Machine Learning for ecology, evolution, and conservation

Course notes and handout

Imperial College London

William D. Pearse

Department of Life Sciences
Imperial College London

March 2024

Preamble

By the end of this course, you will not really understand machine learning. It is impossible to learn machine learning in a week, and I'm not even sure that half of what I cover in this class is machine learning. So I'm sorry about all this: you've been misled.

What you will know by the end of the course is what a lot of terms that pop up in machine learning mean. You will know how to apply many useful machine learning tools, and how to go about figuring out how to use those tools to solve real-world problems.

The reason, however, that you still won't really understand machine learning is that there isn't really such a thing as "machine learning". Machine learning is really just a convenient term for a series of pseudo-statistical approaches that rarely have a basis in maximum likelihood, Bayesian statistics, or any other framework. They do, however, *tend to work extremely well when applied to the sort of problem they were designed for*.

If you pay attention at the start of this course, you will understand the "classical techniques" that can be applied to essentially any problem and almost certainly will work out alright. If you pay attention to the second section, you'll learn some very fancy tools that people *love* to use, and that often help you out if you apply them in the right way.

If you remember to start as simple as you can, try more complicated things *only if you need to*, and keep everything reproducible so you can quickly switch between methods (as has been emphasised throughout your course), you'll be alright. Just don't get too big in your boots: there is no such thing as machine learning, and so you're not an expert in it. You just happen to know a few things that, when they work, work really well. Which, in my opinion, isn't a bad thing to know at all. So maybe you were badly misled, but it hasn't ended up all that bad.

Table of contents

Chapter 1

High-level overview of concepts

Overview

This course provides an overview of what I view as the most important concepts in machine learning (ML) as used in ecology, evolution, and conservation today. It is not a catch-all bag for every technique, but rather an attempt to cover the fundamental concepts that form the 'classical canon' of methods that I personally think every practising biologist should be aware of (the first two sections) and then a very popular new approach (deep learning) that is tremendously powerful but quite a bit more complex. I strongly encourage you to view this course as a sort of pot-luck: you're going to be learning a lot of methods very quickly, so view this as a quick overview that you can dip into later for more details when applying approaches. Learn the critical concepts (described below) and how they are applied in each approach that I describe. The details of each method, while important, are things you can return to later: the general framework within which everything sits is the most important thing for you to learn (and something that is absent from most other resources you will find).

1.1 A historical note

To the lay audience, the explosion in application of machine learning approaches over the last decade is often viewed as unprecedented, terrifying, or transformative to the way we live and conduct science. Before going through, in practical terms, what machine learning is, I think it's important to engage with each of these three concepts.

Despite this section of this course covering what I am terming 'classical' methods, it is the only part of the course that contains anything that would have been unknown to a statistician in the 1980s. There is essentially nothing in artificial neural networks that we are covering in this course, and we are covering some very advanced topics that have powered a number of headlines, that is younger than I am. Thus the advances we are seeing in machine learning are not, in any sense, unprecedented: they are a consequence of Moore's law and the emergence of Graphical Processing Units (GPUs). Moore's law is a prediction (that has, broadly speaking, held until quite recently) made in 1965 that the number of transistors on a computer chip would double every year. The consequence of this is an explosion in the processing power—speed—with which we can carry out work. It is why deep learning methods that, in the 1980s, were essentially impossible to carry out, are possible today.

Indeed, it's why a lot of machine learning research moved on from deep learning in the 1980s, because it was quite clear we wouldn't be able to fit such methods for some time. Changes in computer memory (size and speed of access) have, of course, also helped a great deal—perhaps more than processor power increases more recently. GPUs, which were originally developed to power computer games, have turned out to be incredibly powerful tools for machine learning. GPUs are essentially stupid CPUs (Central Processing Units); glossing over a lot of technical details, whereas CPUs now often have two (or, in particularly powerful machines, 30) processing cores, GPUs in consumer laptops often have hundreds of cores (and powerful server machines tens of thousands). These GPUs have less memory per core, can't do fancy things like vector-instructions or quite standard things like follow independent lines of execution (*e.g.*, they can't execute an 'if' statement independently from other cores), but goodness are there a lot of them. GPUs are very good at doing the same calculation over and over again, and are wonderful for training artificial neural networks.

Machine learning came about because attempts at true Artificial Intelligence (AI) failed. In the 1970s there were reasons to suppose that we might truly be able to build a sentient computer within a few decades; barring the Internet being self-aware, I think it's fair to say we failed at that task. Scientists had the idea that focusing on problems in prediction and statistical learning—which they termed machine learning—might present a lateral move that might one day allow us to move back into true AI. The field you see today results from that lateral move, and as it has expanded it has somehow come to pass that a lot of machine learning methods that seem impressive are termed 'AI' in the media again. People trying to sell self-driving cars and hot-dog detectors liked the positive press, and so the old term stuck. There is nothing in a support vector machine that implies, in any sense, sentience or self-awareness. As you will see, supervised methods do have a sense of 'goals', but they are defined by us and not by the system itself. Artificial neural networks can be made to be 'Turing complete' but this does not change the fact that they are iteratively minimising the sum of squared errors. There is no meaningful sense in which such a network can become self-aware; they are transient extensions of linear models. Anyone who thinks otherwise is ignorant, and anyone who thinks otherwise and continues to fit one is a monster! But there is no need to fear your laptop has begun planning to take over the world.

One thing is for certain, however, and that is that machine learning methods are very powerful. There is, however, a distinction to be drawn between machine learning and big data: Twitter is exerting a huge influence on our lives because of the volume of data and number of people using it, not because of the algorithms it employs. The long-term implications of machine learning on society are likely to be very large indeed, but as with every other piece of technology from stone tools to nuclear reactors the terrifying and transformative aspects of it will be how we choose to use it.

In biology, I have noticed a tendency for researchers to treat machine learning as a black box that can solve a statistical problem. This is a profound waste of the power of ML: it is very difficult to derive general insights from the most complex ML methods, and it is often difficult to assess support for general mechanisms using them. Indeed, at worst it is literally impossible to understand how or why an ML method performs well. What ML is capable of doing, and where I think it may be truly transformative for the life sciences, is in automating tasks that up until recently required human intervention: tasks such as image processing, acoustic recognition, and literature review. Thus, in this class, I will be focusing on methods that are very good at certain classical statistical problems, but also those methods I view as transformative to how we conduct boring tasks in biology.

My aim is to give you an ML superpower: to free you from routine drudgery so that you can focus on biological questions and problems.

1.2 Machine Learning (ML) and algorithm definitions

There is a reason I gave you a historical overview of ML before actually defining it, and that is that there is no good single definition of ML that I have ever found. I define machine learning approaches as 'algorithms that are applied to data with the hope that they solve some particular problem, with no guarantee that they will generalise to other problems or situations'. Let's unpack that statement, and by the end of it you will hopefully have an intuitive understanding of what ML is, and thus why there will never be a good definition of it.

ML is all about algorithms. Algorithms are well (albeit variously) defined, but broadly they are sequences of computational steps that transform an input into an output. They have no guarantee of meaning; indeed, they are meaning free, and they differ from mathematical proofs in that the are procedural and cannot guarantee anything. Mathematical proofs can be written about algorithms, showing that they will always do certain things, but the algorithms themselves cannot guarantee anything. Algorithms have been developed to minimise the sum of the squared errors between predictions and underlying data in bivariate regression, and as a result there are linear regression algorithms and so linear regression is a machine learning algorithm. However, the derivation of linear regression (and its promised features, like passing through the means of its underlying data) are statistically proved, and not part of the algorithm. In other words, someone has *proved* that linear regression will do certain things—that its results will generalise to other situations—but the same does not have to be true of a machine learning algorithm. Thus all statistical methods are machine learning algorithms, because they are embedded within concepts such as probability density functions or Bayesian posteriors, but not all machine learning methods need be and so not all machine learning algorithms are statistical methods.

Now perhaps you see why I introduced the concepts of 'hope' and 'no guarantee' into my definition of ML. By leaving behind the requirement that we can prove something about probabilities or statistics and instead focusing on algorithms, ML frees us to explore weird and wacky things that we couldn't hope to explain to our friends in the mathematics department[1]. But it also means we are left with hopes and dreams and desires, because algorithms are, by definition, free of meaning. We can only hope that they work well, and we must come up with new ways (described in this course!) of assessing whether or not the work well.

And so perhaps you also see why there can never be a good definition of ML. Because ML is a field that is opening its arms, and saying 'come and have a go I'll try anything!' means that almost anything can be classified as ML. An 'if' statement in your first programming script could be described as ML if you really wanted it to, just as could generalised linear mixed effects models—all of them involve an algorithm, and all of them produce an output from an input.

[1] Note that many of your friends in the mathematics department are working on ML, because they hope to derive new algorithms from statistical and mathematical theory. If you don't have a friend in the mathematics department, get one: we could all use more friends, and in my experience maths departement parties are pretty crazy.

1.3 (Un)supervised methods, dimensionality reduction, and prediction

The distinction is often made in ML between *supervised* and *unsupervised* methods. A supervised method is on where you (metaphorically) tell the method what it needs to do given some input data. So, for example, you might have a set of 100 images and you have labelled those 100 as being of either a blackbird or a blue tit. If the ML method is to take the input (the images) and predict the output (the label—the bird species) then it is a supervised method. In an unsupervised method, you don't tell the algorithm whether it has found the right answer or not. So, for example, you might ask it to sort those same 100 images into categories, but you wouldn't tell it what you thought the categories should be. The way I prefer to think of it is that *supervised methods have response variables, and unsupervised methods do not*. Note that, in the example of categorisation I just gave, we might choose to ascribe meaning to the categories the algorithm has created. That's fine, but as that information doesn't feed back into the ML algorithm this is still an unsupervised process. People occasionally find ways to blur the lines between the two categories; good for them, but don't worry about such edge cases.

Unsupervised methods often involve some aspect of *dimensionality reduction*. You are familiar, by now, from your linear algebra about how matrices can have orders and thus datasets can have dimensions. To review, a table (matrix) might have ten columns; if it were a dataset, we might say it has ten variables and thus is a ten-dimensional dataset. Dimensionality reduction seeks to find ways of describing the majority of information within that dataset in lower dimensions: reducing or compressing in the information in those ten dimensions into two dimensions, for example. Sometimes such methods compress the information in columns of continuous numerical data, sometimes into categories. Most of the good methods for this are much older than the field of ML itself (notably principal components analysis).

Supervised methods are often used for *prediction*. Indeed, the supervised methods we'll be covering are analogues and complements to classical statistical techniques like regression that you are already familiar with. In more advanced techniques, and in particular those where we have left behind any hope of deriving or working with a likelihood for the model we are fitting, we must use prediction as our measure of goodness of fit. We often run our model to predict some known *training* data, and then assess its efficiency on some independent *test* data. We might then go ahead and run that model on even more data to solve our particular problem. This is important because advanced methods have a tendency to *overfit* data: they fit the training data so well that they cannot generalise to work with other data. To give an example of such a workflow, we might train our bird identifying model with 100 images, then test its predictive accuracy in 200 test images. If it performed well (let's say it got 98% of the test images right), we could then apply it to $10,000$ additional images about which we didn't know the species. If, however, the accuracy with the 100 training images was very high (98%) but very poor in the 200 test images (10%), we would say our model was overfitting the test data.

1.4 Variance (reduction) and bagging

If most of academia and life is about managing expectations, most of statistics is about managing variance. Sometimes we want to identify the sources of variance (*e.g.*, in ANOVA), sometimes we want to minimise unexplainable variation (*e.g.*, with careful experimental designs). In ML, variance is a problem because of overfitting: we want to minimise the ability of our models to predict variance that's unique to our training data. One way, somewhat paradoxically, is to *bootstrap* your training data: split it up into lots of smaller, random

subsets, and then fit your model to those subsets. In so doing you increase the variability in your training data because each subset is different. Because your model has to perform well across all of these random subsets, it paradoxically is less-prone to overfitting because the training process exposes it to more variation. Think of it as like taking the average of lots of groups: we're averaging out the variation within the training data, and thus allowing our model to see more of the 'big picture' variation within the data. We'll see the power of such approaches when we encounter regression tree models, whose ability to detect complex interactions among variables make them susceptible to overfitting without care and attention.

Such variance-managing techniques are also important for deep learning approaches, which are also very susceptible to overfitting. In such models it's often common to use *augmentation* approaches: to expand our training data by adding variance to it, and then running fitting our model to those augmented data. The idea is, once again, to minimise overfitting by introducing variance: in this case, that variance is (by design) erroneous, and so you're better preparing your model for the 'real world' where there is a lot more variation than is otherwise visible in the smaller subsets of training data you have available. What is surprising, to me, about such approaches is how well they work, even when the augmentation methods are very simplistic (*e.g.*, slight, random colour changes in images) and the data very complex (*e.g.*, images of birds).

In what is an emerging and recurring theme of ML, these approaches are neither unique to ML nor were they invented for ML. Bootstrapping and augmentation are approaches to manage variance in other, classical statistical settings. So don't view these tools as purely ML tools, but rather statistical tools.

1.5 How to choose the right tool for the job

I hate statistical flow-charts because, unlike taxonomic identification, there is rarely a single, unambiguous 'correct' statistical approach for a particular problem (that isn't how statistics works). Regardless, when picking a problem, ask yourself first: do I have a response variable? If so, then you want a supervised approach (*e.g.*, SVM). For supervised problems, ask yourself if you have a lot of explanatory variables (*i.e.*, more than 20). If so, then you want a lasso or least-angle regression. If you have a very complex set of potentially interacting factors, then you probably want some form of regression tree or an artificial neural network. Otherwise, you probably want something else: *and do bear in mind that thing you want is likely a classical statistical test*. If you don't have a response variable and you want to simplify your data somehow, you likely want an unsupervised approach. If all your data are continuous, use a principal components analysis, otherwise use a principal coordinate analysis. If you want to split your data into categories, use some kind of clustering algorithm.

The huge variety of supervised machine learning algorithms—approaches that require response variables—is often overlooked because students are so desperate to learn about artificial neural networks. This is a shame, because the ML toolkit contains a number of extremely powerful tools that are capable of solving a number of very common problems. In this section we are going to cover classical tools that are in many ways the ML complement of Generalised Linear Models (GLM). These approaches are capable of either categorising data (*e.g.*, true/false, red/blue/green) or predicting continuous variables. They set themselves apart from approaches like GLM in terms of their flexibility; some are capable of dealing with overwhelmingly large number of potential explanatory variables (*e.g.*, lasso regression), managing complex and multi-faceted interactions naturally (*e.g.*, regression trees), or are so flexible that they can handle almost any classification problem (*e.g.*,

SVMs). Critically and importantly, even when these approaches work with categories, they are *supervised* approaches, and so you are required to give the aclgorithms exemplars of the different categories for which you are building a model. Whereas a clustering algorithm will choose the identity (and potentially number) of groups for you, in supervised methods you must give data on cluster identity.

1.6 How to approach this course, and what you need to know for the exam

Your time is valuable, and I want you to learn as much as you can from this course. Thus I have deliberately fit vastly too much into this handout and course, so that those of you who are familiar with some of these approaches already can go beyond your current knowledge. Equally, those for you for whom this is all new know that you have this handout as a resource to come back to in the future. Equally, I've set you many, many more questions than I expect you to be able to cover in your afternoon sessions. That way you can pick and choose the questions that seem the most challenging to you in order to learn the most. *Don't be surprised or frustrated if you can't cover all this material—I've designed it to be like that so that I can teach you as much as possible*.

So it should be reassuring to you to know that you do not need to learn everything in this course for the exam. The topics you do need to cover for the multiple-choice exam (there are no essay questions on this material) are: PCA, PCoA, hierarchical clustering, k-means analysis, LAR (and lasso) regression, SVM, regression trees, and artificial neural networks. The material in the symposium is not examined. Of course, I would strongly encourage you to try and learn all the material in this handout, but please remember I'm not requiring you to do so. So please don't freak out that there's too much to learn here: I'm giving you lots of opportunities, not challenges.

1.7 Further reading

For the classical methods, there is no better source than 'The Elements of Statistical Learning' Hastie, Tibshirani, and Friedman; 2008 Springer. Amazingly, the authors have also released the book online for free [2] and so you can download it guilt-free right now. It is an intimidating book, and so if you want a slightly gentler introduction (and I would not blame you!) many of the same authors have written 'An Introduction to Statistical Learning' (James, Witten, Hastie, and Tibshirani; 2013 Springer with a new edition coming soon) which has chapters that are directly analogous to its parent book such that you can flip between them at will (perhaps leaning on the latter for more *R* examples). This second book is also released online for free [3].

Artificial neural networks are covered very nicely in the above books, but they don't go into as much detail about deep learning as might be ideal. There are very few good books on deep learning, in part because the field is expanding so rapidly, but also because there are so many expanding software packages and most resources focus on those rather than the actual underlying concepts. With those pieces of software changing so rapidly, it can be difficult to make headway. 'Deep Learning (Adaptive Computation and Machine Learning Series)'

[2] https://web.stanford.edu/~hastie/ElemStatLearn/
[3] https://www.statlearning.com/

(Goodfellow et al. 2017 MIT Press) is very popular, and is notable for covering a lot of concepts in linear algebra (which you now already know), but I personally find it to be very long and very unclear. I much prefer 'Neural Networks and Deep Learning' (Charu Aggarawal; 2018 Springer); it's shorter, denser, and covers more (I think) than other books but is up-to-date and is much more intelligible. Personally, I think there are enough tutorials of how to use *TensorFlow* to be found online so your focus should be on concepts in books. The tutorials and examples on the website of the *keras* website are brilliant and I think are superior to most other articles I've seen online; I hate learning from YouTube so I can't recommend any videos I'm afraid.

In terms of practical examples, your symposium on Wednesday should give you lots of inspiration for applications and uses of these tools, as well as papers to read with examples of their use. If you want other resources, I strongly recommend 'Machine Learning for Ecology and Sustainable Natural Resource Management' (Eds. Humphries, Magness, and Huettmann; 2018 Springer) which you have available in the library. It's an edited volume and has lots of examples; they're obviously mixed in terms of quality, but broadly are good. 'Methods in Ecology & Evolution' (December 2020) also had a fantastic special issue recently on ML, which outlines techniques and their uses [4].

[4] https://besjournals.onlinelibrary.wiley.com/doi/toc/10.1111/(ISSN)2041-210X.machine-learning-vi

Part I

Classical methods

Chapter 2

Dimensionality reduction—PCA, PCoA, and NMDS

Overview

When faced with high-dimensional, continuous data—lots of columns of numbers in a spreadsheet—one of our first thoughts is often how we can focus on the most important dimensions (columns) of data. But if it's not obvious which column is the most important, perhaps because they are all very similar to one-another, an alternative is to try and compress those many dimensions into fewer dimensions. This compression of data is sort of like finding the fundamental, underlying axes of variation that drive these data: the principal components of variation in the dataset. The three approaches you will learn in this section—PCA, PCoA, and NMDS—are all ways of carrying out such work. A major concern with such approaches is to make sure that you don't throw the baby out with the bathwater: there is always a loss of information and so you must make sure that the information you've retained is still valuable for what you're trying to understand. You must also be careful, when using these approaches, not to fall in love with them: *never mistake the opaque for the profound*. Ask yourself constantly and unceasingly what everything you've just calculated means. Sure, you've compressed 100 columns of data down into 2 columns, but what do those two columns represent? If you don't know, then you can't do anything meaningful with them. Check that your results make basic, logical sense in the context of what you know about the biology of the system you're working with. If it does: congratulations! You might have done it right. Remember that the best way of reducing dimensionality is not with statistics, but with scientific insight. There's no need to compress hundreds of columns of data if, from first principles, you know which column is the most important one anyway. Beware: these approaches assume that all axes of data are equally important, and so if only one of your columns of data was important in the biology of your system, then there is no guarantee that they will highlight that particular axis.

2.1 Principal Components Analysis—PCA

Principal Components Analysis (PCA) is a way of taking many columns of data and rotating them so as to find their principal axes of variation. It never changes the data, but rather highlights the main ways in which they

co-vary. PCA is exactly the same as a major axis regression. PCA loses no information (*i.e.*, it performs no true compression, only highlights correlations), but if you choose to work with only some of its results axes then you explicitly ignore the variation in those axes you do not choose.

2.1.1 An intuitive grasp of PCA

In a standard linear regression, you take a response variable (y), plot it against an explanatory variable (x), and find the line through the two that explains the most variation in y (*i.e.*, that minimises the residual variance in y). Not simple at all, but something you're familiar with. From this line, you're able to get information about the relationship between the two variables (the slope of that line), and you're able to get residuals—the distance of each y value from the line and a measure of the error in your prediction. You may recall that you can do the same thing with more than one explanatory variable, in which case the logic is much the same; you're trying to find a line that minimises the distance between each y value and the line. If I'm describing this in unfamiliar terms, take a quick look at figure 2.1.

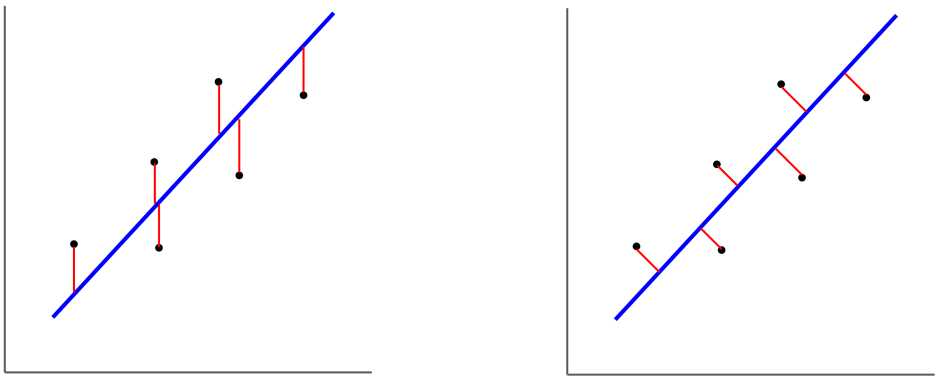
Imagine now that this line you're drawing isn't the line that minimises the error in y (the distance between the line and each y), but instead is a line that minimises the total error in both y and x. If you're a visual person, you would agree from figure 2.1 that such a line would minimise the length of the line in red that are at 90° to the line. This sort of regression has lots of names, and you might have heard of it as a 'major axis regression', a Deming regression, or ... you may not have heard of it. But hopefully it makes a degree of intuitive sense.

If so, congratulations. You now understand a *Principal Components Analysis*, or PCA, because that is mathematically and precisely exactly what it is. At the end of this lecture you will go and tell your friends this, and someone will tell you that "no, he must be simplifying": I'm not, this is all there is to it. In the case of two-dimensional data (y and x are two variables, so there are two dimensions), the *first axis* (or *first principal component*) of a PCA for each data point is how far along each piece of data ((x,y) pair) is along that line. The second axis is how far that data point is from the line itself (a distance that is at 90° to the line—along the red lines in figure 2.1). A PCA is calculated by trying to find the line that minimises the total variance of that second axis—or, put in a more familiar way, trying to draw the line that minimises the overall error of that line.

In the case that we had more than two variables (dimensions), that line would be still be drawn such that it would minimise the error through the entire dataset. The first axis would be defined in exactly the same way—minimising the error. The second axis would be at right angles—*orthogonal*—to that first axis (just as in the two dimensional case), and would extend in whatever direction the most variation remained in the data. The third, and final, axis, would be *orthogonal* to both the first and second axes, and would soak up all the remaining variation. This gives an intuitive reason for two important concepts in PCA: all the axes are *orthogonal* [1] and all of the variation that was in your original data is contained in the new PCA axes.

We're really just *rotating* the data in a fancy way so they match onto this line we've drawn, and some people prefer to describe PCA as a kind of rotation. In this case, you're taking the x and y variables—the dimensions of the data— and then rotating around so we have a new first axis (the first principal component) and then a second orthogonal to that. This rotation could also happen for as many variables, or dimensions, as you have.

[1] I will stop putting this word in italics now, but you must start using it when talking about PCA. Replace it with 'at right angles to everything else' in your head if it helps.



(a) Linear regression (b) PCA

Fig. 2.1 **A graphical overview of Principal Component Analysis (PCA).** In a classic regression (a), we try to find a line that minimises the error in y. In a PCA (b), we try to find a line that minimises the overall error in all of our variables: in this case, y and x. Once we've drawn that line, our first PC axis is the position of each data-point along the line (the centre of the line is '0', and each data-point's value is wherever its red line touches the line), and the second PC axis is the length of the red line (positive values for above the line, negative for below; so it's not really a length but you get the idea). If we were doing this in three dimensions, we could make the third PC axis by measuring distance from the line along that third dimension—depth on the page.

Some people find thinking of this as a rotation more natural; if that's you, then great, but if it's confusing then focus on the error minimising argument (which is actually more general, and is more useful later in the course).

PCA is really useful, because it means we can take big datasets where everything is correlated with everything else, and collapse that variation down into a number of axes that are all independent—orthogonal—of each other. Because the amount of variance associated with each axis isn't the same, we can pick fewer numbers of axes that explain "enough" of our data (more on that later), and analyse those. Since we're just *rotating* the axes of our real data onto these new axes, we can also figure out how each PCA axis *loads* onto the original axes. Loading just means "how much of the variation in a given piece of data is associated with a particular PCA axis". So a variable that loads strongly and positively onto PCA axis 1 is positively correlated with that new axis—the PCA "line" goes through it with a positive slope. A weak negative loading would be a shallow curve with a negative slope. Bear in mind that the sign of the loadings is essentially arbitrary: they only make sense relative to one-another. Thus it's not important that x is negatively loaded only PC1, for example, but it would be important if x were negatively loaded and z were positively loaded, because it implies that the contributions of x and z to the axis are negatively correlated.

2.1.2 Hands on with PCA

OK, let's simulate some data that we can use to figure out how PCA works. This is probably the scariest part of today's practical, because we're going to be drawing data from a multivariate Normal distribution. In practice, what this means is we're going to draw five variables at random: three of them are going to be strongly correlated with one-another, and the other two are going to vary independently. Take a look at the covariance matrix (`covariance`): each row and column refers to one of the variables we are going to be randomly drawing from a Normal distribution, and represents the covariance of the two variables. So, for example, row 1 column 2 (3) means "variables 1 and 2 will have a covariance of 3". Row 1 column 1 (5) means "this variable will have a variance of 5"—covariance with yourself is really just your overall random variation. In cases where the

covariance is 0, that means those two variables won't correlate with each other—they're independently drawn. The line containing `set.seed` is just to make sure that the random numbers you generate are the same as the ones I generate (don't worry about how this works)—it means I can talk about the PCA you've generated and be certain that it's the same as the one I'm working with.[2]

Notice how I load an *R* library called `mvtnorm` using the line `library(mvtnorm)` at the beginning of this script. If *R* gives you an error when running this, that means you don't have that library installed: you can either install it using *RStudio*'s menus, or you can run the line `install.packages("mvtnorm")`. Either is fine, but you'll then have to load the library (installing doesn't load something in *R*, just like other programs on your computer). I'm not going to remind you to do that in the future: I will just assume that, if you can't load a library, you will install it.

```
library(mvtnorm)
set.seed(123)
covariance <- matrix(c(5,3,0,-3,0, 3,5,0,-3,0, 0,0,5,0,0,
  -3,-3,0,6,0, 0,0,0,0,3), nrow=5)
data <- rmvnorm(1000, sigma=covariance)
names(data) <- c("a", "b", "c", "d", "e")
```

Now we've got this data, we can run a *principal components analysis* (PCA) on it. The *R* code is deceptively simple: the first line below runs the PCA, and the second produces what's called a *biplot*. In it we see each point in our data (each data point is labelled according to its row in the data), with arrows that show how each variable (*a*, *b*, *c*, *d*, and *e*) *loads* onto each axis. So you can see that *d* is negatively correlated with *a* and *b*, because they're both in the same 'plane' (*i.e.*, PC1) but on opposite ends. You can also see that *c* is floating out on its own, *orthogonal* to everything else, because it was drawn independently from the others. If you were to look at PC axes 2 and 3 (which you can do using the `argument`, see below) you would see *e* floating around independently of all the other axes too. It's very hard to even see *e* in this particular plot, because it *doesn't load* onto either PC1 or PC2.

```
pca <- prcomp(data)
biplot(pca)
biplot(pca, choices=2:3)
```

How can we see these 'loadings' that I keep talking about? Quite easily: you just print out the PCA as shown below. This gives you the *rotations* of the original data (rows are original data, columns are PC axes), which shows you how strongly each variable *loads* onto each principal component. So *a*, *b*, and *d* all load strongly onto PC1 (in opposite directions, hence the sign changes), while *c* and *e* are doing their own things on PCs 2 and 3 (notice how big their loadings are—the maximum possible is 1, and they're dominating the values on those axes). If you want to prove to yourself that, really, PC2 is essentially exactly the same as *c* then run the second line below, which plots *c* against PC2. Remember: loadings vary from -1 to 1, where -1 means a very strong (perfect) negative correlation between a PC axis and the variable, 1 means a perfect positive correlation, and 0 means no correlation whatsoever. Think of them as correlations and you'll do fine.

[2] Before doing any serious analysis in *R* after this practical, you should close down *R* and start it up again to make sure you get rid of any weirdness my use of `set.seed` would have had. Ask me for more details about how all this works if you're interested.

```
pca
plot(pca$x[,2] ~ data[,3], xlab="'c' variable", ylab="PC2")
```

PCA is meant to be a kind of *dimension reduction* technique: we're trying to take a load of correlated data and reduce it down to a smaller set of independent, or *orthogonal*, principal components. So two critical questions are "how important is each axis" and "how many axes are enough for me to reduce everything down to"? PCA is pretty good at giving us an answer to the first question, but sort of rubbish at the second. In PCA-land, an axis is important if it explains a lot of the variance (variation) in the original data. If we call `summary` on our PCA object, we can find out how much standard deviation (the square root of variance) is associated with each axis, which sort of gets at that first question. A bigger SD is a more important axis. As for how much is enough... Well, most people plot a *scree plot* (the second line below) and find where there's an inflection point in the standard deviations: you find the axes that you can join in a single straight line, and work with those axes plus the next one along. So, in this case, PCs 1 and 2 are on a straight line, PC3 isn't, so we'd go with PCs 1–3. There is a citation associated with this[3], but there's little more justification to this idea than "it seems a good idea". I don't have a definitive answer for you as to how to pick your number of PCA axes, but I will give you two pieces of advice: (1) just because an axis doesn't have a lot of variance associated with it doesn't mean it's not important. The amount of variance is related to all the variables you put into the analysis: if only one of your variables actually mattered to the biology of your system, a PCA isn't going to help you. The whole point of PCA is that each variable *could* be as important as the other, and so if you've put a load of crap into your PCA then you're probably going to get a load of crap out and miss the important stuff. (2) As Mick Crawley infamously advises: "*never mistake the opaque for the profound*". If you don't know what a PCA axis represents biologically, it's essentially useless to you. It's useful to have a single variable that captures how hot and dry a particular region of environmental space is; it's not useful to have a single variable that maps onto thirty different climate variables, none of which you understand. User beware...

```
summary(pca)
plot(pca)
```

Finally, PCAs are quite sensitive to variance among different axes. This isn't so much of a problem in this example, but because the whole thing is based around finding axes that explain the most variance, if the variance isn't equal across your variables you're going to get skewed results. This is essentially a corollary of piece-of-advice 1 above: if everything's meant to be equally important, they should all vary equally, or the thing that varies the most will be 'inflated' in importance. There's a simple solution: conduct your PCA on z-transformed variables (scaled to have a standard deviation of 1), and the option for that is simple to use (see below). **If you don't use a scaled version of your variables you will make a mistake: this isn't an opinion thing, it's a fact thing. Scale your variables.**

```
pca <- prcomp(data, scale=TRUE)
```

[3] Jackson, D. A. (1993). *Stopping rules in principal components analysis: a comparison of heuristical and statistical approaches*. Ecology, 74(8), 2204-2214. It's a good paper; if you're interested, read it.

2.2 Principle Co-ordinate Analysis (PCoA) and Non-Metric Multidimensional Scaling (NMDS)

PCoA is a close relative of PCA; whereas PCA works with the underlying data, PCoA works on a *distance matrix* of the data. This allows PCoA to be applicable to any kind of data you can create a distance matrix for (including categorical data), and in the case that you're working with the covariance matrix of such data you get exactly the same result as if you'd computed a PCA on that data. PCoA projects a distance matrix onto a Euclidean space of as many dimensions as you specify: it's up to you to see whether you're summarised that distance matrix well, and whether it's even possible to approximate the distance matrix well in Euclidean space. Non-metric Multi-Dimensional Scaling (NMDS) is a popular close-cousin, that differs from PCoA in being able to work with the ranks of distances among data-points. The mathematics of it are, in many ways, simpler, but getting it right is much harder. As always, the choice is yours...

2.2.1 Under pressure: an intuitive feeling for PCoA and NMDS

Principal Co-ordinate Analysis (PCoA) is a method that takes a distance matrix and projects it into Euclidean space. Euclidean space just means "distances among objects work in the way that they do in the real world": there are lots of ways mathematicians have invented of representing objects in the world that don't make intuitive sense (*e.g.*, space itself is curved, not straight). So PCoA takes a distance matrix and finds a way of arranging the points that each element of the distance matrix represents so that they can be plotted. If they can be plotted in two dimensions, great, but the data could need as many as three, four, five, six... It proceeds in three steps:

1. Centre the distance matrix. This means that, when we map everything into Euclidean space, the centroid of the data will be at the origin ($(0,0)$) of a graph at the end.
2. Find the *eigenvalues* and *eigenvectors* of the distance matrix. I will discuss this step in more detail below.
3. Scale everything according to the square root of the *eigenvalue*. This is important, because anything with a negative *eigenvalue* can't be represented in Euclidean space (see below), and anything with an *eigenvalue* of zero doesn't matter (see below). So the purpose of this step is simply to figure out what what matters.
4. Finished! Each row of the resulting matrix is a data-point, and each column the position in space where that point should be plotted.

Bad news: to do PCoA we really do have to understand what *eigenvectors* and *eigenvalues* are. Well, they're the things we were calculating in our PCA: the eigenvectors are regression equations through our data that are *orthogonal to each other*. The *eigenvalues* are how much variation in our original data is associated with each of these equations—they're identical to the standard deviations we worked with during the PCA session. In an ecological context, you can map different environmental variables onto these axes as well, in much the same way that you can look at loadings of variables in a PCA.

The magic of PCoA is that some of these *eigenvalues* become zero when we run all of this through. That's really cool, because it means the whole problem of figuring out how many axes are "significant" is done for us:

all of them are "significant"[4], and so that's that[5]. There is, however, a major catch: not all of the variation in the original data can be mapped onto Euclidean space, and there is nothing PCoA can do about that. There are lots of ways of measuring the extent of that missing variation, and they all have names like *stress* or *strain*, because they reflect the extent to which your data are resisting being mapped onto this space.

If this loss of information is something that really worries you, it's possible to use something called *Non-Metric Multidimensional Scaling* (NMDS; sometimes just called MDS), which sacrifices statistical pleasantry for guaranteeing that it can compress the information in your data into a set number of dimensions. The problem with NMDS is that it requires you to pick a number of dimensions onto which you will project your data, and how do you pick the number of dimensions? Well, there's no single answer, but you have to check the *strain* that your data is put under when you map them onto a set number of dimensions, and there's no clear way to pick that. If this doesn't sound like a solution, that's because (in my opinion), frankly it's not. Either way, you have to decide how much variation you're happy throwing out. Happily, there is some light at the end of the tunnel: *quantile regression* lets you work with the original distance matrix itself for statistical analysis, and so all of these problems become purely visualisation issues.

2.2.2 What's an extra 'o' among friends? Hands on with PCoA

OK, let's get our hands dirty with PCoA to get an intuitive feel for what's going on. As ever, we're going to simulate some data, but this time it's going to be some community ecological data: compositions of species across two environmental gradients. This is the most complex set of data I'm going to simulate for you in this class: we're going to simulate intercepts and slopes of species' responses to two environmental gradients, then simulate their abundances across those gradients. If it doesn't make much sense, just treat it as magic: the details don't matter, just take the story for granted. There are simpler ways to simulate data such as this, but if you're interested in the distribution of diversity you might find this a useful starting point for your own investigations.

```
# Draw species' parameters
intercepts <- rnorm(20, mean=20)
env1 <- rnorm(20)
env2 <- rnorm(20)

# Create environment
env <- expand.grid(env1=seq(-3,3,.5), env2=seq(-3,3,.5))
biomass <- matrix(ncol=20, nrow=nrow(env))
for(i in seq_len(nrow(biomass)))
    biomass[i,] <- intercepts + env1*env[i,1] + env2*env[i,2]
```

OK, so we've got a lot of data. Notice these are biomasses, thus we've got lots of continuous data[6]. Let's summarise it using a PCoA:

[4] Some, of course, are more significant than others...

[5] As my other footnote indicates, in reality that's not that but hey ho.

[6] If you're interested in extending this to work with abundance data, talk to me. It's quite fun to do so, but would add extra complexity I don't want to bother the class as a whole with.

```
library(vegan)
dist <- dist(biomass)
pcoa <- cmdscale(dist, eig=TRUE)
barplot(pcoa$eig)
```

The second line creates a distance matrix, as you know. If you're an ecologist, you're probably already thinking that it would be a bad idea to work with a Euclidean distance, in which case `vegan`'s `vegdist` has got you covered. The third line runs the PCoA, and by asking it to return the `eigenvector` information we're guaranteeing that we get all the PCoA axes and lots of additional information. The fourth line plots the *eigenvalues* of each axis: how much variation they explain in the space. I hope you would agree with me that the first two axes explain the overwhelming majority of information in this space, so we're fine working with just those. Note that the "scree test" I taught you for PCA does not apply here: that test is based around expectations of eigenvalues under various null models for PCAs, not for PCoAs.

```
plot(pcoa$points[,1:2], xlab="PCoA1", ylab="PCoA2")
plot(pcoa$points[,1:2], type="n", xlab="PCoA1", ylab="PCoA2")
text(pcoa$points[,1:2]+.25, labels=env[,1], col="red")
text(pcoa$points[,1:2]-.25, labels=env[,2], col="black")
```

The code above shows how you can plot the various axes (`points`) of the PCoA. The first line shows how you would plot each site's position in the two-dimensional space that we have mapped our distance matrix onto; the next three show the values of each site in the first (in red) and second (in black) environmental axes. All I want you to take from this is that we simulated data that varied across two axes, and as a result our PCoA has found two axes across which it can distribute the data.

I'm not going to dwell on two things that are normally mentioned in screeds on PCoA at this point: *stress* and *loadings*. Stress, which you'll recall is how well our distance matrix can be mapped onto a certain number of axes, is manifest in PCoA through negative eigenvalues[7]. If you want to see an example where there is some stress in a plot, run `example(cmdscale)` in *R*. Loadings give you what biplots gave us when we looked at PCAs: they show how variables (*e.g.*, environmental conditions) load onto each of the dimensions the PCoA has estimated. Remember that calculating the PCoA of a distance matrix is the same as running a PCA on the covariances of the same data, and so if you understand biplots in PCA you essentially understand it in PCoA. I describe, in the section on *quantile regression*, why I think using this sort of thing with a PCoA is a bad idea, however, and so I am not going to spend any more time explaining something I don't really think you should be doing[8].

2.2.3 What's an extra 'n' among friends? Hands on with (N)MDS

You don't need to have an analytical solution for how to find the Euclidean space onto which a given distance matrix should be mapped. You can, instead, keep moving points around until you find an arrangement where the distance between points in that space is sufficiently close to the distance in the real data. That difference,

[7] A negative eigenvalue means there's no real (in the mathematical sense) solution to the regression equation you're trying to fit. Thus their presence denotes non-Euclidean variation (variation we can't find a way to map), and the size of the eigenvalue denotes its importance.

[8] My class, my rules.

when scaled in a few different ways, is either called *stress* or *strain* depending on the precise method you're using. The result is quite dependent on the algorithm used to find a good arrangement of points in space, and (you guessed it) there's no single way to know how many dimensions you should map the data onto or what algorithm to use. With that in mind... Let's give this method, which is called NMDS, a try.

```
nmds <- metaMDS(dist)
plot(nmds)
stressplot(nmds)
```

No great surprise, the answer is essentially the same as before. You will likely get a warning that the stress is very low: this is because these data were simulated to be very well-behaved, and is nothing to worry about. The first two lines are reasonably self-explanatory, but the third line is a *Shepard diagram*, and can be used to diagnose how well your model has fit. You ideally want a straight 1:1 line, and if you can't get that then you want the red dots to be close to the blue line. The line is broken up into steps because during NMDS fitting the distances are monotonically transformed to maximise the relationship between the original distances and the new distances. This isn't a parametric fit—hence the line is broken, as we don't know what shape it should be so this is the most conservative—and the line can't decrease—it's a monotonic fit.

```
plot(nmds)
orditorp(nmds, display="sites")
ordisurf(nmds, env[,1])
```

I've already described that I don't like mapping environmental responses onto these kinds of plots, but the code above will do it for you. All I can say, repeatedly, is that you should be tremendously careful about fitting a smoothed regression to transformed-distance data, because there are so many steps and so many ways for error to creep in (how much variation loads onto each NMDS axis? How well does the smoothed regression fit? etc.). Regardless, above is how you would do it. It's also possible to fit convex hulls around groups of points that you think correspond to particular treatments: I refuse to show you how to do this because they're so over-used at conferences, but there are plenty of guides online[9]

2.3 Coda: quantile regression

NMDS and PCoA are fantastic ways to summarise data, but because you lose information in the process I don't think they're good foundations for analysis. I am not as militant about this as I have probably led you to believe over the last few pages, and every analytical tool has its place, but overall I prefer to work with the original distance matrix itself when performing statistics—particularly when the questions related to actual distance matrices themselves. I think this is particular important for analyses of β-diversity, and while NMDS is commonly used to understand how microbial sequencing data varies across time and space, I am not a fan of using those NMDS axes of a distance matrix in analysis. So what do I like doing instead?

```
mantel(dist, dist(env[,1]))
mantel(dist, dist(env))
```

[9] If you must learn it, Jonathan Lefcheck is sufficiently smart that it might as well be from him: https://jonlefcheck.net/2012/10/24/nmds-tutorial-in-r/

A *Mantel test* is the equivalent of a correlation coefficient for (distance) matrices. There are lots of caveats to its use (read Legendre *et al.*, 2015; Methods in Ecology and Evolution, 6(11), 1239-1247), but if you understand a correlation coefficient then you intuitively understand it. It is rare that you will go wrong with it. In the first line above, I run a Mantel test on a single variable, and in the second I run it on a distance matrix constructed from all the environmental data merged into one matrix. There are lots of version of Mantel tests: I only use one, the basic form of it, and it is perfect for when you want to see if two distance matrices are correlated. I would advise you to ignore things like 'partial' Mantel tests, if you come across them, and instead use quantile regressions (see below).

```
library(quantreg)
data <- data.frame(dist=as.numeric(dist),
  env1=as.numeric(dist(env[,1])), env2=as.numeric(dist(env[,2])))
model <- rq(dist ~ env1 + env2, data=data)
summary(model)
```

If you have more than one variable and you want to know the influence of each on your variable of interest, use a *quantile regression*. A quantile regression is exactly like a standard regression, only it works on the quantiles of your data—by default, the *median*. Magically, as long as the pseudo-replication in your data is equal, all its estimates are trustworthy: this is math-speak for saying "it's fine to use with distance matrices". If you have more than one explanatory variable, this is the method you should use to understand what is driving pattern in your data.

```
complex.model <- rq(dist ~ env1 + env2, data=data, tau=c(.2,.5,.8))
summary(complex.model)
```

Quantile regression can be run on quantiles other than the median, by varying the τ parameter ('tau'), as I show you above. This makes them particularly cool if you think things are going on in the extremes of your data: I use them all the time, and find them to be very useful in ecology in general, not just distance matrices[10].

Let me try to convince you, one last time, to not do what everyone else does and draw circles around groups of species or sites in an NMDS/PCoA plot and write stories about that. PCoA and NMDS are ways of *visualising* complex data, and as they simplify it they may remove signal from that data. Any relationships or groupings you draw on a PCoA or NMDS plot come from the distance matrix of the data: any story you can tell about those groupings is fundamentally a property of the distances, which you have transformed. Quantile regression works with those original distance matrices, and gives you regression plots based around those original values. Why use something that (1) throws away variation in your data and (2) relies on a transformation of your data into another form that doesn't represent it? Why not, instead, just plot your data?

```
simple.eh <- rq(dist ~ env1, data=data)
with(data, plot(dist ~ env1, pch=20,
  xlab="Environmental distance", ylab="Site similarity"))
abline(simple.eh, col="red", lwd=3)
```

[10] e.g., Pearse *et al.* (2013). Ecology, 94(12), 2861-2872.

2.4 Exercises

Remember, you have two tasks. (1) To read through the material above, make sure you understand it, and that you can interpret and understand everything that is going on in the code examples. (2) Carry out the exercises below. Note also that they involve answering questions and thinking; you could write the code in seconds (as is the case for much of the work in machine learning), and so your task is to make sure you understand what is going on.

1. Below is code to download global temperature and precipitation data. Use it to carry out the following exercises.

```
library(raster)
r <- getData("worldclim",var="bio",res=10)
e <- extent(150,170,-60,-40)
data <- data.frame(na.omit(extract(r, e)))
names(data) <- c("temp.mean","temp.diurnal.range", "isothermality",
"temp.season","max.temp","min.temp","temp.ann.range","temp.wettest",
"temp.driest","temp.mean.warmest","temp.mean.coldest","precip",
"precip.wettest.month","precip.driest","precip.season",
"precip.wettest.quarter","precip.driest.quarter",
"precip.warmest","precip.coldest")
```

   (a) Perform a *scaled* PCA on the data.

   (b) Produce a *biplot* and *scree plot* of the data.

   (c) How many important axes of variation are there in global temperature and precipitation on the basis of this data? Describe each axis, but *if you write more than a sentence per axis you are writing too much*.

   (d) Calculate the correlation matrix for these data, and use it to confirm for yourself the conceptual links between a PCA and PCoA.

   (e) Perform an NMDS on these data. Do you learn anything more or less than the PCA or PCoA?

2. These questions are based around a classic ecological dataset: the Barro Colorado Island 50-hectare plot. Before the Panama Canal was built, it was a small patch of a much larger forest; now it is a roughly 100-hectare island upon which much of modern ecology has been based. Read into that what you will! You can load the data into *R* by running `data(BCI)` (it's shipped with `vegan`).

   (a) Perform a PCoA of the BCI plot dataset using a Euclidean distance matrix.

   (b) Perform an NMDS of the BCI plot dataset.

   (c) Repeat question 1a using a Bray-Curtis distance matrix (use the function `vegdist`). Compare your new PCoA with the output from question 1b. Why is it more similar to your answer to 1b?

3. These questions are based around a dataset of plant abundances across a gradient in temperature. You will find the code below useful to both load that data, plot it out, and calculate distance matrices from it.

Your task is to figure out what is driving differences in community composition within this data: one of your supervisors thinks there are two kinds of communities here (hot vs. cold environments), while the other thinks species are responding to a gradient of temperature. Your task is to help them figure out what's going on.

```
# Load the data in
comm <- as.matrix(read.table("hot-sites.txt"))
site.data <- read.table("site-data.txt")
# Build datasets of distances for quantile regression
dist.data <- with(site.data, data.frame(
dist=as.numeric(your.distance.matrix),
temp=as.numeric(dist(temp)),
groups=as.numeric(dist(outer(groups, groups, `==`), method="binary"))
))
# Color plots of data according to a categorical variable
some.plot.function(some.data, col=ifelse(groups=="hot", "red", "blue"))
```

(a) Perform an NMDS or PCoA of these data.

(b) Visualise this data, colouring them according to the groupings (hot vs. cold) one of your supervisors favours.

(c) Add to your visualisation from the above question some contour lines reflecting the environmental gradient.

(d) Use a quantile regression to determine whether there are two kinds of communities, or whether there is a graded, continuous response to the environment.

Chapter 3

Clustering—hierarchical, k-means, and model-based approaches

Overview

The distinction between dimensionality-reduction and clustering approaches can sometimes be hard to understand, because they both involve simplifying data. Fundamentally, clustering methods focus on detecting *groups* within data. Whereas something like a PCA tries to summarize lots of continuous variable into fewer variables, cluster analysis tries to detect discrete groupings in that data. Which you end up using depends on what you think is going on with your data: if you're interested in species delimitation, you'll probably prefer cluster analysis, for example. Clusters are also easy to present to non-scientist stakeholders (people like to put things into boxes, in my experience), but you must be careful to emphasize the uncertainty associated with clusters. While I'll give you some examples of some of the techniques that can be used to delimit clusters, the truth is that there is no single way to pick the "correct" numbers of clusters. So be careful, and use your understanding of the biology of the system to drive your approach!

3.1 Hierarchical cluster analysis

Hierarchical cluster analysis is all about splitting your data into groups. If you have a spreadsheet of data, it will split that data into as many groups as you have rows of data. That doesn't sound very useful, but the trick is that it makes those groups in a hierarchy—a series of splits—that allow you to stop splitting the data whenever you want, leaving you with as many or as few groups as you like. That leaves, of course, the rather knotty question of where you should stop splitting, but in practice when you look at the output of a hierarchical cluster analysis that's pretty obvious from eyeballing the data. If it isn't... then you probably want another method, and we'll get to that.

3.1.1 Let me break it down for you: an intuitive overview

The aim of *hierarchical cluster analysis* is to separate your data into a *dendrogram*—a tree—where each data-point (row) is a tip on the tree. Starting at the root of the tree, each split in the tree takes you closer and

closer to data, and as you go further along the tree all the data subtending (falling off from) each node should be increasingly similar. Building such a dendrogram requires the original data to be transformed into a *distance matrix*: essentially a table where all the rows and columns represent data points in the original data, and all the entries in the table are the distances among data-points. There is an alternative, called *model-based cluster analysis*, which doesn't rely on a distance matrix, but we won't be covering that today.

The devil is in the detail with any cluster analysis, and the most important detail is how you construct the distance matrix. The simplest is to use the *Euclidean distance* among points: this is just the distance between points if they were plotted according to whatever variables you've collected. So, for example, if you had data on how long and how wide a series of boxes were, then the Euclidean distance of these boxes would just be the distances among all the points when plotted in a scatter-plot. The calculation is much the same if you have three variables, or even thirty, but of course you can't visualize the plot if there are thirty variables! There are lots of different ways to measure distances, and the best one to use boils down to how your data are distributed, but you rarely go too far wrong with Euclidean distance if your data are Normally distributed. If you're dealing with discrete data, something like the *Hamming* or *binary distance* is often useful, which is literally just the count of the number of ways that two things are different (*e.g.*, "Will" vs. "Bill" would have a Hamming distance of 1, because "W"!="B").

Once you have your distance matrix, you can proceed to form your dendrogram. You start from the 'bottom' of the dendrogram, and find the two pieces of data that are most similar to each other. They then form a *cluster*, and the distance matrix is updated to reflect every other point's distance from that cluster. You then find the data point or cluster that is closest to another row in the distance matrix, replace that with a cluster, re-calculate, and keep going until everything has been grouped together into one big cluster. By keeping track of all the times you joined things together in the cluster analysis, you're able to draw a dendrogram—the dendrogram is just a plot of when each data-point joined up with another point, with the lengths of the branches in that tree proportional to the distance between that group/data-point and the group/data-point it joined. It makes a lot more sense once you see it plotted, I promise!

The final complication is how you update the distance matrix when you're changing everything around. If you replace a group with the average of all its members' distances, you're doing an average or Unweighted Pair Group Method with Arithmetic Mean (UPGMA) clustering. This is a popular method if you're an evolutionary biologist because, under the assumption of a molecular clock, it will produce dendrograms you might reasonably call phylogenies[1]. The default in *R* is *complete linkage clustering*, which replaces a group with its maximum distance to each other data-point/data-point in a group. I'm afraid there's no "best" method to use; just take a look at what you get and decide for yourself what makes the most sense.

3.1.2 Hands-on with hierarchical cluster analysis

To get started, let's simulate some data that fall into three fairly obvious groups. As always, if the method of simulation doesn't make much sense to you, don't worry about it. Focus on the plot at the end, and make sure you can see that there are roughly three groups.

[1] Of course, the molecular clock is an absurd over-simplification and isn't true, but don't let reality get in the way of a good story...

```
data <- data.frame(rbind(
  cbind(rnorm(50),rnorm(50)),
  cbind(rnorm(50,5), rnorm(50,5)),
  cbind(rnorm(50,-5), rnorm(50,-5))
))
data$groups <- rep(c("red","blue","black"), each=50)
names(data)[1:2] <- c("x","y")
with(data, plot(y ~ x, pch=20, col=groups))
```

Great, now we've got some data! So, let's create a distance matrix, using *R*'s default of a Euclidean distance matrix because we're dealing with well-behaved Normally-distributed data.

```
distance <- dist(data[,c("x","y")])
upgma <- hclust(distance, method="average")
plot(upgma)
comp.link <- hclust(distance)
plot(comp.link)
```

You'd probably agree, looking at this plot, that there are three main groups. Despite being built from the 'bottom-up', it is best to read hierarchical dendrograms from the top-down. Starting at the very top, you can see that there are three main splits, and then at the very bottom lots of sub-groups. There are three groups that seem quite distant from each other: look at the *height* and how there are three splits very high in the tree, representing groups that, when joined together in the cluster, have a very big distance from each other. 'Lower' in the tree, data-points within the clusters have lesser distances from each other and so join together at lower heights. You can also see that, for all the fuss that's made about it, in this case both the UPGMA and the complete linkage methods are forming similar numbers of clusters.

But what if we want to *cut* our data into groups? Well, luckily *R* has us covered and it's quite simple to cut the dendrogram at a certain point or into a certain number of groups.

```
cut.by.groups <- cutree(upgma, k=2)
cut.by.height <- cutree(upgma, h=8)
```

3.1.3 How many clusters?

The question everyone asks at this point is "but how many groups" and/or "but at what height" should you be doing your cutting? The answer is, surprisingly, quite difficult to get definitively[2]. The problem can be thought of as a ratio problem: what should the ratio of the average distance *among* clusters to the average distance *within* clusters be? Any choice of cut height will determine this ratio, as will a choice in the number of clusters (they are, essentially, the same thing). You might agree that lower intra-cluster distance and higher inter-cluster distance would be the best choice, but in practice it's difficult to find where the best 'stopping point' is (*i.e.*, what ratio of inter- to intra-cluster distance is 'correct').

[2]If you're interested, come by my office and borrow my MSc thesis. I assure you that you won't enjoy the experience of reading it; I certainly didn't enjoy the experience of writing it.

In my experience, one of the best approaches is the *DD-weighted gap statistic*, as defined by Yan & Ye (2007; *Determining the number of clusters using the weighted gap statistic*; Biometrics 63: 1031–1037). This metric is appropriate for hierarchical clustering and *k-means* clustering (more on that next time), and although it tends to work best with spherical clusters so does everything else I've seen that was worthwhile. It's a slightly non-standard install, but I promise you it's worth the effort:

```
install.packages("paran")
install.packages("ape")
install.packages("splits", repos="http://R-Forge.R-project.org")
library(splits)
gap.stat <- ddwtGap(data[,c("x","y")])
with(gap.stat, plot(colMeans(DDwGap),pch=15,type='b',
  ylim=extendrange(colMeans(DDwGap),f=0.2),
  xlab="Number of Clusters", ylab="Weighted Gap Statistic"))
gap.stat$mnGhatWG
```

...the take-home from this is the DD-gap-statistic (as you can see from the plot) is at its maximum when we assume there are three clusters in the data, which there were. The DD-gap-statistic (I always think of the "DD" as standing for "double-differenced") compares the average distances among points within a cluster for a given number of clusters with the distances if there were one fewer, or one more, clusters than a particular value. What this means, in practice, is that when you hit the 'correct' number of clusters *the DD-gap-statistic peaks*, and so to pick your number of clusters you find the peak and that's your number. This is also statistically testable via a parametric bootstrap approach based vaguely on the distribution of your data (see the help file for details). Or, if you're feeling lazy, you can just print out that final value (`mnGhatWG`), which is the "answer" according to this method: 3.

I'm afraid that, fundamentally, there is no universally-accepted definition of what a "good" separation of data looks like, so your choice is either to look at the plot and pick for yourself, or to trust something that someone has told you is quite good and that you like. So your choices are to pick for yourself, read the paper I reference above and see if you agree with me that it's good, or to just trust me[3].

3.2 K-means clustering

K-means is, intuitively, I think much easier to grasp than hierarchical clustering, and it is blindingly fast. However, it also happens to be the method that requires the most assumptions going in. All clustering methods require care in the selection of number of clusters, but this is perhaps most apparently in k-means.

[3]Oh, alright, I'll give you one more piece of advice: Brian Ripley has written reams about this topic, and he's really quite smart. So take a look at his work.

3.2.1 NP-hard? I don't know the meaning of the word

k-means clustering is so simple that it's probably the only algorithm in this course where I'm not going to simplify the explanation[4]. It proceeds in steps:

1. Pick as many centers—*centroids*—at random as you are trying to find clusters in your data. Note that there are many ways of randomly choosing those centroids, and some would argue you shouldn't do it randomly.
2. Assign each data point to the cluster whose *centroid* it is closest to.
3. Calculate the new centroid (center/mean) of all the points in each cluster.
4. Go back to step 2 and keep running through and repeating until the assignments of your data to clusters no longer change in step 3.

Amazingly, this often works quite well. If you restart the search lots of times and pick the "best" outcome, where best is defined as the run with the smallest within-group variance, it works even better. Leaving aside the problem that you have to pick the number of clusters to work with, there is one other issue, however: *k-means* isn't guaranteed to work as there's no way to be certain that it will ever return the perfect solution. This is because the problem is what is known as *NP-hard* in computer science: it means that the only way to be certain you had gotten to the best solution, as far as we know, is to try every single possible assignment of data-points to clusters and check. I mention this solely because, frankly, I find that amazing, and also because you're going to hear the phrase 'NP-hard' quite a lot if you Google around computer science things. There are lots of very, very complicated definitions of NP-hard floating around on the Internet: try and keep a hold of this slightly more intuitive, if not quite as precise, one, because it'll help you stay afloat[5].

3.2.2 Hands-on with k-means

Alright, this time let's throw something a bit more tricky at our clustering algorithm. It's not that hierarchical clustering can't handle it, but rather that now we're ready for a little more complexity.

```
data <- data.frame(rbind(
    cbind(rnorm(50),rnorm(50)),
    cbind(rnorm(50,2.5),rnorm(50,2.5)),
    cbind(rnorm(50,-2.5),rnorm(50,-2.5)),
    cbind(rnorm(50,5), rnorm(50,5)),
    cbind(rnorm(50,-5), rnorm(50,-5))
  ))
  data$groups <- rep(c("red","blue","grey80","grey60","grey20"), each=50)
```

[4]There are, however, multiple algorithms that are called the k-means algorithm. This is the most common, and is sometimes called *Lloyd's algorithm* (arguably not correctly). As with everything in life, if it's simple, someone's got to complicate it.

[5]There are so many definitions and articles about it because one of the classic unsolved problems in computer science: whether every problem whose answer can quickly be checked (*NP* problems) can quickly be solved (*P* problems). If *P*, meaning every quick to check problem can't quickly be solved, then we're all fine and dandy. $P = NP$ then pretty much every intuition we have about computing and statistics is wrong. Gulp.

```
names(data)[1:2] <- c("x","y")
with(data, plot(y ~ x, pch=20, col=groups))
```

As you can see, this data is a little bit more messy. Let's not give our algorithm very many random re-starts, and see how it does. Notice how we're going to use a contingency table to figure out whether our algorithm correctly assigns points to groups.

```
k.means <- kmeans(data[,-3], centers=5, nstart=10)
table(k.means$cluster, data$groups)
```

Huh. Even with only ten re-starts (which is vastly too few for a real-world application), we did OK! But the problem is, as I'm sure you've recognized, that in order to even get started we had to decide how many groups we wanted. What if we didn't know? Well, let's see how our old friend the gap statistic does in this case.

```
library(splits)
gap.stat <- ddwtGap(data[,c("x","y")])
with(gap.stat, plot(colMeans(DDwGap),pch=15,type='b',
  ylim=extendrange(colMeans(DDwGap),f=0.2),
  xlab="Number of Clusters", ylab="Weighted Gap Statistic"))
gap.stat$mnGhatWG
```

You might have noticed that it took a little longer than it did last time; this is because we've thrown a lot more data at the method. If you had a much larger dataset, you'd probably have to look into other ways of measuring intra- vs. inter-group variance, but truth be told I don't think you'd find yourself getting an answer much quicker or much better than the one we have here. Because, as you've probably noticed—it worked! Once again, we've detected the correct number of clusters!

3.2.3 Fuzzy clustering

If you're like me, you probably don't like putting people into boxes and categories. If so, then good news—neither do the people who develop clustering algorithms! *Fuzzy clustering* algorithms are extensions of standard clustering algorithms where, instead of assigning data-points to categories, we give them *membership coefficients* (often in a matrix called w that sum to 1 for each data-point) describing how each data-points 'belongs' to each category. Using them in *R* is pretty straightforward, once you've installed the relevant packages:

```
library(fclust)
fuzzy <- FKM(data[,c("x","y")], k=5)
# Get lots of summary statistics
summary(fuzzy)
# Plot the predictions out
plot(fuzzy)
```

This algorithm is essentially the same as *k-means*, only now we're measuring the mean intra-cluster distance weighted by all the relative membership coefficients (a *weighted mean*). Where do those coefficients come from? Well, it depends... But they're always some kind of weighted distance to the centroid. The precise

form of the weighting function is up to you to decide (or go with the default); this is a remarkably active area of research and if you're interested I recommend the review in the opening of Campello & Hruschka (2006; "*A fuzzy extension of the silhouette width criterion for cluster analysis*" in Fuzzy Sets and Systems 157: 2858–2875).

There are extensions of the gap statistics I have taught you for fuzzy clusters, but to be honest you're better off using model-based clustering. Indeed, if you're going to acknowledge that groupings are fuzzy such that something is almost in several clusters, you might be wondering where you draw the line—what's the most likely cluster a point is in? Those of you who have taken my other statistics class might be wondering if there is some way to find the most likely cluster (...the maximum likelihood cluster...) for some data. Well, if you are, then you really should use model-based clustering, which is a natural extension of fuzzy methods!

3.3 Model-based clustering

If you're the sort of person who's not happy with approximations to things, you're probably wondering what that "model-based clustering" I was talking about in the last section is all about. Well, it's actually quite simple to do in *R*, even if it might not be so easy to understand. You can think of cluster-able data as coming from a series of different statistical distributions, each with their own parameters and possibly functional forms. It should, in principle, be possible to simply fit a load of different potential models to the data, then calculate the fit of those models using something like AIC or BIC and pick the model with the best fit[6]. Such a thing is possible, and is precisely what the package `mclust` does for you. Because, for speed, it operates in a frequentist perspective, it uses an *expectation maximization* algorithm; this basically means it tries to fit all the independent parts of its equation one step at a time, using (1) current parameter estimates to estimate the groupings, then (2) using those groupings to re-estimate the best parameter estimates, and then going back to step (1). This looping repeats until the algorithm has converged on a set of numbers. In well-behaved data, it also works like a charm...

```
library(mclust)
model <- Mclust(data[,-3])
summary(model)
model
plot(model, what="BIC")
```

If you're interested in learning more about this approach, I can't recommend enough reading Ezard *et al.*'s 2010 paper *Algorithmic approaches to aid species' delimitation in multidimensional morphospace* in BMC Evolutionary Biology (10:1, 175). It's a (remarkably approachable, given the subject matter) introduction to the topic, and it uses these techniques to solve a very interesting species delimitation problem.

[6] I would argue this is no longer a machine learning approach—it's just a complicated classical statistics problems. Indeed, this is actually *precisely* how we simulated our example data above

3.4 Exercises

Remember, you have two tasks. (1) To read through the material above, make sure you understand it, and that you can interpret and understand everything that is going on in the code examples. (2) Carry out the exercises below. Note also that they involve answering questions and thinking; you could write the code in seconds (as is the case for much of the work in machine learning), and so your task is to make sure you understand what is going on.

1. Below is code that will load in over 100 years of electoral data from the US, recording the proportion of the vote in each state that the Republican party received. Notice how we use a *logistic* transformation on the data to normalize them because percentages are not normally distributed; if you are unfamiliar with this feel free to ask me or not to worry about it and take my class next semester on the analysis of such data. I am asking you to analyze voting data, not to pass judgment on anyone who votes; I am not intending to somehow influence your voting intentions, I'm just showing you some history.

```
library(cluster)
votes <- na.omit(cluster::votes.repub)
logit <- function(x) log(x / (1-x))
transformed <- logit(votes/100)
```

   (a) Perform a hierarchical cluster analysis of these data.

   (b) Use `ddwtGap` with the argument `genRndm="uni"` to see how many statistically significant clusters there are according to this metric.

   (c) Do you think the output of `ddwtGap` is correct? Why (not)?[7]

2. Download the file `board-game-mechanics.csv` from the course website. This is a binary matrix of the top-100 games on https://boardgamegeek.com/, detailing the mechanics of each board game[8]. Download that data, load it in using the code below, and use it to carry out the following exercises. *You do not require any knowledge of board games to answer these questions, and I strongly encourage you not to attempt to develop any in order to answer these questions.*

```
data <- read.csv("board-game-mechanics.csv", rownames=1)
```

   (a) Build a hierarchically-clustered dendrogram of these data. They're *binary* data, so create a `binary` distance matrix.

   (b) Compare, graphically, your dendrogram from part (a) with a dendrogram built using the `average` method. What are the main differences you can see? Which dendrogram do you think would be the best to present to a policy-maker, and why?

[7] There is no correct answer to this question. I am trying to get you to develop a healthy suspicion of the output of any machine learning algorithm.

[8] Those long winter evenings just fly by in the Pearse household!

(c) It takes too long for us to use `ddwtGap` easily on this dataset[9], but try using `cutree` to split your data into groups. Does this change your opinion about which of the methods you would prefer to work with? How do you think the best way to split this tree is?[10]

3. The code below reminds you how to load the `iris` dataset into *R*; use it to carry out the following exercises.

```
data(iris)
head(iris)
```

(a) Use `ddwtGap` to determine the optimal number of clusters in this dataset.

(b) Run a *k-means* analysis with at least ten restarts, using the number of clusters you identified above.

(c) Build a contingency table of your *k-means* clustering versus the reality of the species definitions. Do you think that *k-means* has done a better job than PCA did of describing this dataset? Why?

(d) Repeat this analysis using a model-based clustering approach.

4. The following exercise makes use of data from Wright *et al.* (2004; The worldwide leaf economics spectrum. Nature, 428(6985) 821), and this data should not be shared outside this class in any way[11]. It concerns data collected on a number of traits related to plant leaves (the details don't matter); each plant's average biomass when grown in an experimental garden is also listed in the dataset as a response variable[12]. You can find the data in the file `glopnet-biomass.csv` on your course website.

(a) Fit a standard linear model (*e.g.*, something like `lm(biomass ~ some.variables, data=data)`) to see what plant traits explain biomass in this experiment. What does you model tell you? Why is it not working very well?

(b) You now know at least three different ways of summarizing data for downstream analysis: PCA, hierarchical cluster analysis, and k-means cluster analysis. Explore this dataset using these methods to see what you think summarizes the data best.

(c) Use whichever method you think performs best to fit a new linear model of the data. Does this perform better or worse than the original model? Why?

[9] Try it if you're interested, but be patient and set the `maxClust` to be a bit higher than the defaults.

[10] There is no perfect answer to this question. Use your brain, use your intuition, and think about what seems the most intuitively appealing.

[11] I am required to state this by the conditions of the data release; I would also recommend you not work uncritically with data I have manipulated for a classroom exercise

[12] Please don't think about the experiment too much, because *it's not real*.

Chapter 4

Supervised methods—trees, lassos, and SVMs

Overview

The huge variety of supervised machine learning algorithms—approaches that require response variables—is often overlooked because students are so desperate to learn about artificial neural networks. This is a shame, because the ML toolkit contains a number of extremely powerful tools that are capable of solving a number of very common problems. In this section we are going to cover classical tools that are in many ways the ML complement of Generalised Linear Models (GLM). These approaches are capable of either categorising data (*e.g.*, true/false, red/blue/green) or predicting continuous variables. They set themselves apart from approaches like GLM in terms of their flexibility; some are capable of dealing with overwhelmingly large number of potential explanatory variables (*e.g.*, lasso regression), managing complex and multi-faceted interactions naturally (*e.g.*, regression trees), or are so flexible that they can handle almost any classification problem (*e.g.*, SVMs). Critically and importantly, even when these approaches work with categories, they are *supervised* approaches, and so you are required to give the aclgorithms exemplars of the different categories for which you are building a model. Whereas a clustering algorithm will choose the identity (and potentially number) of groups for you, in supervised methods you must give data on cluster identity.

4.1 Regression trees

This is our first ML algorithm where we're going to be predicting (continuous) data, and so it's our first supervised machine learning algorithm. Because of this, I'm going to walk you through how we validate machine learning algorithms by setting aside some of our data for testing. Machine learning algorithms have no coherent statistical philosophy or definition—they are literally just computational instructions that happen to work well—and, as such, verifying your models using independent data is a vital thing to do. If you remember one thing from this course, let it be this: *woe betide those who fit machine learning algorithms but do not test their performance with independent data*.

4.1.1 An informal introduction to regression trees

When we try to make decisions, we often use rules of thumb or heuristics. An example is "*I'm going to boil that egg for five minutes*": we know that boiling it for four minutes and fifty-nine seconds would give much the same result, as would boiling it for five minutes and one second, but we pick a rule ("*stop at five minutes*") and stick with it. Similarly, while you're used to regression methods that work with continuous data ($y = 2 \times x$), *regression trees* split continuous data into discrete decisions. For small, continuous datasets, that may seem like a bad idea, but for datasets with lots of explanatory variables it's often quicker to split things like this, because it makes it clear what variables are important and makes it very easy to visualise interactions.

Regression trees work exactly like this, and they look remarkably similar to the hierarchical clustering outputs you're already familiar with. Starting at the top of the tree, you can follow the course of it all the way down, asking yourself at each step whether the particular row of data you're trying to predict you're in has a lesser (go to the left branch) or greater (go to the right branch) value than the decision in front of you. Once you get to the bottom of the tree—you're finished! You've made your prediction. The process for categorical data is much the same: the only difference is you split things according to the categories they all fall into.

The details of regression trees we're going to learn through experience. But there are four main kinds of regression tree we're going to cover (briefly) today, that I list in both the order you'll encounter them and relative complexity:

1. **Regression trees.** Exactly what I've just described above.
2. **Bagged** regression trees. Split your training data into lots of little subsets, fit across them all, and then average across all those trees.
3. **Random forests.** Exactly the same as bagged trees, but each random subset also has a subset of the total set of explanatory variables.
4. **Boosted** regression trees. Fit a regression tree to data, then fit a regression tree to the residuals of that model. Average across the two, then fit another model, take those residuals, and fit another model. Keep repeating until you get bored.

Regression trees can also be run with a categorical response variable, in which case they're called *decision trees*. I'm sure you'll agree, however, that four new kinds of machine learning technique are more than enough for your first lesson, so don't worry about them!

4.1.2 Hands-on with regression trees

First of all, let's simulate some data that we're going to work with. We're going to continue our example of plant diversity across a series of (a)biotic gradients—temperature, humidity, soil Carbon, and herbivore diversity—and simulate some data under a reasonable biological model (greater diversity in tropics, and a sort-of trophic cascade effect of herbivory). Notice that we're going to model diversity of plants using a Poisson distribution to give some noise to the data; if you're not familiar with the Poisson, just treat it as a way of getting some (very noisy) variation around the relationships we're simulating. Also don't worry if you don't track what's going on with the model of plant diversity; just use `boxplot` to examine what's going on with the

relationships of all the variables, and notice that you would have had a great deal of difficulty picking out most of these relationships by eye.

```
# Build model of species diversity
data <- expand.grid(temperature=seq(0,40,4), humidity=seq(0,100,10),
  carbon=seq(1,10,1), herbivores=seq(0,20,2))
data$plants <- runif(nrow(data), 3, 5)
data$plants <- with(data, plants + temperature * .1)
data$plants[data$humidity > 50] <- with(data[data$humidity > 50,],
  plants + humidity * .05)
data$plants[data$carbon < 2] <- with(data[data$carbon < 2,], plants - carbon)
data$plants <- with(data, plants + herbivores * .1)
data$plants[data$herbivores > 5 & data$herbivores < 15] <-
  with(data[data$herbivores > 5 & data$herbivores < 15,], plants - herbivores * .2)
# Draw random data from Poisson based on this
for(i in seq_len(nrow(data)))
  data$plants[i] <- rpois(1, data$plants[i])
```

Now let's fit a regression tree to that data, and then plot it out. Once you see the plots, suddenly the structure of a regression tree is going to make much more sense to you. It's a series of 'decisions' you make, working from the top of the tree to the tips, where each decision maps onto the value of the explanatory variables. At the very bottom, we have the estimate of the response variable (plant richness) that we would expect. Notice that you can get the residual variation out of the model, just as you would with a standard linear regression.

```
library(tree)
# Pick some training data and then fit a model to it
training <- sample(nrow(data), nrow(data)/2)
model <- tree(plants~., data=data[training,])
# Examine the model
plot(model)
text(model)
# Look at the statistics of the model
model
summary(model)
```

However, if we *really* want to be sure of whether our model is doing well, we should test its performance on the data we didn't fit to it. Remember: this is important to do in machine learning methods because they're not necessarily based on some fundamental, deep aspects of statistics: they just happen to work very well in the right circumstances. So we need to be careful. We can also cross-validate our model to see how it performs under different tree depths (numbers of nodes), seeing how the Mean Squared Error (calculated just as for a normal regression) changes. It's your job to decide where to draw the line and what constitutes a good fit, but in this case it's clear that we're doing a pretty good job (the line goes down), albeit we could fit a simpler model without much change.

```
# Check performance outside training set
cor.test(predict(model, data[-training,]), data$plants[-training])
# Check cross-validation of model
plot(cv.tree(model))
```

Maybe you're not so impressed with the roughly 0.60 sort-of r of this dataset. Do you think our use of `rpois` could have had anything to do with it? How could you find out?...

4.1.3 Bagged regression trees and random forests

The problem with fitting regression trees is that they fit your data so well: that's why we've been splitting our data in half and only training on one half of it to check we've not *over-fit* and so built a bad model. One way, potentially, of dealing with this problem is so randomly take subsets of our training data, fit regression trees to those *bagged* bootstrap replicates, and use the average of all those regression trees. It might not seem obvious, but by averaging across all of our trees like that, we reduce the variance among them and so (hopefully) reduce the variance associated with only working with a sample of data (*i.e.*, the degree of over-fitting in our data). Maybe that makes sense, maybe it doesn't, but the take-home is that *bagged regression trees*, where you work with the average of many regression trees fit to many bootstrapped subsets of your data, are a good solution to the problem of over-fitting in data. Fitting them is simple, and they can be tested just as you tested regression trees.

```
library(randomForest)
model <- randomForest(plants~., data=data[training,], mtry=ncol(data)-1)
cor.test(predict(model, data[-training,]), data$plants[-training])
```

You might be wondering what that `mtry` argument to `randomForest` is all about. This tells the `randomForest` package to make sure, each time it's trying to make a new split in the tree, to consider all the variables available to it. If we go with the default (or set any number less than the total number of variables), we restrict the number of options available randomly each time. When we randomly change the available explanatory variables each time a split is being considered in each of the bootstrapped trees, we are building a *random forest* model. It might sound a bit strange, but this actually improves things because it means the bootstrapped trees resemble each other less. When you take the average of things that are correlated with one-another (resemble each other) more than you expect, you don't tend to reduce variance as much as you would hope, and so by 'decorrelating' your trees like this you improve estimates.

Below I show you how to fit random forest models, and also show off *variable importance*. This is simply the average decrease in Mean Squared Error each time a split in a regression tree is fit to a particular explanatory variable: it's essentially the r^2 of each variable. Since we can't look at a single regression tree anymore (we've fit thousands of them!), this is perhaps the simplest way to understand what variable is doing what in your model.

```
model <- randomForest(plants~., data=data[training,], importance=TRUE)
importance(model)
cor.test(predict(model, data[-training,]), data$plants[-training])
```

4.1.4 Boosted regression trees

Boosted regression trees are perhaps the most 'meta' of the regression tree family. They also fit a series of models, but each time to the residuals of the previous model, which is then added in to the set of predictions from the previous model, and the process repeated again. So you end up with a model that's a hybrid of a regression tree fitted to the original data, and a series of models that are trying to fit the variation that the first model didn't fit very well. The intention is to avoid over-fitting the data through this approach, but the disadvantage is it makes the meaning of the model a little more obscure. Of course, as with all machine learning algorithms, our intention here isn't necessarily to be easy to interpret, but to perform well!

There are a lot more parameters to play with in this model-type, but there are two I want to draw your attention to. The first is that `gbm` allows you to fit on a link function, much as you can in a Generalised Linear Model, so here I've told it that we're dealing with count data (`distribution="poisson"`). If you're familiar with this, then that's all great, but if you're not then you'll be fine not setting this option (as I do in the second example). Worry about it once you've covered Generalised Linear Models. Second, there is a *shrinkage* parameter that controls the relative importance of earlier vs. later fitted models in the boosting process. A smaller value means trees later in the process (residuals of the residuals of the residuals of the...) are given relatively more weight than if the parameter is greater. Generally, smaller values give better results, because the whole purpose of the exercise is to allow those later trees to matter, but as with everything experimentation pays dividends.

```
library(gbm)
model <- gbm(plants~., data=data[training,], distribution="poisson")
summary(model)
# A plot of variable importance should also have appeared now
faster.model <- gbm(plants~., data=data[training,], distribution="poisson", shrinkage=.1)
```

4.2 Lasso regression and Least Angle Regression (LAR)

Least angle regression (LAR) is essentially something called the *lasso regression* on steroids, so we're going to start off by learning how to use the lasso and then move onto its somewhat more unwieldy younger brother LAR. What's interesting about these two is that, unlike regression trees, they were designed to help us pick between very large numbers of candidate explanatory variables. Whereas regression trees are quite happy to make use of all the factors, and relative variable importance is something of a side-note, that's not the case in LAR. Here the whole purpose is to figure out which explanatory variables you can throw away and ignore, and so it's a favourite of bioinformaticians, busy MSc students, and lazy lecturers. I think you'll like it: I do!

4.2.1 A wordy introduction to lasso regression and LAR

All the regression techniques you learnt before this course involved *least squares*: you were trying to find an equation that minimised the squared error of the difference between a response variable and a regression line. The obsession with squaring everything was explained to you as a way of making over- and under-prediction comparable, and those of you who said "why can't we just use the absolute (modulus) of the error instead" were likely told to be quiet. Well, now it's time to grow up and learn the real reason: working the absolute (modulus) of anything is much harder, but it's what we use in *lasso regression*. In lasso regression, we are trying

to minimise the modulus of the error, *subject to the constraint that we don't want the sum of absolute value of our coefficients to be too great*. The sum of the absolute value of our coefficients is called the 'L_1 arc length', or the *lasso penalty*, and we want to minimise it because we want to keep the complexity of our model as low as possible: a greater L_1 means we've got more 'stuff' in our model, and more stuff means more complexity and we hate complexity. This is exactly the same as a standard linear regression with multiple variables: the only difference is we can't use any of the normal calculus[1] to estimate what the 'significant' variables are, and so instead we must try and minimise some sort of penalty.

To figure out what the best model is, we can make this penalty greater or lesser, and plot out how much better or worse our model predictions get. The one big advantage we have here is that, when the arc length is infinitely small, only one variable will have a non-zero coefficient, and as we increase it variables will slowly add in one-by-one. So if we can figure out the arc-length at which the model fit isn't really much better, we've picked our best model! Breathe for a minute (because that was quite a lot!), and if you get lost focus on the basic principle which will become much clearer once you start fitting these models: we are minimising the absolute error of a model, penalising ourselves to have as simple a model as possible and so finding the best set of predictors in our model. Figure 4.1 shows, on the left-hand-side, an example of how this process works. Lasso starts at the left-hand side of the figure, where all the coefficients are equal to zero. It then slowly increases the arc length (the same thing as decreasing the penalty), slowly increasing the absolute value of a coefficient. Each dashed line represents the point where, now the arc length is allowed to be a certain size, another variable is able to 'jump in' and making its coefficient non-zero increases model fit. Somewhere between the far left-hand and right-hand sides of the plot lies a model that optimally trades complexity for explanatory power.

Least angle regression (LAR) is so similar to the lasso that the statistical machinery underlying it is often used to more efficiently estimate lasso regression coefficients. As with lasso, there is a penalty term that we can increase from zero; as we do so, we find the explanatory variable most correlated with the response variable and increase its coefficient as much as the penalty/arc length will allow. Eventually, another variable will correlate more with the left-over variation, and so increasing its coefficient would give us a better fir than increasing the coefficient of the explanatory variable we've been working with: so we then increase *both* variables' coefficients (still constrained by the penalty). This process continues as we increase the arc length (decrease the penalty), adding in more and more explanatory variables as we do so until everything has been added in (the far right-hand side of figure 4.1). This is called a least angle regression because, geometrically, when we're adding all these variables in we're moving the 'angle' of the coefficients in a direction that matches the residual variation in the response variable at each step. Don't worry about the name, but do notice that both panels in figure 4.1 look incredibly similar—in practice, lasso and LAR are extremely similar.

4.2.2 Going to the rodeo

Enough talk, let's do it! Let's start out by simulating a dataset with 1000 explanatory variables, only two of which (columns 123 and 678) are significantly related to our data. Consider how you would approach the problem of determining what 'significantly' explains our response variable with this much data using something like a linear regression.

[1] Actually, we can't use any standard calculus, and that's the problem. See? I told you using the absolute value made things harder. Those of you who know about Lagrange multipliers will recognise that the lasso penalty is often called λ for a reason...

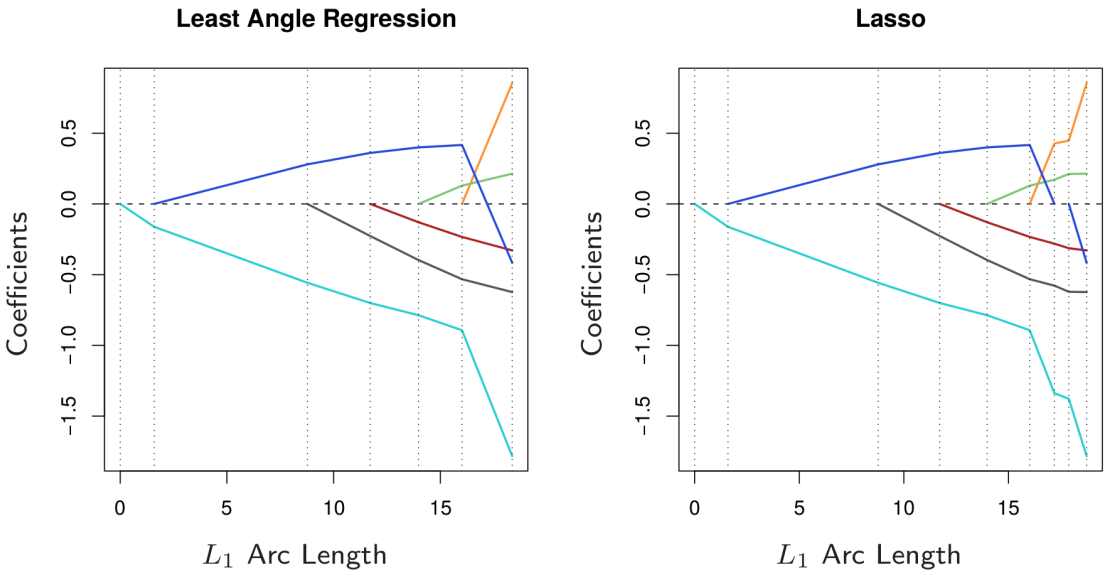


Fig. 4.1 **Model coefficients and arc length in lasso and LAR.** Taken from Friedman *et al.* (2001; The elements of statistical learning.).

```
explanatory <- replicate(1000, rnorm(1000))
response <- explanatory[,123]*1.5 -explanatory[,678]*.5
```

Now let's fit a *lasso regression*. Notice how we're using the package `lars`: because the LAR approach can be used to generate lasso estimates more efficiently, this package does everything we need for today.

```
library(lars)
model <- lars(explanatory, response, type="lasso")
plot(model)
```

That plot should look remarkably familiar, as it's basically the same as what you see in figure 4.1, only this time with some real (simulated) data. Which is all well and good, but (as you've probably noticed if you've played around a bit) it's not that easy to get estimates of coefficients out of this model. That's because, while lasso (and LAR) has its own stopping criteria for when a model is 'good enough', it gives you the coefficients for each step in that search. Luckily, however, it's reasonably trivial to write your own 'wrapper' function that will grab the coefficient estimates in a format you can work with. If you disagree, then you're in luck, because I've written one for you:

```
signif.coefs <- function(model, threshold=0.001){
  coefs <- coef(model)
  signif <- which(abs(coefs[nrow(coefs),]) > threshold)
  return(setNames(coefs[nrow(coefs),signif], signif))
}
signif.coefs(model)
```

Hopefully the above code isn't complete gibberish to you, but you are quite welcome to treat it as a magic function that will tell you the final coefficients whose coefficients are interesting in the final model. You can, and should, play with the `threshold` for what defines an 'interesting' coefficient (see the next section for the importance of variable scaling, which will affect this too).

In case you've missed it, lasso regression works amazingly well, particularly given how quick it is. **From one thousand input explanatory variables, it correctly estimates the two variables that mattered.** If that's not impressive, then I don't know what is.

4.2.3 This isn't your first rodeo

Least angle regression (LAR) operates in much the same way as the lasso, because it's the same package.

```
model <- lars(explanatory, response, type="lar")
plot(model)
signif.coefs(model)
```

So now is an excellent opportunity to show you the importance of *scaling your variables*. Everyone remembers that standard linear regression assumes independent, Normally-distributed variables, but what people often forget is that the optimisation routines inherent in essentially every single statistical solver (and often the mathematics itself) assume that your variables are on the same scale and centred. This means that each variable should have roughly the same standard deviation, and a mean of 0—*your explanatory variables should be z-transformed*. There are many reasons for this, but perhaps the most intuitive is that the maths/computing is going to be driven by a variable that varies over a much greater range, because all its coefficients are going to be much larger. Numerically, a computer might miss a coefficient whose value is 10^{-23}, even if when the variable is z-transformed it would be much "more significant" than a variable with a greater (but less important) coefficient. If you understood the importance of scaling variables in a PCA, then it might be helpful to know that the underlying logic and reasoning is the same in both cases. Anyway, you can see the effect for yourself below when I turn off `lars`'s default to automatically normalise all the variables. Notice that, because I simulated the data from a Standard Normal distribution, I have to set the threshold to 0 to see the effect.

```
bad.model <- lars(explanatory, response, type="lar", normalize=FALSE)
signif.coefs(bad.model, thresh=0) # Wow what a lot of coefficients!
signif.coefs(model, thresh=0)     # Nothing wrong here :D
```

...So maybe you *should* standardise those variables before you fit that regression your advisor is desperate for you to run, eh?

4.3 Support Vector Machines (SVM)

Support Vector Machines (SVMs) are useful for splitting data into groups. Thus, while they can be re-purposed for continuous response variables, you can think of them as the supervised complement to the clustering algorithms you already know. They are tremendously flexible and efficient, which makes them very useful in odd circumstances that tend to crop up a lot. For example, one-class variants of them are very popular in ecology because they can be used to predict where species should be found without any data on where they aren't found. Equally, because they are so efficient, until quite recently they were used in smartphones to detect when someone wanted their phone to pay attention to them [2]

[2] "Hey Siri, record everything I'm saying and relay it to the government please". For more details in the context of the evolution of bird song, see Pearse et al. (2018; Evolution 72(4):944–960).

4.3.1 An informal introduction to SVMs

Imagine you have data that are split into two categories (classes): an example is drawn in figure 4.2 with light-blue (on the left) and dark (on the right) categories. If you can draw a single line—which we'll call a *separating hyperplane*[3]—separating the two classes, then you could probably draw an infinite number of them by just tweaking the line a little up/down or altering its slope somewhat. You could, however, find a *single* line that has the furthest minimum distance to the points in the data; let's agree, for the sake of argument, that such a line is the "best" line and is called the *maximal margin hyperplane*, and that minimal distance is the *margin*. The crazy thing about this line/hyperplane/whatever, is that it doesn't really depend on any of the data other than the points lying on the margin[4], which is wonderful because it means that no matter how big our dataset is we can just focus on these and we'll be alright. We call these important points the *support vectors* of our classifying hyperplane, and identifying these points—and so the hyperplane that we can use to classify our data—is the sole goal of a *support vector machine*.

What's really exciting about support vector machines (SVMs) is that they don't just have to be fit with straight lines. Such *linear* SVMs are, of course, quite common, but we can fit essentially any kind of fancy complex classifier we want. Instead of drawing a straight line, we could draw a polynomial—or even a circle—by specifying a different equation for that line, which we call a different *kernel*[5]. These *non-linear* (*i.e.*, not straight!) kernels are really useful and let us fit all classify all kinds of whacky datasets, as I give an example of in figure 4.3.

I'm not going to give you anything more profound a statistical insight about this than what I've already given you. SVMs are extremely complex, and it would be a disservice to claim that I can teach your their theoretical underpinings in a single class. You do now, however, know enough about them to use them and, critically, to understand when they perform well.

4.3.2 Practical examples

To get a feeling for how SVMs work, let's simulate some data. We're going to make a dataset with two random variables, and then set make a set of points "in the middle" as somehow different. If that doesn't make a great deal of sense to you right now, don't worry: just look at the plot you'll get out at the end and things will make sense.

```
data <- data.frame(replicate(2, rnorm(1000)))
data$y = (rowSums(data) > (median(rowSums(data)) - 1)) &
  (rowSums(data) < (median(rowSums(data)) + 1))
with(data, plot(x, pch=20, col=ifelse(y, "red", "black")))
```

You could pretend, if you wanted, that the y variable here represented whether or not a particular species tends to be found in a particular climatic envelope. Pretend the x-axis in your plot represents the temperature of a series of sites, and the y-axis represents the humidity.

[3] *Separating* because we've separated the data, and *hyperplane* is just a fancy word for a line

[4] If you think about it, you'll have to agree that there must be at least two such points

[5] This is something of a simplification, but if you are familiar with how Generalized Linear Models use *link functions* to fit equations to transform data in order to fit it in a different parameter space, then you understand what's going on. If you don't: don't worry about it, we're just drawing lines with different shapes.

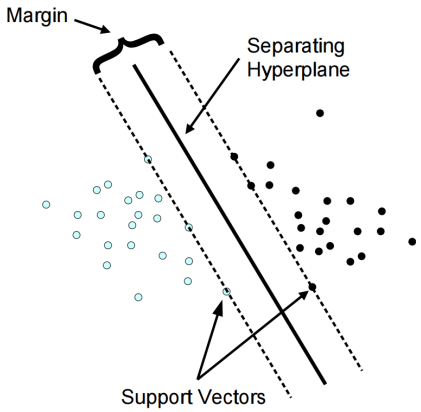


Fig. 4.2 **A simple overview of Support Vector Machines.** If we want to find a way to classify these two classes of data, we need to draw a line—a *separating hyperplane*—somewhere through this graph. To find the best line, we pick the one that has the smallest minimum distance from all the points—that distance defines the *margin*—and the points that lie on that margin are the *support vectors*. So a *support vector machine* is something that classifies data using some kind of hyperplane, and that hyperplane is defined using the support vectors themselves. This figure is taken from the vignette for the *R* package `e1071`, which you will be using in class.

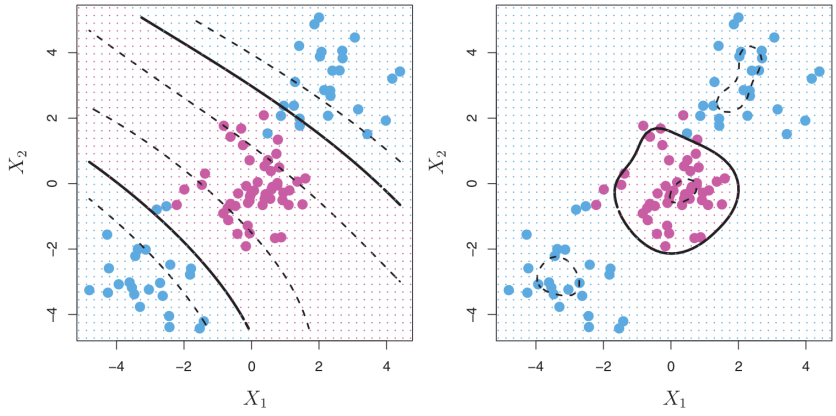


Fig. 4.3 **SVMs with non-linear kernels.** Not everything needs to be drawn on a straight line, and it's possible to have polynomial (left) and radial (right) kernels. These figures are taken from *An introduction to statistical learning* by James *et al.* (2013).

Now let's see if we can fit an SVM, in this case using the default *radial* kernel but it could be with anything else you chose (*e.g.*, a simple linear kernel), and see if we can classify between these two different classes.

```
library(e1071)
training <- sample(nrow(data), nrow(data)/2)
model <- svm(y~., data=data[training,], type="C")
plot(model, data[training,])
```

Success! In our training dataset of 50% of the data, it seems our model does a good job. Note that I'm stating that on the basis of the plot, *which you should generate for yourself in* R. Note how the SVM has identified the region in the center as different, and, according to the colors and the symbols, seems to have predicted the training data well. See if you can figure out, looking at the plot for yourself, what is going on. There are many kinds of SVM, and I'm only teaching you one of them in this class. Notice that I can make double-sure that `svm` fits a classification SVM to our data (`type="c"`): this function can be a bit picky, and you may find (for example, in the exercises) that you need to *force* `svm` to fit the kind of model you want—this is how to do that. Of course, to fit our model properly, we would want to test our model on some independent data that were not used to train it. So let's see what happens when we do that:

```
table(predict(model, data[-training,]), data$y[-training])
```

Success! Even with data that wasn't used to fit the SVM, we correctly identified the overwhelming majority of cases. I can tell this because, in this contingency table, most of the data are in the diagonals: the model predicted `TRUE` most often when the data were `TRUE`, and `FALSE` when the data were `FALSE`.

If you're interested, you might want to read about the parameters that actually underlie the way the SVMs are fit. All SVMs include a *cost* parameter, which represents how much 'wiggle room' the SVM is allowed when trying to find the margin in comparison with error in the data. Another parameter is the γ (*gamma*) parameter, which you can think of as a non-linearity 'fudge' parameter. Larger values mean the kernel becomes more non-linear, which is a good thing if the data really are non-linear but a bad thing if they're not as it opens you to over-fitting problems. Perhaps the best way to see if you γ parameter is leading you astray is to check the performance of your parameter on non-training data—but is that cheating?...

```
tune.svm(factor(y)~., data=data[-training,], gamma=c(.5,1,10), cost=c(1,10))
```

4.3.3 Details and other applications

There are two extensions to SVMs as presented here that are worth mentioning, but I don't want you to worry about the details right now. The first is SVMs that can deal with multiple classes of data to predict: a military application of such data might be predicting whether a person is an enemy combatant, a friendly combatant, or a civilian (three classes). One approach for such data is to fit models that classify data into all the pairwise combinations of classes (*e.g.*, enemy–friendly, enemy–civilian, and friendly–civilian)—this is *one-versus-one* classification. Whichever class is most frequently predicted is treated as the prediction. The second approach is to fit models predicting each class vs. some other class (*e.g.*, enemy–other, friendly–other, civilian–other), measure which of the models most confidently predicted each particular piece of data under observation, and treat the most confident estimate as the correct one. This is *one-versus-all* prediction. These two options work

better than you might imagine, although we'd probably all agree we'd rather not be anywhere near something like this being applied in the real world!

The second extension, which is quite common in ecology, is the *one-class* SVM. This has become an extremely popular method in ecology, where it is used to predict where a species might be found on the basis of environmental data. This is often called *niche modeling*, and while there are many statistical techniques that can be used when we have confirmed presences *and* absences of species on a landscape, dealing with the more common case where we only know where a species is, but do not know with much certainty where it isn't, is much more difficult. This approach was pioneered by Guo *et al.* (2005; Ecological Modelling 182; 75–90) and is well-worth a read if you're interested.

4.4 Exercises

Remember, you have two tasks. (1) To read through the material above, make sure you understand it, and that you can interpret and understand everything that is going on in the code examples. (2) Carry out the exercises below. Note also that they involve answering questions and thinking; you could write the code in seconds (as is the case for much of the work in machine learning), and so your task is to make sure you understand what is going on.

1. The following questions require you to use a (slightly simplified) version of data from: Quinlan (1993) "*Combining Instance-Based and Model-Based Learning*" in Proceedings on the Tenth International Conference of Machine Learning, 236–243. This dataset describes the fuel efficiency of several kinds of car (`mpg`—miles per gallon), and can be described as a function of various other properties of the cars. The dataset is available on your course website as `auto-mpg.txt`.

   (a) Fit a regression tree to a training subset of data.

   (b) Plot out the resulting model, and explain in a few short sentences what the output shows.

   (c) Validate your regression tree using independent data (*i.e.*, not your training data).

   (d) Fit a lasso regression to these data. Plot out your model and explain, in a few sentences, what the fitting process is showing you as it progresses.

   (e) Now fit a LAR to these data. Compare your results with those of parts (a), (c), and (d) above. Which model do you find the easiest to interpret? Why?

2. The following questions concern a dataset of forest fires (see http://www3.dsi.uminho.pt/pcortez/fires.pdf). Your task is to model the area of forest that was on fire as a function of various weather factors and some forest-fire indicators that the Canadian government uses. As your response variable is an area, you might want to employ a good-ole'-fashioned `log10(area+1)` transformation to it[6].

   (a) Fit a regression tree to this data. Explain what it means intuitively.

   (b) Fit a bagged tree, a random forests model, and a boosted tree to a training subset of these data.

[6] Although, of course, as this is a statistics class I am required to remind you that you shouldn't really do so 'in the wild': O'hara & Kotze (2010) Methods in Ecology and Evolution, 1(2), 118–122.

(c) Using your independent data, determine which of these models has performed the best.

(d) Fit both a lasso and a LAR model to these data. Explain their results, and contrast their differences.

(e) Fit a PCA to these data, extract the most significant terms, and fit a LAR to these data. How different is this model from the one you fit in (d)? Why do you think that is, and which model do you prefer?

3. Many of the classical statistical methods we use today come from the brewing and distillation industries[7]. This dataset, which comes from a tasting tour of Scottish distillaries, was published by Lapointe & Legendre (1994; Applied Statistics, 237–257). You can find it on your course site as `scotch.csv`.

   (a) Load the whisky dataset into *R*. Make sure the data is correctly formatted for an analysis in *R* (not column names that are names, etc.)

   (b) The variable `spey` describes whether a whisky is, or is not, a "*speyside*" whisky. Run an SVM and, using training data, validate whether you can predict a whisky's type on the basis of its characteristics.

   (c) Train your model's γ parameter, and see if you do a better job.

   (d) How useful is this analysis? Is it something you could easily present to a policy-maker? Why (not)?

   (e) If you were to do another analysis on this data (don't!), what would it be and why?

4. These questions work with the `iris` dataset that you might well already be very familiar with, and you can load using `data(iris)`. If you've got 'iris-fatigue' then I would encourage you to run the exercises below using the Palmer Panguins dataset instead (see https://allisonhorst.github.io/palmerpenguins/); you will obviously have to make trivial changes to the example code but the dataset's website should help with that.

   (a) Perform an SVM on the `iris` dataset. Validate your model using training data.

   (b) Train your model's γ parameter. Does your model fit any better?

   (c) It's difficult to plot data with more than two dimensions, such as this `iris` dataset. See if you can figure out what the code below is doing, and do remember that you can always look at the help file for how to `plot` an `svm`. Make an informative plot of how your SVM is performing.

```
plot(model, iris, Sepal.Length ~ Sepal.Width, slice=
list(Petal.Width=median(iris$Petal.Width),Petal.Length=median(iris$Petal.Length))
)
```

[7] If you're interested, look up the history of "'Student's' t-test": https://en.wikipedia.org/wiki/William_Sealy_Gosset.

Part II

Artificial neural networks and deep learning

Chapter 5

Artificial Neural Networks

Overview

Now we're going to learn how to predict continuous data using *artificial neural networks*. These are perhaps the most over-hyped family of models I'm going to teach you, but they do have their uses and they are impressive to talk about with your friends. Try to remember, when using them, that there really is no definitive way to understand *how* these models work, and thus there is no definitive way (other than by testing their predictive power) to tell whether a particular model specification is the best choice. In my explanation of these models, I will focus on univariate regression; extending these principles to multiple (categorical) response variables is reasonably trivial. I will also explain, without using mathematics, the principles behind how basic artificial neural networks are fit to data, but these principles are not precisely the same as those used in the software you'll be using in *R*. The basic principles are, however, the same, and so I wouldn't worry too much if I were you.

5.1 Come with me if you want to live: a gentle introduction to artificial neural networks

Artificial neural networks are so-named because they are intended to mirror the structure of brains. They consist of an *input layer* where each node (often called neurons, but I prefer to use the term node) represents an explanatory variable, an *outer layer* of a single layer that represents predictions of the explanatory variables (we will only be working with a single response variable), and, most importantly, one or more *hidden layers* inbetween that form the basis of the model. If you would like to see a diagram of that, then hold tight: we'll be plotting them out in the next section.

These hidden layers contain nodes, each of which takes inputs from all of the nodes in previous layer: the input feeding into the first hidden layer, the first hidden layer feeding into a subsequent hidden layer if there is one, and then finally the output layer. Each connection from a node to another node has a *weight*, which determines how strongly the signal from a node passes to the other node. Each node also has an *activation function*, which determines how it handles inputs from other nodes.

These activation functions are extremely important, and they're the only reason that artificial neural networks are in any sense interesting. If the activation function were perfectly linear (*i.e.*, an input of strength 0.5 causes an input to the next node of 0.5) then an artificial neural network is mathematically identical to a standard regression. The weights are also important because they modulate the *magnitude* and *direction* (positive/negative) of each node's effect on each other node.

Fitting artificial neural networks to data is, frankly, a total and complete nightmare. The standard fitting approach is *backpropagation*: the output of the model is compared with the data (in a so-called *feedforward step*), and the error is propagated (spread) back through the network, with each node/neuron being altered depending on how much of the error is attributable to it. There's no magic to how it determines what error is attributable to each neuron: the network knows what the value should have been for each node, so simply finds the difference at each node. The weights are then altered—often not as much as they 'should be', since slowing the *learning rate* in this way yields more stable results—and the whole process is repeated. This is often called a *gradient descent* approach because, mathematically, it's like we're in a valley defined by our current set of coefficients and we're trying to slide down to the bottom where the best coefficients are.[1]

Finally, I have literally never known an artificial neural network to work unless all of the input data are normalized, including the response variable. This is because the fitting process requires error to evenly, but if the variables are not directly comparable this is essentially impossible and the whole thing falls apart.

5.2 Skynet begins to learn at a geometric rate; your models, not so much

Let's start by fitting an artificial neural network so simple that it doesn't even have any hidden layers. In fact, don't try adding them in, because it could crash *R*!

```
x <- rnorm(1000)
y <- -x
data <- data.frame(scale(cbind(x,y)))
library(neuralnet)
model <- neuralnet(y ~ x, hidden=0, data=data)
plot(model)
```

What you now see is an artificial neural network—a learning computer[2]. This one is so simple that it doesn't have any hidden layers: all you can see is the data going in (x), the *weight* that node/neuron has on the output (y), and the prediction flinging off to the right (the final arrow at the far right). You can also see a blue "1" flying in from the top: this is the function normalizing everything, which it has to do when you're trying to estimate an intercept and slope. It's literally just a nuisance term in the mathematics, so don't give it any thought.

[1] If you know what Newton's method for finding the minimum of a function is, let me save you some trouble: that's exactly what this whole thing is. It's not described like that on the Internet because most people don't seem to notice it, and academics don't simply use Newton's method because the likelihood surface is so strange that it often fails (all the interactions between layers mess things up). Don't believe everything you read on the Internet about artificial neural networks...

[2] Alright, that's the last Terminator reference, I promise.

Now let's simulate some more complicated data, and see how a more reasonably artificial neural network can deals with that.

```
explanatory <- data.frame(replicate(10, rnorm(400)))
names(explanatory) <- letters[1:10]
response <- with(explanatory, a*2 -0.5*b - i*j + exp(abs(c)))
data <- data.frame(scale(cbind(explanatory,response)))
```

Hopefully, you will agree with me that this is a pretty complicated problem. We've got two standard linear terms, an interaction, and a weird exponent of a non-negative number. So how is it going to perform?

```
training <- sample(nrow(data), nrow(data)/2)
model <- neuralnet(response~a+b+c+d+e+f+g+h+i+j,
  dat=data[training,], hidden=5)
cor.test(compute(model, data[-training,1:10])$net.result[,1],
  data$response[-training])
plot(model)
```

Very well! We've got a wonderful r^2, and the output looks very... well, confusing. Yes, you can see the hidden layer in there now, and you can see lots of things feeding through it, but what we can't tell is what's actually going on. You see, *everything* looks important in an artificial neural network, and that's sort of the problem. Just in the same way that we can't really tell why someone thinks something just by staring at their brain (yet), we can't necessarily tell what's going on just by staring at the coefficients in this model. Indeed, we've actually fitted so many coefficients (in the form of weights) that it's not even necessarily clear whether this is a very parsimonious model (which some people would say is that same thing as a good model). Help is at hand, however, in the form of the "Garson test", which is capable of figuring out what the most important explanatory variables are in a particular model.

```
library(NeuralNetTools)
garson(model, bar_plot=FALSE)
garson(model)
```

Two things strike us: firstly, this package was written by someone who loves `ggplot2` more than easy-to-use code, and so it's impossible to get both the plot and the data underlying the plot in the same function call. Secondly, by summing up the absolute magnitude of all the weights that connect explanatory variables to the (single) response variable, we can get some measure of relative importance of variables. This method only works for certain kinds of neural networks (*e.g.*, the ones I've taught you), and is obviously somewhat unhelpful in that it doesn't show you whether something is positively or negatively associated, or what the functional form of the association between a variable and the response is. If you want that, then you'll simply have to play around with dummy datasets, manipulating some variables while holding others constant, to see what's going on. This is also a good way to test model fit but, ultimately, demonstrates the most important thing to remember about artificial neural networks: they're really powerful, but we don't always know what they're doing.

5.3 Exercises

Please remember to scale your data before analysis using something like `data <- data.frame(scale(data))`. You can also get terms to include in your models with the following handy line of *R* code: `cat(names(data), sep="+")`. Complete *only one* of the following exercises. You will not receive extra credit for completing two of the following exercises.

1. The following exercises will make use of the simulated data from your *regression tree* practical. The dataset you will be working with will be the plant dataset I simulate for you in that handout. *Only fit your models with a training dataset of, at most, 500 samples*—this is simply for speed of calculation.

   (a) Fit an ANN with a single layer of 5 neurons to your training data. Validate that model using your validation data.

   (b) Does your ANN perform better or worse than a regression tree on the same data? Do you think this is a fair test?

   (c) Calculate the relative importance of each term in your ANN. Describe the results in, at most, a few short sentences.

   (d) Fit an ANN with two layers of neurons (you pick the number in each layer). Does it perform better, or worse, than your first model?

2. The data for this exercise come from a study of red wine quality (Cortez *et al.* 2009; Decision Support Systems, 47(4):547–553) and are available on your course website in `winequality-red.csv`.

   (a) Fit an ANN with a single layer neurons (you decide the number) to the data. Validate the model using independent data.

   (b) What are the most important, if any, predictors of wine quality?

   (c) How useful would this model be to wine producers versus restaurant owners? Why?

Chapter 6

Deep Learning and TensorFlow

Overview

This material follows neatly from our introduction artificial neural networks, and essentially forms a review of material in the context of a more advanced and flexible software package for model-fitting: TensorFlow. TensorFlow has emerged as the *de facto* platform of choice for this sort of work for a number of good reasons. The first is its ease of use and portability: TensorFlow started as Python library but has 'bindings' (sort-of life cross-language APIs—formal ways for different programming languages to share code) for languages as diverse as JavaScript and Ruby. The second reasons is its flexibility: TensorFlow can do almost anything you could want from a set of artificial neural networks, and can be extended to have whatever kind of nodes or activation functions you want with remarkably little code overhead. Third, and perhaps the most important part, it's developed by Google, and that name has a lot of cultural and engineering capital. Google often releases the last generation of their internal code as open source packages, and TensorFlow's blistering speed and scalability is a direct result of that. Deep learning is a field filled with buzz-words that is developing very fast indeed: you will find very few all-encompassing books to it, and most articles online are essentially shallow tutorials because the concepts themselves can be very difficult and are hidden in the primary literature. My aim with these two sessions is to give you a guide tour of what can be done with TensorFlow, focusing on fundamentals (this section) and image analysis (the next session). I am deliberately not covering the specifics of model fitting because, frankly, by the time you read this Google have likely totally changed everything to make it all even faster. This is only a good thing.

6.1 Tensors and their 'flow'

TensorFlow is fast because it views the world as a series of *tensors*. Thus I think a good place for us to begin is defining a tensor, and then using that definition to understand, fundamentally, how TensorFlow operates.

You may have heard of *scalars* and *vectors*. A scalar is what you might think of as a natural number: it's something that changes the overall magnitude of other numbers[1]. For example, the length of a rule is a scalar

[1]Please note that I am deliberately trying to be informal in my definitions for ease of presentation; if this informality is causing you trouble then you likely already know what a tensor is, so feel free to skip these two paragraphs.

(it's the 'scale' of the object), and the speed of a ruler as it flies through the air is a scalar. A vector has both a scale component and a single *direction*. Thus, for example, a ruler's velocity is a vector because it contains both a speed (speed alone would be a scalar) and a direction (this is the part that makes it a vector). Having a magnitude alone makes something a scalar, having a single direction makes something a vector. A tensor is a generalization of scalars and vectors to deal with cases where you have any number of directions. Thus a tensor of *rank 0* is the same thing as a scalar: it's a magnitude with no direction. A tensor of *rank 1* is a magnitude and a single direction. A tensor of *rank 2* is a magnitude and two directions.

Tensors become extremely important in university-level physics and engineering, because they allow us to formally examine concepts of stress and strain on a system. A tensor is essentially a way of describing how a system will be transformed, because a tensor will take any kind of input data and move them according to its magnitude and direction. For example, imagine you have a single row of information about two columns of data (x,y)—$(1,1)$, for example—to which a tensor is applied. That tensor is going to transform that data by moving it as far as that tensor's magnitude in whatever direction(s) the tensor specifies. Indeed, this may lead you to realize that such data was a vector all along: $(1,3)$ is the same thing as a vector that says "go up one unit along the x-axis and three along the y-axis". Just as how, once humans developed a formal definition of addition and multiplication, we were able to do much more complicated and useful things to numbers, now that we have a more formal definition of how transformations of data can take place, we can do more cool things. We can, for example, combine multiple transformations (tensors), see if there is redundant information in those transformations (reduce the rank of a tensor, called contraction), and model very complicated things.

TensorFlow is so-called because it views the entire world as a series of tensors. Tensors are written out in mathematics as tables[2] that specify the magnitude and direction of the tensor(s). To TensorFlow, there is no fundamental difference between data and calculation: everything is just a table[3]. This allows TensorFlow to do clever things like feed the results of computation back into data, distribute computation and data across multiple processors (if you enable GPU acceleration on your laptop, hundreds, but TensorFlow is used routinely across millions of processors), and rapidly calculate the output of artificial neural networks by 'simply' multiplying all the tensors together. Thus TensorFlow is absolutely obsessed about asking about the dimensionality of everything you load into it, because it has to create an appropriately-dimensioned set of tensors to handle all of that information. It means that TensorFlow can sometimes take a little while to initially compile (build) its models, but once it does they often can make predictions and be fit blisteringly fast.

6.2 Regression reloaded

The easiest way to see how TensorFlow works is to try it out. Let's start out by simulating essentially the same data from our first introductory class in artificial neural networks. Note that, because TensorFlow is much faster than what we were using before, I'm now adding some noise to the response variable to make things a little trickier, and I'm keeping my data in `matrix` format and not creating a `data.frame`.

```
# Simulate (this time adding noise to the response variable)
exp <- replicate(10, rnorm(400))
```

[2] If you're joining me from skipping the last two paragraphs—what's the difference between a table and a matrix among friends?...
[3] Technically a 'multi-dimensional array', but the meaning is just the same

```
resp <- exp[,1]*2 -0.5*exp[,2] - exp[,7]*exp[,8] + exp(abs(exp[,3])) + rnorm(nrow(exp))
# Scale data and making training subset
exp <- as.matrix(scale(exp)); resp <- as.numeric(scale(resp))
training <- sample(nrow(exp), nrow(exp)/2)
```

Great. To use TensorFlow, we're going to make use of a high-level wrapper[4] called *Keras*. Keras is the `ggplot2` of the artificial neural network world: it interfaces with a number of different packages, and is quite a bit easier to use than working with TensorFlow directly via *R*. To be honest with you, very few people either use Keras for things other than TensorFlow, or access TensorFlow through *R* using anything else[5], and essentially all the lower-level features of TensorFlow are available through Keras. Let's now build our model.

```
# Get Keras ready
library(keras)
install_keras()

# Specific model
model <- keras_model_sequential()
model %>%
  layer_dense(units = 15, activation = 'relu', input_shape=10) %>%
  layer_dense(units = 15, activation = 'relu') %>%
  layer_dense(units = 1)
```

The code above sets up an artificial neural network with two hidden layers, each with 15 nodes, and an output layer of a single node. Artificial neural networks of the kind we have covered so far are all *sequential*: data comes in, flows through the network in a straight-line without repeating back through any nodes, and then generates a prediction. Not all networks are like this, and so we have to tell Keras we want such a network in the first line above. Each *layer* in the node is a single line of code (generated using the `layer_dense` function), and we can build the network up piece-by-piece using *R*'s *pipe* operator (%>%). The pipe operator basically means "add the thing on the right on to whatever you're building on the left", and so you can see that we're modifying `model` (which is our network) by appending all these layers onto it. We specify how deep (how many nodes) each layer has with the `units` argument, specify the activation function with the `activation` argument, and the shape of its input with `input_shape`. Remember I said TensorFlow is obsessed with the dimensionality of our data because it's based around tensors? Our network has a *shape* of 10 because there are ten explanatory variables in our input data. That's where this number comes from: each 'pulse' of data coming through our network is ten pieces of data, and the resulting output need to be 1 thing (we have a single response variable). This is our data's format. Now we're ready to fit our model.

```
# Compile model
model %>% compile(
  loss = 'mean_squared_error',
  optimizer = optimizer_rmsprop(),
```

[4]'High-level' means you don't have to worry about how TensorFlow is programmed (unless you want to). *R* is a high-level language because you can use it on a Mac or a Windows computer and it works the same way, despite them both being fundamentally different kinds of computer. The low-level complexity is hidden from you.

[5]Not everyone, of course, and if you know someone who's an exception then I'm sure they have a great reason to do so.

```
  metrics = c('mean_squared_error')
)
# Train model with data
model %>% fit(exp[training,], resp[training], epoch=500)
```

TensorFlow models are fast because they are *compiled*: the computer takes the very flexible instructions within *R* and then turns them into inflexible computer instructions that can't be changed but can run much more quickly. To complete our model, TensorFlow needs to know how to tell when the model is doing poorly—it needs a definition of error so it can propagate that error back through the network. TensorFlow is much more sophisticated and flexible than the package we were using in our previous session, so we have a lot more options here and it can't just pick for us. I've specified this measured `loss` of predictive power as the mean squared error (just the same as what we used last time)—this is exactly the same definition of residual error as used in standard linear regression. I then ask TensorFlow to use a built-in optimizer—the way it propagates error throughout the network and trains the model—that is essentially the default option for squared error (`optimizer_rmsprop`). Notice that this optimizer is a function, and so you can write your own if you wish (...please don't in this class...), and this flexibility is one of the reasons I'm not going into the details of the fitting process for TensorFlow in this class. Finally, I ask it to report and record `metrics` on its fit to the training data, in this case the same as the one I'm asking it to use in the fitting process. Finally, I fit the model to data, using 500 training iterations, which as we discussed before are called *epochs*. As ever, let's validate out model in independent test data.

```
plot(predict(model, exp[-training,])[,1] ~ resp[-training])
cor.test(predict(model, exp[-training,])[,1], resp[-training])
```

I would suggest that the model has done quite well. So, let's finish up our tour of TensorFlow by unpacking some of the options it provides for different activation functions and node types.

6.3 Tweaking all the dials

Activation functions are often the special sauce of networks, and so it's worth knowing a little bit about the options you have available. Some common ones are:

- `linear`. The identity function: what you put in is exactly what you get out. Generally only useful if you want something that is one layer deep and is mathematically identical to a multiple regression (see previous session). Thus, if you find yourself using this, ask yourself whether you need to be using TensorFlow at all.
- `softmax`. This is often described online as if it's some sort of magic function that turns TensorFlow into either a Bayesian or a frequentist approach; this is not true. The softmax function is a generalization of the logistic function to deal with high-dimensional data: it takes multiple inputs and then smoothly rescales them to a number between 0 and 1. It's defined as $\frac{e^x}{\sum e_i^x}$ for a load of i input variables (x). If given ten input variables, it will exponentiate those input variables and divide them all by the sum of the exponents. It's *really useful* as the last step in a model where you're trying to predict whether the input

data is one of a number of things (*e.g.*, cute kitten, very cute kitten, or something else) because it will give what you can treat as predicted probabilities of each of those categories.

- `relu`. The *rectifier* function; there are many related functions. It's formally defined as $max(0,x)$, and it essentially means "throw away all the negative values and leave everything else unchanged". There are 'smooth' variants of it ($log(1+e^x)$), leaky variants of it (if $x > 0$ then x, otherwise $0.01x$), and they're all very important because they really do help models train faster. Small differences among these families can matter a great deal—use whichever one is generally recommended for your types of problems, or just experiment and see what happens.

There are also quite a few different options for loss functions:

- `mean_absolute_error`. This one's the absolute error, which we can't use so easily in standard statistical models, but is fair game in machine learning. Avoid if you like mapping things onto Normal distributions and the like, but use if you like simplicity.
- `logcosh`. $log(\cosh(error))$, which basically means "the same as the mean squared error but not so affected by outliers". Much more helpful than you might think, because if you knew what your outliers were ahead of time you might not need an artificial neural network!
- `kullback_leibler_divergence`. If you have taken an advanced class in model-averaging and AIC techniques, you might be excited to see this. If not, note that you can almost get a likelihood out of a network this way.
- `poisson`. I put this here as a reminder that you can essentially fit Generalized Linear Models in TensorFlow (if you've taken my class, note that).
- `categorical_crossentropy`. A useful option if you're doing some categorical models (there are other options for that in Keras). It's essentially $-\sum p_i$ where p_i is the probability ascribed to the correct label in your dataset. Thus if you had ten data-points and your model estimate a probability of 1 that each was the correct image, then your cross-entropy would be at its lowest possible value (-10). There is a correction of this for when you don't know the true value (*i.e.*, you're applying your model to new data).

We will be covering different kinds of network architecture next time, so I don't want to dwell on that now, but I do want to show you the very cool training dashboard that comes with Keras. If, while training, you tell Keras where to store log-files, you can visualize them later. Try running the following, for example:

```
model %>% fit(
  exp[training,], resp[training], epoch=500,
  callbacks = callback_tensorboard("folder/on/your/computer")
)
tensorboard("folder/on/your/computer")
```

Behold! A very fancy set of diagnostics appears on your computer about your run. Remember that you might need to clean out `"folder/on/your/computer"` at the end of a load of tests, and you need to replace that text with a folder on your computer. If the folder doesn't exist, it will be created. I should add that the line above sometimes causes an error on computers configured to be skeptical of programs setting up web servers and then

opening them in browsers; if so, open up a browser and trying going to http://127.0.0.1:4298 in it and, with a bit of luck, your dashboard should appear there.

6.4 Exercises

1. Today's data come from `boardgamegeek.com`[6]. They describe ratings given to board games by people on the Internet (`rating`), and information about each board game such as the `year` in which they were published, their `complexity` (how difficult they are to learn, rated out of 5), and various other variables that boring people such as myself find interesting. TensorFlow is capable of dealing with discrete/categorical variables, but to keep things simple I'm not including them in today's exercise.

```
# Load the data in like this (what does the row.names bit do?)
data <- read.csv("~/Desktop/boardgames-continuous.csv", row.names=1, as.is=TRUE)
# Plot the data out like this
with(data, plot(rating ~ year))
# Format the data like this
response <- data$rating
explanatory <- as.matrix(data[,-1])
# Get the dimensions of the data
ncol(explanatory)
# Don't forget to scale them!...
# If you have trouble with rownames, you can get rid of them like this
rownames(explanatory) <- NULL
```

   (a) Train a deep learning network with at least two hidden layers on these data. Use the same activation function throughout the network.

   (b) Validate the data using independent data. How well does the model perform?

   (c) Use a different activation function in one, or all, of the layers. Assess, using independent data, whether this difference makes the model perform better.

[6] Downloaded using code from https://github.com/willpearse/boardgamegeek—you can only imagine how exciting I am in my free time.

Chapter 7

Convolutional Neural Networks

Overview

Now we are launching into convolutional neural networks. These are an extremely popular class of neural networks whose topology was designed with the mammalian visual cortex in mind. They're extremely powerful tools, and have the additional advantage that they can carry out pre-processing of images on our behalf. This means that, with sufficiently well-designed networks, you can simply whack your images into the model and the model will down-scale and clean them up for you. I'm sure you can imagine how useful having the computer do the work for you can be!

7.1 Basic image classification

Let's start off by working with a built-in dataset within Keras and classifying the data using the techniques we've already learned. First of all, let's download and clean up the demonstration data. It's roughly 100 Mb in size, so this may take a moment.

```
# Setup
library(keras)
install_keras()

# Load data
raw.data <- dataset_fashion_mnist()
resp <- raw.data$train$y
exp <- raw.data$train$x

# Do a bit of renaming and scaling, and plot for sense
lookup <- c("T-shirt/top", "Trouser", "Pullover", "Dress",
  "Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot")
exp <- exp / 255
par(mfrow=c(5,5))
```

```
for(i in seq_len(5*5))
image(t(exp[i,28:1,]), main=lookup[resp[i]+1], col=grey.colors(255))
# ... the images are upside-down, hence the 28:1 statemnt,
# ... and R needs matrices rotated to plot, hence the use of *t*ranspose
# ... and keras needs labels (resp) to start at 0, hence +1
```

Hopefully you should now see 25 images of clothing from a standard machine learning dataset on articles of clothing. If it seems odd that this is a standard test dataset, take a minute to consider that almost everyone you have ever met owns at least one pair of clothes, so we have a lot of data and a lot of motivation to make good models of what people wear! Let's now fit a standard artificial neural network to these data, making use of two new kinds of layer (see if you can spot them before I describe them).

```
# Define model
model <- keras_model_sequential()
model %>%
  layer_flatten(input_shape = c(28, 28)) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 10, activation = 'softmax')

# Compile model
model %>% compile(
  optimizer = 'adam',
  loss = 'sparse_categorical_crossentropy',
  metrics = c('accuracy')
)

# Fit model and independently validate
model %>% fit(exp, resp, epochs = 5)
test.resp <- raw.data$test$y
test.exp <- raw.data$test$x
test.exp <- test.exp/255
model %>% evaluate(test.exp, test.resp)
predictions <- model %>% predict(test.exp)
table(apply(predictions, 1, which.max)-1, test.resp)
```

The two new kinds of layer are the *flattening* layer (`layer_flatten`) and the *dropout* layer (`layer_dropout`). The flattening layer is more of a processing layer than anything: it takes the two-dimensional input data (the images) that we have supplied and turns them into one-dimensional vectors. Notice that it specifies its input shape as a 28-by-28 object (`input_shape=c(28,28)`)—we have flattened two dimensions of data into one. You have only been working with such data in this class so far. The dropout layer is a weird idea that can often work well in neural networks. The idea is to randomly drop nodes from the network during each training epoch with a set probability for each node. Thus the network topology is different during each training session. Then,

when it comes to using the network on real (or validation) data, all the nodes are brought back into the network but each node's 'downstream' (subtending) connection weights are multiplied by the probability of that node being in the network during training. This tends to sort of 'damp down' the network, and reduces the likelihood of overfitting. Conceptually, this approach works for reasons very similar to the *random regression trees* we covered a few sessions ago: we're playing around with the variance within the network. We are using a new optimizer because we are dealing with a different kind of data (it is beyond the scope of this course to explain this optimizer, as I described last time), and the new loss function we covered last time.

7.2 Convolutional neural networks

Dealing with images as flat data is fine, but conceptually it feels odd to throw away what might important data as to what pixels are close to one-another. A modern class of networks—*convolutional neural networks*—are named after the mathematical operation they use (a *convolution*) to emulate the operation of the visual cortex in mammals. To explain these networks, I'm going to (very briefly!) outline the basics of vision in humans, then describe what a mathematical convolution is, and finally show you the network structure of these models.

We are still learning about the human visual cortex, but it is fair to describe it as a series of layers that take input from the eyes and apply a series of *detectors* to process that input into a form that is more useful to us. For example, we have layers whose job is to detect edges, and others whose job is to detect movement in particular directions. The 'Waterfall Illusion', which was first described by Aristotle, is a direct consequence of these detectors: if you stare at a waterfall for a long time, it tires (saturates) the downward detectors in your visual cortex. When you then look away from the waterfall, because those downward detectors have saturated they fire less intensely than they would normally, and are weaker than the upward detectors they normally counter-balance when we look at something at rest[1]. Thus when you look away from the waterfall, suddenly everything seems to be rising[2]. Most detectors work to generate an image from whatever the eye is seeing at that moment in time, and as such take input from several adjacent regions of the retina and combine that information in order to detect a particular kind of feature. For example, edge detectors take input from several regions (let's call them cells) and compare them to see if some are lighter than others. If they are, then it sends a signal that an edge exists at that point in the visual field.

In biology, neurons are cheap[3], but in computing, nodes are not. So when computer scientists wanted to use similar techniques to analyze images in artificial neural networks, they hit a snag: in our brains there are almost as many detectors as there are inputting rods and cones in our eyes. That makes sense: each 'pixel' in our vision must be compared, and so each 'pixel' needs a receptor. Each layer of our visual cortex is massive because of this. Having that many inputs would become unwieldy in machine learning: we would need thousands of nodes just to model a single reasonably-sized image. The solution is to cheat, and use *convolution* to set up only a single receptor (a convolution layer) and then apply that convolution layer across all the possible subsets of the

[1] All neurons within your brain fire at a background rate; it is deviations from this background rate that convey information.

[2] You don't believe me, do you? It is possible to make your vision almost buckle by staring at this video (https://upload.wikimedia.org/wikipedia/commons/d/d3/Illusion_movie.ogv) for about twenty seconds and then looking away. There are versions of these kinds of effects that can alter your vision for some time; this isn't one of them, but after one very irritating experiment as an undergraduate I perceived musical staves as a very pale light-green for a month or so.

[3] Well, they're not super cheap, but let's ignore that for a moment

image to get the same effect done. Thus, using convolution, it's possible to have a single detector that examines 5x5 pixel sections across an entire 1000x1000 image, using only 5x5 nodes.

Which means we need a quick note on what, exactly, a convolution is... Technically, if you have two functions you are convolving, called f and g, their convolution is the integral of the product of the two functions once one has been reversed and shifted by a set amount. In scary equations, therefore, it's:

$$(f * g)(x) = \int f(x)g(-x)dx \tag{7.1}$$

Where $*$ means "convolve". This ability to essentially multiply things together is what allows the convolution to be performed across so many different inputs in succession: we're convolving everything together. The confusing part is what x represents, and why $f(x)$ is working on the reversed set of data from $g(-x)$. x represents being fit across all parts of the image in our case, and so is the application of the detector to the various parts of the image (first the top-left, then the top-middle, then the top-right, then the middle-left, then the...). The reversal is often described as being in order to make the whole operation commutative [$f(x) * g(x) = g(x) * f(x)$, which is not always the case in math], which is reassuring and everything but doesn't always seem to make the reasoning much clearer... Remember that x represents the convolution happening, and the convolution is basically multiplying all the parts of the functions together. Think of the two functions as curves that are being multiplied by each other when they cross over, and then think of the 'crossing over' as the two curves being dragged past each other and then the convolution being the multiplication of the points that line up. As f is 'dragged' to the right, we are 'dragging' g to the left, and so by flipping g we are actually preserving the ordering of the functions. Sadly, this is something that is difficult to describe, but straightforward to draw, so pay attention in my lecture if this doesn't make much sense to read[4].

7.3 Hands-on with convolution and network design

OK, so let's go right ahead and fit a convolutional neural network in Keras.

```
# Model specification
conv <- keras_model_sequential() %>%
  layer_conv_2d(filters = 20, kernel_size = c(3,3), activation = 'relu',
                input_shape = c(28,28,1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  #layer_dropout(rate = 0.25) %>%
  layer_flatten() %>%
  layer_dense(units = 20, activation = 'relu') %>%
  layer_dense(units = 10, activation = 'softmax') %>%
  compile(
    optimizer = 'adam',
    loss = 'sparse_categorical_crossentropy',
```

[4]Or read the above, carefully, and then watch the videos in the 'Visual explanation' at https://en.wikipedia.org/wiki/Convolution#Visual_explanation

```
    metrics = c('accuracy')
  )

# Re-arrange data and fit model
array.exp <- array(exp, dim=c(dim(exp), 1))
conv %>% fit(array.exp, resp, epochs = 10)

# (If you want to subset the data a bit more, try)
subset.exp <- exp[1:500,,]
s.array.exp <- array(subset.exp, dim=c(dim(subset.exp), 1))
conv %>% fit(s.array.exp, resp[1:500], epochs = 10)
# Much faster!
```

A few things to note before we go into the specifics of convolutional networks. First, you know how to validate models, so I'm not going to bother showing you that again. If you test this, you should find it performs better than the previous model (albeit taking more time to train; more on that in a moment). Second, we're experienced with Keras now, so I don't need to split my model specification across multiple lines: you can see that it's possible to define the model without multiple <- calls, and so I'm doing it all in one. You will see people online writing functions to contain their model definitions; this apparently makes them feel clever but is unnecessary. Thirdly, because we now are dealing with images, I have to restructure the data a little bit. Keras comes with built-in functions to do this (`array_reshape`) but there's no need to use these if you're careful. Critically, however, we have made our data four-dimensional: the fourth dimension is how many *channels* of color there are in our image. Because ours is one channel (black–grey–white), this extra dimension contains a single column, but you can fit these models with multiple colors if you wish. You will have to do this to your validation data as well (you should be able to do this yourself using my code).

`layer_conv_2d` does the convolution work for us here. We specify the number of times we want to apply the detector to the image via `filters`; I've picked 20 arbitrarily, and if you pick too many or too few for it to work neatly then Keras will 'pad' your output for you. We also have to specify the input shape of the image (28x28, with one channel), and specify how big our detector ('kernel' in Keras-speak) is going to be. In this case... 3x3 images. This is it: this is the whole shebang. As the name implies, you can do 3D convolutions (videos; consider the Waterfall Illusion I discussed above), and one-dimensional convolutions which are useful in classic time-series-type analyses.

`layer_max_pooling_2d` is amazing. It downscales whatever it's given to reduce its dimensionality: it takes 2-by-2 cells (defined in the layer), and compresses them (averages them) to output a single cell for each of them. This is wonderful, because it allows us to do the processing of images within our neural network. In more advanced cases, you can even have the degree of processing be determined by the network itself, such that your model will figure out what needs to be done to your data in order to get the best result[5].

As you might have noticed, I've left an additional layers in the code above commented-out. This is to help you with your exercises below; ignore them for now.

[5]If this seems somehow 'naughty', recall that all modeling really involves is the selection of transformations to your data that best explain it...

7.4 Freezing networks and shattering images

A really neat feature of the %>% operator adding on to whatever has happened before is you can continue training your model from wherever you left off last. Try it now, if you wish, by providing additional training data in the form of the reduced subset code I give at the end of the example. This allows you to, for example, train a model in small batches until it's good enough (defined however you wish) for whatever you're trying to do. It is even possible to train part of a neural network, then plug those trained components into an even larger network. When doing so, the weights will carry over to the new network, and training will begin with those as starting points.

```
new <- keras_model_sequential() %>%
  conv() %>%
  layer_dense(units = 10, activation = "softmax") %>%
    compile(
      optimizer = 'adam',
      loss = 'sparse_categorical_crossentropy',
      metrics = c('accuracy')
    )

new %>% fit(s.array.exp, resp[1:500], epochs = 10)
```

Re-using things is great and all, but the trouble is the training process doesn't necessarily get much quicker with these additions because, even though the starting point is a little different, there are still a lot of connections to train and they interact with the connections that are already trained. A better approach is to *freeze* a neural network, or part of a neural network, and then combine it with something else. That way its weights can't be updated, making it faster to train. This might seem a bit weird, but it's super useful if you're working with some data the kind of which has been seen before. Say, for example, that you are trying to classify some faces. If someone has already built a neural network that does face classification (Google, for example), then you can take their network, leave it as-is, and then bolt some additional nodes at the bottom. It isn't ideal, but if Google trained a network of over a million nodes on a dataset of thousands of faces, it's going to be better than anything you could hope to do yourself. You can also, of course, selectively *unfreeze* certain layers if you want. The example below shows this, although I must emphasize that we're gaining very little by doing all this freezing and unfreezing in such a trivial example.

```
# Freeze the whole thing
freeze_weights(conv)
# ...or...
# Look at the model to figure out the layer names
conv
# Selectively unfreeze the last layer (your layer name will vary slightly)
unfreeze_weights(conv, from="dense_2")
# Re-compile the model (you must do this before use after any freezing or unfreezing)
 new %>% compile(
```

```
    optimizer = 'adam',
    loss = 'sparse_categorical_crossentropy',
    metrics = c('accuracy')
    )
# ...if you've taken my programming class, you might be surprised that we don't have to
# ...re-compile conv here. Remember TensorFlow is a Python library - call by name
# ...is going on here, and so freeze_weights has done the work there for us already
```

I want to leave you with a note on variance. Back in the regression tree exercises, we discussed how a good way to deal with over-fitting is to *bootstrap* your data. Bagged trees and the like use random subsets of the data to add variance to the training set, and in-so-doing, and rather counter-intuitively, make the training process much more powerful. The same is true of neural networks[6], and in image analysis is often given very fancy names like *augmenting*. Indeed, it's excellent because it reduces (if you're clever about it, to zero) the number of times a network is trained with exactly the same data. There are built-in ways of doing this within Keras (look up `image_data_generator`), but it's also possible to do it yourself and I think it serves as an excellent demonstration of how you can go a long way with a little thought. Before I show you how below, think for a moment about how, given everything you know, you might write some code that would introduce a little noise in your training subset. Make sure to unfreeze your layers first...

```
# Let's do 20 training epochs
for(i in 1:20){
  # Randomly alter the training data
  augmented <- array.exp + rnorm(length(as.numeric(exp)), sd=.1)
  # Train once on new data
  conv %>% fit(augmented, resp, epochs=1)
}
```

Does your model fit any better now? Is there something I could do a little differently? I might have guess**sd sd**omething wrong, maybe?

7.5 Exercises

1. Today's data come from a classic handwriting dataset that comes bundled with Keras. There are many guides online as to how to load this dataset, but I should add that it's in precisely the same structure as the data I have been using in this session. So please do go ahead and read those guides, because you might just learn something new!

```
library(keras)
mnist <- dataset_mnist()
x_train <- mnist$train$x
y_train <- mnist$train$y
x_test <- mnist$test$x
y_test <- mnist$test$y
```

[6] I might argue both artificial and real!

(a) Train a "flat" deep learning network with at least two hidden layers on these data. These are handwritten digits (the numbers 0–9), so set up your output layer accordingly.

(b) Fit a convolutional neural network to the data of the same hidden structure as in my example code above.

(c) Add, subtract, and/or alter hidden layers in order to get your model to fit faster than it does in part (b) above.

(d) Which of the two models, (b) or (c), do you think is the best fit to your data? Things to think about: if your model fits faster, does it matter if its quality increases more slowly across epochs?...

Chapter 8

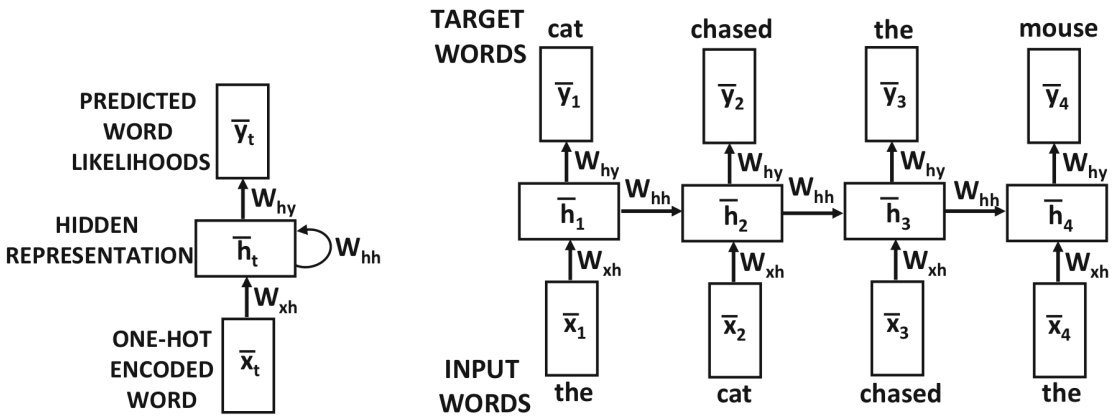
Recurrent neural networks

Overview

Up until now, we have covered the kinds of problems that deep learning can tackle that can be formulated in ways that we could imagine traditional statistical models working. Even image analysis is an example of this—we are essentially taking a set of input variables (pixel values within an image) and predicting a binary variable (*e.g.*, cat/dog) on the basis of those explanatory variables. Recurrent neural networks allow us to go further, and consider explanatory 'variables' such as input sentences (*e.g.*, "where is the rain in Spain?") and response 'variables' that are themselves sentences (*e.g.*, "mainly on the plain"). To describe such data in traditional statistical terms—time-series data—is technically correct but misses the true magnitude of difference. Today we will be dipping our toes into recurrent neural networks—networks architectures that are capable of feeding back into themselves—in order to be able to fit deep networks to time-series data. These kinds of models are tremendously powerful, but are also tremendously difficult to get right, and so you should view this session as more of your first steps into the ocean of deep networks. It is important to remember, in reading around this literature, that such recurrent architectures are technically 'Turing complete' and as such you will read a lot of pontificating on the Internet about what this means for the future of artificial life. I would encourage you to be sceptical of the idea that a model you can write in a few minutes on a laptop can become sentient...

8.0.1 Time-series data and neural networks

A time-series dataset is one where there is an inherent temporal order to the data. Stock prices are a common example: each 'row' in such a dataset represent the price of a stock at a given time, and as we move from the top to the bottom of the dataset we are moving forwards in time. It is not always the case that the intervals in time between the entries need to be the same, but for our purposes we will assume they are. This makes it easy to indexed time-series for our convenience; a variable x is measured at a time t, such that x_{t+1} was measured one unit of time (an hour, a day, whatever convention we define) after x_t. Such data could be handled in a very natural way with the network architectures you already know: we could have an input node for each time-point in a time-series dataset, such that a stock dataset of 100 days of prices would have 100 input nodes, and then have all that information feed forward into whatever kind of architecture you wanted. You could even design such a network such that the linkages between the layers were incomplete; perhaps every x_t is linked to nodes



(a) RNN (b) Time-layered representation of (a)

Fig. 8.1 **Overview of the structure of a recurrent neural network.** The goal of this network is to take some input word data and predict the next word in the sentence. The complete sentence that is being predicted is "the cat chased the mouse". On the left (a) we see the 'true' form of a recurrent neural network, in this case a vector of data representing words (bottom, x_t) feeding into a hidden layer which can feed back into itself to affect the next time-step (mediated by weights, w), the hidden layer itself then outputting a predicted next work (y). On the right (b), we have 'expanded out' the recurrent neural network to see how the network would operate. Note that we can now see all the of input words (our explanatory variables), how they modify the hidden layer, and how that hidden layer itself feeds forward to inform the inference of the next word. These two representations of a neural network are the same, but in (b) we have 'unrolled' the network to show it in a form that is perhaps more familiar and makes the expect input/output dimensions more explicit. Taken from Aggarawal 'Neural Networks and Deep Learning' (2018; Springer).

directly downstream from x_{t+1} and x_{t+2}, but x_{t+1} is not so-linked to x_t (*i.e.*, information can move forward in time but not backwards).

The problem with such architectures is they become huge very quickly, and are extremely inefficient because there is no obvious way to generalise very simple properties in time-series analysis such as a constant temporal lag across observations. In the example I gave earlier, consider that the network would have to learn weights for each possible temporal lag ($t+1$, $t+2$, etc.) for each time-point separately: there is no way to share the weights of connections across spacings. There are many possible solutions to this problem, but the most popular is the *recurrent neural network*, where nodes are allowed to feed back into the network. An example of this is shown in figure 8.1, where we see the structure of the network compressed and also 'rolled out'. In a recurrent neural network the state of the network for a given time-step is allowed to propagate forward to affect the state of variables in later time-step(s). You can think of this as allowing the network to have a form of memory: the signal sent from a node at one point in time can affect future states of nodes.

There are a few practical considerations for such recurrent networks; many of them involve problems that crop up a lot with language translation that I am introducing you to in order to allow you to read the literature widely, not because I am hoping you will become linguists. The first is whether we want them to 'look' ahead or to the past. While you might think that only allowing a model to propagate information forward is best (because otherwise you are 'cheating' and going backwards in time), remember that not all time-series dataset involve things that are moving through time. For example, sentences (see figure 8.1) are often treated as time-series datasets, and in networks designed to translate from one language to another it is quite common to have separate

parts of the network that look forward through the sentence and that look backward. The second problem is how to encode words so that the network can work with them. Perhaps your first thought is to make a factor variable with one level per word; thus, for example, the sentence "Will gave Will a gift" could be encoded as a numeric sequence "1 2 1 3 4" (1='Will', 2='gave', 3='a', 4='gift' [1]). Such one-hot encoding works well apart from when you have extremely large vocabularies (consider the problems inherent in "Will" vs. "Sam" and "gave" vs. "lent") and grammar (consider "gave" vs. "will give"), and there is a vast literature on hierarchical data structures to try and account for such problems. Finally, there is also the problem of how to predict when a sentence (or paragraph, or time series more generally) will end. Again, the trivial and obvious solution (a special terminator signal, much like the `NULL` terminators used in strings) works well, but it creates problems when the model must predict forward or backwards from such a terminator. For example, special care must be given to the training of models that incorporate full stops—what should the model output as the most likely word to follow a full stop? What about a question mark?... We will happily ignore such problems by focusing on more traditional time-series datasets!

8.0.2 Hands-on with recurrent neural networks

Alright, let's just get on with it shall we? Let's fit a very simple RNN to a single, simulated time series dataset.

```
# Get Keras ready
library(kerasR)
install_keras()

# Simulate a time series (y) with first-order autocorrelation
y <- rep(0, 101)
for(i in seq(2, length(y)))
  y[i] <- y[i-1] + rnorm(1)
y <- as.numeric(scale(y))
# ... why are we scaling the variable?

# Make a predictor variable that is the previous time-step (x)
#   and then split into training/test data
x <- y[1:100]
x_train <- array(t(matrix(x[1:50], 5, 10)), dim=c(10,5,1))
x_test <- array(t(matrix(x[51:100], 5, 10)), dim=c(10,5,1))
y_train <- y[seq(6,51, by=5)]
y_test <- y[seq(56,101, by=5)]
# ... note we are using arrays, grouping our data into runs of
#     5 sequential points, with 10 training/test replicates,
#     and an overall dimension of 1 (i.e., a single variable)

# Build the model itself - spot the RNN layer!...
```

[1] Perhaps a rather lonely example, but you get my point...

```
model <- keras_model_sequential() %>%
  layer_dense(input_shape=c(5,1), units=5) %>%
  layer_simple_rnn(units=5) %>%
  layer_dense(units=1)

# Train the model
model %>%
  compile(
    loss = "mean_squared_error",
    optimizer = optimizer_rmsprop(),
    metrics = list("mean_squared_error")
  )

# Let store the training data so that we can...
history <- model %>% fit(
  x_train, y_train, epochs = 500
)
# ...plot it out, along with model predictions
plot(history)
plot(predict(model, x_test)[,1] ~ y_test)
```

I don't want you to focus too much on the problem of how the data were simulated yet, other than to note that we had to scale that data because that will be relevant for the next section. For now, focus on the dimensionality of the data: we have a time series of length 101 (y), which we cut into training/validation subsets in runs of 5 (making 10 such runs in the training and subset data). This allows us to give our data to TensorFlow in a very specific form: an `array` with the different training runs (10 of them, each of length 5) grouped up. Take a look at `dim(x_training)` and `dim(y_training)` until you're 100% clear on what is going on: our response variable is a 1-dimensional variable of length 10, and our training data is a 3D array with 10 runs, each of length 5. Note the use of 1, 5, and 10 in the neural network structure itself: we have to be crystal clear with TensorFlow what we're giving it, because otherwise it doesn't know how to recurse from the explanatory input variables onto the response variables. Once the data is in the right format, the actual recursion itself is trivial—it's 'just another layer' and we specify the dimension of the recursion itself.

8.0.3 Exploding/vanishing gradients (and recurrency)

That's the end of the session, right? We're done! Sadly, no. Because recurrent networks involve so much propagation of signal throughout their network, they tend to suffer very badly from the *exploding* or *vanishing gradient problem*. You've probably not thought about it very much before, but if you keep multiplying numbers by themselves they tend to get very large or very small very quickly (compare 10^{10} with 0.1^{10}). Neural networks with many layers (*i.e.*, deep networks) are basically just lots of multiplications all together, and it means that signals can get very large as they propagate through that network. Things like bias neurons can help with this to

some extent [2], but in the face of very deep networks—or recurrent networks where signals can propagate across time-series—they cannot keep up. This creates a real problem because the signals passing through networks become so large that we can't effectively train them anymore. Because the multiplication of the weights makes everything very large or very small, it becomes effectively impossibly to see the impact that quantitative changes in the weights have on overall model performance [3]. Thus we can't fit the network anymore, and out predictions are uniformly awful.

The band-aid solution to this is to scale your input data, as I did in the last code example, but that can only do so much. Another option, which we will not explore here in detail but is worth knowing about, is to allow inputs to enter at different points within the network. So-called *skip layers* allow for the outputs from particular nodes to go around other layers in the network. Such structures are actually reasonably common in mammalian brains, where external stimuli (*e.g.*, retinal cells) are allowed to directly innervate neurons deep within our cortex (often in several places). This perhaps somewhat alleviates these problems by re-introducing the original signal to ensure that it doesn't get lost in all the noise. Of course it's not entirely clear that back-propagating artificial neural network problems back onto real neural networks is a valid approach. I mention all of this simply to remind you that the kind of linear, sequential network structures we are exploring in this course are not the only kinds of networks that can be fit.

A popular solution are so-called *Long Short-Term Memory* layers (LSTM), which are a much better idea than their totally nonsensical name implies. Their goal is to obviate this problem by dampening down the propagation of information throughout the network: creating a form of short-term memory that can be propagated back into the network, modified through time, and of course completely reset. I would advise you to ignore any diagrams you find that attempt to explain what they 'look like' because they are really more akin to memory registers in a computer and thus don't really 'look' like conventional nodes in a network. Just as with circuit-boards, they have 'gates': *input*, *forget*, *output*, and *new cell state* gates. Think of these gates as ways of accessing the value (the memory) of the LSTM node itself: depending on how strongly they are activated these gates 'open' and perform an operation on the LSTM node itself. If these gates are sufficiently innervated (they received input from other nodes), they perform their particular function: the input gate will add value to the node's value, the forget gate will wipe the value (memory) clear, and the output gate will cause the node to propagate its value on to its connected nodes (just as if it were a regular node in a regular layer). The 'new cell state' gate is somewhat special and gives the node its initial value. In some cases the node's cell state (its value, the memory itself) is something that you can inspect and derive meaning from. A trivial example is that in networks associated with language production such a node may contain an encoded value that is the subject of a sentence ('1' for 'Will' in the example I gave earlier in this section). There are simplified versions of the LSTM gate ('GRU' gates) but, frankly, they're not much more simplified so I think you might as well wrestle with the LSTM gate and be done with it.

The good news is that fitting an LSTM layer is very straightforward in TensorFlow. I leave it as an exercise to you, dear reader, to replace `layer_simple_rnn` with `layer_lstm` and see what happens. Depending on the parameters you pass to the LSTM layer you may be able to obviate the need for rescaling of your

[2] remember those from the first section?

[3] (Advanced side-note): This is the neural network equivalent of problems exploring a very fine-grained likelihood space in maximum likelihood-estimation, or 'divergent transitions' in Hamiltonian MCMC in Bayesian problems.

input variables (why not take this opportunity to look at the fantastic documentation the Keras API provides; https://keras.io/api/layers/recurrent_layers/lstm/).

8.0.4 Autoencoders and the 'secret' of deep learning

I thought it would be nice to finish where we started: dimensionality reduction techniques. Neural networks are so flexible that, for some architectures, they can be shown mathematically to be identical to classical methods. In our first session I taught you about PCA, but didn't cover factor analysis—an approach that is similar to PCA but where you choose the number of principal axes to estimate[4]. *Autoencoding* networks are notable because they use neural networks to build reduced-dimensionality representations of their input data, and as such their hidden layers encode a representation of the input data but using fewer dimensions. They always have the same dimensionality going in as going out, because they are trained to replicate their input data. Below is an example of how to build one, and also how to extract that reduced-dimensionality representation.

```
# Simulate data and verify they have one major axis
pc1 <- rnorm(100)
xs <- replicate(10, pc1+rnorm(100, sd=.2))
biplot(prcomp(xs, scale=TRUE))

# Build our autoencoder (from scratch)
# - Note I have labelled the layer with the encoder
autoenc <- keras_model_sequential() %>%
  layer_dense(input_shape=10, units=5) %>%
  layer_dense(units=1, name="auto-encoder-layer") %>%
  layer_dense(units=5) %>%
  layer_dense(units=10)

# Fit autoencoder
autoenc %>%
  compile(
    loss = "mean_squared_error",
    optimizer = optimizer_rmsprop(),
    metrics = list("mean_squared_error")
  ) %>%
fit(xs, xs, epochs=100)

# Extract the output from the autoencoder layer
out.autoenc <- get_layer(autoenc, "auto-encoder-layer")$output
t.autoenc <- keras_model(inputs=autoenc$input, outputs=out.autoenc)
est.pc1 <- predict(t.autoenc, xs)
```

[4]Or 'factors' as they are called in such approaches. Bear in mind my aim here is not to teach you factor analysis (...what we will be doing isn't formally factor analysis, in fact), but rather to show you how to make use of what is the real power of deep learning.

```
# Verify that we've found the major axis
plot(est.pc1 ~ pc1)
cor.test(est.pc1, pc1)
```

As you can see, by fitting a network that tapers to a single layer, and where we are minimising the difference between the input and an identical output, we have encoded a reduced dimensionality representation of the underlying data (*i.e.*, the underlying major axis of the data) inside the network itself.

What is perhaps the most surprising thing about this network is that it performs well *by being simple*. Although it is tempting, when first starting to get to grips with deep learning, to fit networks with many layers, each layer dense with nodes, the truth is that doing so misses the true underlying power of deep learning. Deep learning is powerful *because successive layers of relatively few nodes force the network to simply encode the properties of your data*. Multiple, successive layers are flexible because they represent multiple, successive applications of complex mathematical transformations on data. Yet having fewer nodes within each layer forces the network to compress signal and extract higher-level features of the data within the network. Equally, having too many layers leaves you open to gradient problems in training, making it difficult for the network to learn finer features of your data. Thus the expert user of neural networks knows that the best performance comes from having a handful of layers, each with relatively few nodes.

8.1 Exercises

1. Today we have covered the most advanced topics in the course, and so if you want to develop the ability to work with such data I'm afraid you're going to have to experiment a little on your own. Below I give an example of a stock-market analysis you could perform using Keras, but I encourage you to follow your own interests. Bear in mind that many of these questions are harder than they seem; do not be dispirited and experiment with different architectures. The code below will load some stock data into *R* for you.

   ```
   library(tidyquant)
   apple <- getSymbols("AAPL", from="2018-01-01", to="2018-12-31")
   ```

   (a) Train a standard (*i.e.*, not recurrent) neural network to predict a stock's value on the basis of the stock values for the past five days.

   (b) Fit a recurrent neural network that will do the same task as in (a).

   (c) Fit a recurrent neural network that will predict stock values for the next week on the basis of the last week's worth of stock values.

   (d) Fit a recurrent neural network that will predict six months of stock values on the basis of the last six months. (Note: this is harder than it sounds and will require careful thought as to how to prepare your training data, as well as needing more training data).

   (e) Fit a recurrent network to multiple stocks over whatever time period you choose. Note that this is a multivariate time-series analysis, and so will require you to use something like the `time_distributed` layer (https://keras.io/api/layers/recurrent_layers/time_distributed/)—read the documentation for this layer carefully and consider this a challenge exercise.

(f) Improve your answer to (e) above using an autoencoder layer, and explore the behaviour of its reduced representation through time. By carefully choosing input stocks, develop an accurate model of the stock market and then send it to me[5].

[5]This last part is a joke...