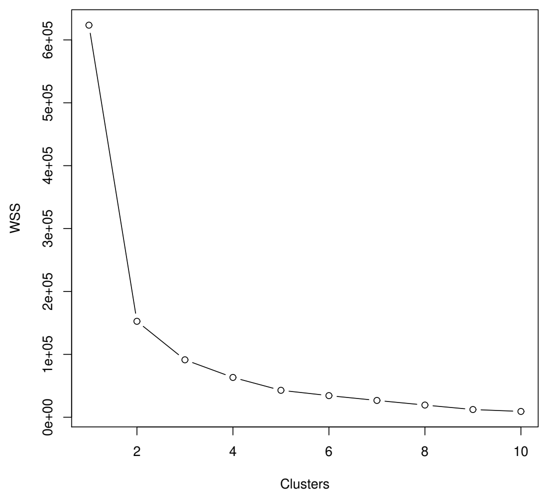
Eventually (quite quickly, usually), this converges, and we get a clustering of the n objects into k clusters, but this may depend on the initial choice of cluster centres; you may wish to run this several times, and choose the best.

The output of the algorithm consists of the best cluster centres, and then each data point is assigned to the group with the nearest cluster centre. To measure how successful this is, we can compute the percentage of the contribution of the total sum of squares coming from points in different clusters; we hope that distances within clusters will be small, so we hope that this percentage is large. In the case of the `mtcars` dataset, 4 clusters account for 88.1% of the total sum of squares (note that since different choices of initial centres may converge to different local optimum values, if you try this, you may well get a different answer).

Then there remains the question of which value of k we should use. We can repeat this method for each $k = 2, 3, \ldots$, and form a scree plot of the optimum total sum of squares against k, or alternatively the percentage of sum of squares coming from between clusters, and select an appropriate value of k by looking for an “elbow” in the plot.

All the methods try to minimise variance within the clusters and maximise variance between clusters. So the method resembles a kind of reverse ANOVA, in that we are trying to move observations around so as to get the most significant ANOVA results.

One way is to use the within clusters sum of squares, and make a scree plot. We should remember that, as already mentioned, running the algorithm several times gives different answers. So it is best to run the method a number of times and take the best solution. Then we can plot this against the number of clusters:



12.3 Density based clustering

It makes some sense to ask for clusters where there are a lot of points near each other. This is the idea of *density-based clustering*, of which a web search for DBSCAN and OPTICS may be useful.

DBSCAN (Density-based spatial clustering of applications with noise) works by selecting some threshold distance d, and a number k. We start with an arbitrary point, and consider all points within a distance of d from it. If there are at least k such points, the point is a *core point*, and a cluster is started containing all these points, and otherwise it is regarded as noise (for the moment). If other points in the cluster are also core points, then all of its nearby points also are added to the cluster. We repeat the process starting from any point not in a cluster until the clusters are stable.

The process results in a number of clusters, and then some points not belonging to any cluster, called *noise*.

OPTICS (Ordering points to identify the clustering structure) is fairly similar, in that we need the same two parameters d and k as DBSCAN (although d isn't that important this time—the process works by ordering the points by distance apart). It's a bit harder to describe, so I'll leave it for a web search.

12.4 Further reading

Wikipedia lists alternative methods. For example, some methods suppose some statistical distribution of each cluster, and try to measure how good an assignment of points to clusters is by using the Mahalanobis distance. You should look for "model-based clustering" in the literature. We won't say more about this, through lack of time, and because the details would take quite a while to discuss.

See also an excellent Stack Overflow answer.