Contents

CHAPTER 1

Some Preliminaries

1.1 NOTATION AND CONVENTIONS

A dataset as a collection of d-tuples (a d-tuple is an ordered list of d elements). Tuples differ from vectors, because we can always add and subtract vectors, but we cannot necessarily add or subtract tuples. There are always N items in any dataset. There are always d elements in each tuple in a dataset. The number of elements will be the same for every tuple in any given tuple. Sometimes we may not know the value of some elements in some tuples.

We use the same notation for a tuple and for a vector. Most of our data will be vectors. We write a vector in bold, so $\mathbf{x}$ could represent a vector or a tuple (the context will make it obvious which is intended).

The entire data set is $\{\mathbf{x}\}$. When we need to refer to the i'th data item, we write $\mathbf{x}_i$. Assume we have N data items, and we wish to make a new dataset out of them; we write the dataset made out of these items as $\{\mathbf{x}_i\}$ (the i is to suggest you are taking a set of items and making a dataset out of them). If we need to refer to the j'th component of a vector $\mathbf{x}_i$, we will write $x_i^{(j)}$ (notice this isn't in bold, because it is a component not a vector, and the j is in parentheses because it isn't a power). Vectors are always column vectors.

When I write $\{kx\}$, I mean the dataset created by taking each element of the dataset $\{x\}$ and multiplying by k; and when I write $\{x + c\}$, I mean the dataset created by taking each element of the dataset $\{x\}$ and adding c.

Terms:

- $\mathsf{mean}\,(\{x\})$ is the mean of the dataset $\{x\}$
- $\mathsf{std}\,(\{x\})$ is the standard deviation of the dataset $\{x\}$
- $\mathsf{var}\,(\{x\})$ is the variance of the dataset $\{x\}$
- $\mathsf{median}\,(\{x\})$ is the standard deviation of the dataset $\{x\}$
- $\mathsf{percentile}(\{x\}, k)$ is the $k\%$ percentile of the dataset $\{x\}$
- $\mathsf{iqr}\{x\}$ is the interquartile range of the dataset $\{x\}$
- $\{\hat{x}\}$ is the dataset $\{x\}$, transformed to standard coordinates
- Standard normal data is defined in definition **??**,
- Normal data is defined in definition **??**,
- $\mathsf{corr}\,(\{(x, y)\})$ is the correlation between two components x and y of a dataset
- $\emptyset$ is the empty set.

- Ω is the set of all possible outcomes of an experiment.
- Sets are written as $\mathcal{A}$.
- $\mathcal{A}^c$ is the complement of the set $\mathcal{A}$ (i.e. $\Omega - \mathcal{A}$).
- $\mathcal{E}$ is an event (page 228).
- $P(\{\mathcal{E}\})$ is the probability of event $\mathcal{E}$ (page 228).
- $P(\{\mathcal{E}\}|\{\mathcal{F}\})$ is the probability of event $\mathcal{E}$, conditioned on event $\mathcal{F}$ (page 228).
- $p(x)$ is the probability that random variable X will take the value x; also written $P(\{X = x\})$ (page 228).
- $p(x, y)$ is the probability that random variable X will take the value x and random variable Y will take the value y; also written $P(\{X = x\} \cap \{Y = y\})$ (page 228).
- $\underset{x}{\text{argmax}}\ f(x)$ means the value of x that maximises $f(x)$.
- $\underset{x}{\text{argmin}}\ f(x)$ means the value of x that minimises $f(x)$.
- $\max_i(f(x_i))$ means the largest value that f takes on the different elements of the dataset $\{x_i\}$.
- $\hat{\theta}$ is an estimated value of a parameter θ.

1.1.1 Background Information

Cards: A standard deck of playing cards contains 52 cards. These cards are divided into four suits. The suits are: spades and clubs (which are black); and hearts and diamonds (which are red). Each suit contains 13 cards: Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack (sometimes called Knave), Queen and King. It is common to call Jack, Queen and King *court cards*.

Dice: If you look hard enough, you can obtain dice with many different numbers of sides (though I've never seen a three sided die). We adopt the convention that the sides of an N sided die are labeled with the numbers $1 \ldots N$, and that no number is used twice. Most dice are like this.

Fairness: Each face of a fair coin or die has the same probability of landing upmost in a flip or roll.

Roulette: A roulette wheel has a collection of slots. There are 36 slots numbered with the digits $1 \ldots 36$, and then one, two or even three slots numbered with zero. There are no other slots. A ball is thrown at the wheel when it is spinning, and it bounces around and eventually falls into a slot. If the wheel is properly balanced, the ball has the same probability of falling into each slot. The number of the slot the ball falls into is said to "come up". There are a variety of bets available.

1.2 SOME USEFUL MATHEMATICAL FACTS

The gamma function $\Gamma(x)$ is defined by a series of steps. First, we have that for n an integer,

$$\Gamma(n) = (n-1)!$$

and then for z a complex number with positive real part (which includes positive real numbers), we have

$$\Gamma(z) = \int_0^\infty t^z \frac{e^{-t}}{t} dt.$$

By doing this, we get a function on positive real numbers that is a smooth interpolate of the factorial function. We won't do any real work with this function, so won't expand on this definition. In practice, we'll either look up a value in tables or require a software environment to produce it.

1.3 ACKNOWLEDGEMENTS

Typos spotted by: Han Chen (numerous!), Henry Lin (numerous!), Paris Smaragdis (numerous!), Johnny Chang, Eric Huber, Brian Lunt, Yusuf Sobh, Scott Walters, — Your Name Here — TA's for this course have helped improve the notes. Thanks to Zicheng Liao, Michael Sittig, Nikita Spirin, Saurabh Singh, Daphne Tsatsoulis, Henry Lin, Karthik Ramaswamy.

1.4 THINGS YOU SHOULD KNOW BEFORE WE BEGIN

The list of terms, above, should contain no term that puzzles you (though you might not like the notation). Your linear algebra should be reasonably fluent at a practical level. Fairly soon, we will see: matrices; vectors; orthonormal matrices; eigenvalues; eigenvectors; and the singular value decomposition. All of these ideas will be used without too much comment.

You should be able to pick up a practical grasp of a programming environment without too much fuss. I use R for this sort of thing. You should too. In some places, we will use Python.

Remember this: most simple questions about programming can be answered by searching. When someone asks me one (say, how does one write a loop in R?) in office hours, I very often answer by searching for `R loop` (or whatever), and then pointing out that I wasn't required. The questioner is embarressed at this point. You could cut out the middleman in this transaction.

CHAPTER 2

Learning to Classify

A **classifier** is a procedure that accepts a set of features and produces a class label for them. Classifiers are immensely useful, and find wide application, because many problems are naturally classification problems. For example, if you wish to determine whether to place an advert on a web-page or not, you would use a classifier (i.e. look at the page, and say yes or no according to some rule). As another example, if you have a program that you found for free on the web, you would use a classifier to decide whether it was safe to run it (i.e. look at the program, and say yes or no according to some rule). As yet another example, credit card companies must decide whether a transaction is good or fraudulent.

All these examples are two class classifiers, but in many cases it is natural to have more classes. You can think of sorting laundry as applying a multi-class classifier. You can think of doctors as complex multi-class classifiers: a doctor accepts a set of features (your complaints, answers to questions, and so on) and then produces a response which we can describe as a class. The grading procedure for any class is a multi-class classifier: it accepts a set of features — performance in tests, homeworks, and so on — and produces a class label (the letter grade).

A classifier is usually trained by obtaining a set of labelled training examples and then searching for a classifier that optimizes some cost function which is evaluated on the training data. What makes training classifiers interesting is that performance on training data doesn't really matter. What matters is performance on run-time data, which may be extremely hard to evaluate because one often does not know the correct answer for that data. For example, we wish to classify credit-card transactions as safe or fraudulent. We could obtain a set of transactions with true labels, and train with those. But what we care about is new transactions, where it would be very difficult to know whether the classifier's answers are right. To be able to do anything at all, the set of labelled examples must be representative of future examples in some strong way. We will always assume that the labelled examples are an IID sample from the set of all possible examples, though we never use the assumption explicitly.

Definition: 2.1 *Classifier*

A classifier is a procedure that accepts a set of features and produces a label. Classifiers are trained on labelled examples, but the goal is to get a classifier that performs well on data which is not seen at the time of training. Training a classifier requires labelled data that is representative of future data.

2.1 CLASSIFICATION: THE BIG IDEAS

We will write the training dataset $(\mathbf{x}_i, y_i)$. For the i'th example, $\mathbf{x}_i$ represents the values taken by a collection of features. In the simplest case, $\mathbf{x}_i$ would be a vector of real numbers. In some cases, $\mathbf{x}_i$ could contain categorical data or even unknown values. Although $\mathbf{x}_i$ isn't guaranteed to be a vector, it's usually referred to as a **feature vector**. The y_i are labels giving the type of the object that generated the example. We must use these labelled examples to come up with a classifier.

2.1.1 The Error Rate, and Other Summaries of Performance

We can summarize the performance of any particular classifier using the **error** or **total error rate** (the percentage of classification attempts that gave the wrong answer) and the **accuracy** (the percentage of classification attempts that give the right answer). For most practical cases, even the best choice of classifier will make mistakes. For example, an alien tries to classify humans into male and female, using only height as a feature. Whatever the alien's classifier does with that feature, it will make mistakes. This is because the classifier must choose, for each value of height, whether to label the humans with that height male or female. But for the vast majority of heights, there are some males and some females with that height, and so the alien's classifier must make some mistakes.

As the example suggests, a particular feature vector $\mathbf{x}$ may appear with different labels (so the alien will see six foot males and six foot females, quite possibly in the training dataset and certainly in future data). Labels appear with some probability conditioned on the observations, $P(y|\mathbf{x})$. If there are parts of the feature space where $P(\mathbf{x})$ is relatively large (so we expect to see observations of that form) *and* where $P(y|\mathbf{x})$ has relatively large values for more than one label, even the best possible classifier will have a high error rate. If we knew $P(y|\mathbf{x})$ (which is seldom the case), we could identify the classifier with the smallest error rate and compute its error rate. The minimum expected error rate obtained with the best possible classifier applied to a particular problem is known as the **Bayes risk** for that problem. In most cases, it is hard to know what the Bayes risk is, because to compute it requires knowing $P(y|\mathbf{x})$, which isn't usually known.

The error rate of a classifier is not that meaningful on its own, because we don't usually know the Bayes risk for a problem. It is more helpful to compare a particular classifier with some natural alternatives, sometimes called **baselines**. The choice of baseline for a particular problem is almost always a matter of application logic. The simplest general baseline is a know-nothing strategy. Imagine classifying the data without using the feature vector at all — how well does this strategy do? If each of the C classes occurs with the same frequency, then it's enough to label the data by choosing a label uniformly and at random, and the error rate for this strategy is $1 - 1/C$. If one class is more common than the others, the lowest error rate is obtained by labelling everything with that class. This comparison is often known as **comparing to chance**.

It is very common to deal with data where there are only two labels. You should keep in mind this means the highest possible error rate is 50% — if you have

a classifier with a higher error rate, you can improve it by switching the outputs. If one class is much more common than the other, training becomes more complicated because the best strategy – labelling everything with the common class – becomes hard to beat.

2.1.2 More Detailed Evaluation

The error rate is a fairly crude summary of the classifier's behavior. For a two-class classifier and a 0-1 loss function, one can report the **false positive rate** (the percentage of negative test data that was classified positive) and the **false negative rate** (the percentage of positive test data that was classified negative). Note that it is important to provide both, because a classifier with a low false positive rate tends to have a high false negative rate, and vice versa. As a result, you should be suspicious of reports that give one number but not the other. Alternative numbers that are reported sometimes include the **sensitivity** (the percentage of true positives that are classified positive) and the **specificity** (the percentage of true negatives that are classified negative).

		Predict					
		0	1	2	3	4	Class error
True	0	151	7	2	3	1	7.9%
	1	32	5	9	9	0	91%
	2	10	9	7	9	1	81%
	3	6	13	9	5	2	86%
	4	2	3	2	6	0	100%

TABLE 2.1: *The class confusion matrix for a multiclass classifier. This is a table of cells, where the i, j'th cell contains the count of cases where the true label was i and the predicted label was j (some people show the fraction of cases rather than the count). Further details about the dataset and this example appear in worked example 3.1.*

The false positive and false negative rates of a two-class classifier can be generalized to evaluate a multi-class classifier, yielding the **class confusion matrix**. This is a table of cells, where the i, j'th cell contains the count of cases where the true label was i and the predicted label was j (some people show the fraction of cases rather than the count). Table 2.1 gives an example. This is a class confusion matrix from a classifier built on a dataset where one tries to predict the degree of heart disease from a collection of physiological and physical measurements. There are five classes $(0 \dots 4)$. The i, j'th cell of the table shows the number of data points of true class i that were classified to have class j. As I find it hard to recall whether rows or columns represent true or predicted classes, I have marked this on the table. For each row, there is a **class error rate**, which is the percentage of data points of that class that were misclassified. The first thing to look at in a table like this is the diagonal; if the largest values appear there, then the classifier is working well. This clearly isn't what is happening for table 2.1. Instead, you can see that the method is very good at telling whether a data point is in class 0 or

not (the class error rate is rather small), but cannot distinguish between the other classes. This is a strong hint that the data can't be used to draw the distinctions that we want. It might be a lot better to work with a different set of classes.

2.1.3 Overfitting and Cross-Validation

Choosing and evaluating a classifier takes some care. The goal is to get a classifier that works well on future data *for which we might never know the true label*, using a training set of labelled examples. This isn't necessarily easy. For example, think about the (silly) classifier that takes any data point and, if it is the same as a point in the training set, emits the class of that point; otherwise, it chooses randomly between the classes.

The **training error** of a classifier is the error rate on examples used to train the classifier. In contrast, the **test error** is error on examples not used to train the classifier. Classifiers that have small training error might not have small test error, because the classification procedure is chosen to do well on the training data. This effect is sometimes called **overfitting**. Other names include **selection bias**, because the training data has been selected and so isn't exactly like the test data, and **generalizing badly**, because the classifier must generalize from the training data to the test data. The effect occurs because the classifier has been chosen to perform well *on the training dataset*. An efficient training procedure is quite likely to find special properties of the training dataset that aren't representative of the test dataset, because the training dataset is not the same as the test dataset. The training dataset is typically a sample of all the data one might like to have classified, and so is quite likely a lot smaller than the test dataset. Because it is a sample, it may have quirks that don't appear in the test dataset. One consequence of overfitting is that classifiers should always be evaluated on data that was not used in training.

Now assume that we want to estimate the error rate of the classifier on test data. We cannot estimate the error rate of the classifier using data that was used to train the classifier, because the classifier has been trained to do well on that data, which will mean our error rate estimate will be too low. An alternative is to separate out some training data to form a **validation set** (confusingly, this is sometimes called a test set), then train the classifier on the rest of the data, and evaluate on the validation set. The error estimate on the validation set is the value of a random variable, because the validation set is a sample of all possible data you might classify. But this error estimate is **unbiased**, meaning that the expected value of the error estimate is the true value of the error. You can see this by thinking about the error estimate as a sample mean and applying the ideas of Chapter **??**.

However, separating out some training data presents the difficulty that the classifier will not be the best possible, because we left out some training data when we trained it. This issue can become a significant nuisance when we are trying to tell which of a set of classifiers to use — did the classifier perform poorly on validation data because it is not suited to the problem representation or because it was trained on too little data?

We can resolve this problem with **cross-validation**, which involves repeat-

edly: splitting data into training and validation sets uniformly and at random, training a classifier on the training set, evaluating it on the validation set, and then averaging the error over all splits. Each different split is usually called a **fold**. This procedure yields an estimate of the likely future performance of a classifier, at the expense of substantial computation. A common form of this algorithm uses a single data item to form a validation set. This is known as **leave-one-out cross-validation**.

Remember this: *Classifiers usually perform better on training data than on test data, because the classifier was chosen to do well on the training data. This effect is known as overfitting. To get an accurate estimate of future performance, classifiers should always be evaluated on data that was not used in training.*

2.2 CLASSIFYING WITH NEAREST NEIGHBORS

Assume we have a labelled dataset consisting of N pairs $(\mathbf{x}_i, y_i)$. Here $\mathbf{x}_i$ is the i'th feature vector, and y_i is the i'th class label. We wish to predict the label y for any new example $\mathbf{x}$; this is often known as a query example or query. Here is a really effective strategy: Find the labelled example $\mathbf{x}_c$ that is closest to $\mathbf{x}$, and report the class of that example.

How well can we expect this strategy to work? A precise analysis would take us way out of our way, but simple reasoning is informative. Assume there are two classes, 1 and -1 (the reasoning will work for more, but the description is slightly more involved). We expect that, if $\mathbf{u}$ and $\mathbf{v}$ are sufficiently close, then $p(y|\mathbf{u})$ is similar to $p(y|\mathbf{v})$. This means that if a labelled example $\mathbf{x}_i$ is close to $\mathbf{x}$, then $p(y|\mathbf{x})$ is similar to $p(y|\mathbf{x}_i)$. Furthermore, we expect that queries are "like" the labelled dataset, in the sense that points that are common (resp. rare) in the labelled data will appear often (resp. seldom) in the queries.

Now imagine the query comes from a location where $p(y = 1|\mathbf{x})$ is large. The closest labelled example $\mathbf{x}_c$ should be nearby (because queries are "like" the labelled data) and should be labelled with 1 (because nearby examples have similar label probabilities). So the method should produce the right answer with high probability.

Alternatively, imagine the query comes from a location where $p(y = 1|\mathbf{x})$ is about the same as $p(y = -1|\mathbf{x})$. The closest labelled example $\mathbf{x}_c$ should be nearby (because queries are "like" the labelled data). But think about a set of examples that are about as close. The labels in this set should vary significantly (because $p(y = 1|\mathbf{x})$ is about the same as $p(y = -1|\mathbf{x})$. This means that, if the query is labelled 1 (resp. -1), a small change in the query will cause it to be labelled -1 (resp. 1). In these regions the classifier will tend to make mistakes more often, as it should. Using a great deal more of this kind of reasoning, nearest neighbors can be shown to produce an error that is no worse than twice the best error rate, if the

method has enough examples. There is no prospect of seeing enough examples in practice for this result to apply.

One important generalization is to find the k nearest neighbors, then choose a label from those. A (k, l) nearest neighbor classifier finds the k example points closest to the point being considered, and classifies this point with the class that has the highest number of votes, as long as this class has more than l votes (otherwise, the point is classified as unknown). In practice, one seldom uses more than three nearest neighbors.

2.2.1 Practical Considerations for Nearest Neighbors

One practical difficulty in using nearest neighbor classifiers is you need a lot of labelled examples for the method to work. For some problems, this means you can't use the method. A second practical difficulty is you need to use a sensible choice of distance. For features that are obviously of the same type, such as lengths, the usual metric may be good enough. But what if one feature is a length, one is a color, and one is an angle? It is almost always a good idea to scale each feature independently so that the variance of each feature is the same, or at least consistent; this prevents features with very large scales dominating those with very small scales. Another possibility is to transform the features so that the covariance matrix is the identity (this is sometimes known as **whitening**; the method follows from the ideas of Chapter 4). This can be hard to do if the dimension is so large that the covariance matrix is hard to estimate.

A third practical difficulty is you need to be able to find the nearest neighbors for your query point. This is surprisingly difficult to do faster than simply checking the distance to each training example separately. If your intuition tells you to use a tree and the difficulty will go away, your intuition isn't right. It turns out that nearest neighbors in high dimensions is one of those problems that is a lot harder than it seems, because high dimensional spaces are quite hard to reason about informally. There's a long history of methods that appear to be efficient but, once carefully investigated, turn out to be bad.

Fortunately, it is usually enough to use an **approximate nearest neighbor**. This is an example that is, with high probability, almost as close to the query point as the nearest neighbor is. Obtaining an approximate nearest neighbor is very much easier than obtaining a nearest neighbor. We can't go into the details here, but there are several distinct methods for finding approximate nearest neighbors. Each involves a series of tuning constants and so on, and, on different datasets, different methods and different choices of tuning constant produce the best results. If you want to use a nearest neighbor classifier on a lot of run-time data, it is usually worth a careful search over methods and tuning constants to find an algorithm that yields a very fast response to a query. It is known how to do this search, and there is excellent software available (FLANN, http://www.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN, by Marius Muja and David G. Lowe).

It is straightforward to use cross-validation to estimate the error rate of a nearest neighbor classifier. Split the labelled training data into two pieces, a (typically large) training set and a (typically small) validation set. Now take each element of the validation set and label it with the label of the closest element of the training

set. Compute the fraction of these attempts that produce an error (the true label and predicted labels differ). Now repeat this for a different split, and average the errors over splits. With care, the code you'll write is shorter than this description.

Worked example 2.1 *Classifying using nearest neighbors*

Build a nearest neighbor classifier to classify the MNIST digit data. This dataset is very widely used to check simple methods. It was originally constructed by Yann Lecun, Corinna Cortes, and Christopher J.C. Burges. It has been extensively studied. You can find this dataset in several places. The original dataset is at http://yann.lecun.com/exdb/mnist/. The version I used was used for a Kaggle competition (so I didn't have to decompress Lecun's original format). I found it at http://www.kaggle.com/c/digit-recognizer.

Solution: I used R for this problem. As you'd expect, R has nearest neighbor code that seems quite good (I haven't had any real problems with it, at least). There isn't really all that much to say about the code. I used the R FNN package. I trained on 1000 of the 42000 examples in the Kaggle version, and I tested on the next 200 examples. For this (rather small) case, I found the following class confusion matrix:

		Predict									
		0	1	2	3	4	5	6	7	8	9
True	0	12	0	0	0	0	0	0	0	0	0
	1	0	20	4	1	0	1	0	2	2	1
	2	0	0	20	1	0	0	0	0	0	0
	3	0	0	0	12	0	0	0	0	4	0
	4	0	0	0	0	18	0	0	0	1	1
	5	0	0	0	0	0	19	0	0	1	0
	6	1	0	0	0	0	0	18	0	0	0
	7	0	0	1	0	0	0	0	19	0	2
	8	0	0	1	0	0	0	0	0	16	0
	9	0	0	0	2	3	1	0	1	1	14

There are no class error rates here, because I couldn't recall the magic line of R to get them. However, you can see the classifier works rather well for this case. MNIST is comprehensively explored in the exercises.

Remember this: *Nearest neighbors has good properties. With enough training data and a low enough dimension, the error rate is guaranteed to be no more than twice the best error rate. The method is wonderfully flexible about the labels the classifier predicts. Nothing changes when you go from a two-class classifier to a multi-class classifier.*
There are important difficulties. You need a large training dataset. If you don't have a reliable measure of how far apart two things are, you shouldn't be doing nearest neighbors. And you need to be able to query a large dataset of examples to find the nearest neighbor of a point.

2.3 NAIVE BAYES

One straightforward source of a classifier is a probability model. For the moment, assume we know $p(y|\mathbf{x})$ for our data. Assume also that all errors in classification are equally important. Then the following rule produces smallest possible expected classification error rate:

> For a test example $\mathbf{x}$, report the class y that has the highest value of $(p(y|\mathbf{x}))$. If the largest value is achieved by more than one class, choose randomly from that set of classes.

Usually, we do not have $p(y|\mathbf{x})$. If we have $p(\mathbf{x}|y)$ (often called either a **likelihood** or **class conditional probability**, compare Section **??**), and $p(y)$ (often called a **prior**, compare Section **??**) then we can use Bayes' rule to form

$$p(y|\mathbf{x}) = \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})}$$

(the **posterior**, compare Section **??**). This isn't much help in this form, but write $x^{(j)}$ for the j'th component of $\mathbf{x}$. Now *assume* that features are conditionally independent conditioned on the class of the data item. Our assumption is

$$p(\mathbf{x}|y) = \prod_j p(x^{(j)}|y).$$

It is very seldom the case that this assumption is true, but it turns out to be fruitful to pretend that it is. This assumption means that

$$\begin{aligned} p(y|\mathbf{x}) &= \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})} \\ &= \frac{\left(\prod_j p(x^{(j)}|y)\right) p(y)}{p(\mathbf{x})} \\ &\propto \left(\prod_j p(x^{(j)}|y)\right) p(y). \end{aligned}$$

Now to make a decision, we need to choose the class that has the largest value of $p(y|\mathbf{x})$. In turn, this means we need only know the posterior values up to scale at $\mathbf{x}$, so we don't need to estimate $p(\mathbf{x})$. In the case of where all errors have the same cost, this yields the rule

$$\text{choose } y \text{ such that } \left[\left(\prod_j p(x^{(j)}|y)\right) p(y)\right] \text{ is largest.}$$

This rule suffers from a practical problem. You can't actually multiply a large number of probabilities and expect to get an answer that a floating point system thinks is different from zero. Instead, you should add the log probabilities. Notice that the logarithm function has one nice property: it is monotonic, meaning that $a > b$ is equivalent to $\log a > \log b$. This means the following, more practical, rule is equivalent:

$$\text{choose } y \text{ such that } \left[\left(\sum_j \log p(x^{(j)}|y)\right) + \log p(y)\right] \text{ is largest.}$$

To use this rule, we need models for $p(y)$ and for $p(x^{(j)}|y)$ for each j. The usual way to find a model of $p(y)$ is to count the number of training examples in each class, then divide by the number of classes.

It turns out that simple parametric models work really well for $p(x^{(j)}|y)$. For example, one could use a normal distribution for each $x^{(j)}$ in turn, for each possible value of y, using the training data. The parameters of this normal distribution are chosen using maximum likelihood. The logic of the measurements might suggest other distributions, too. If one of the $x^{(j)}$'s was a count, we might fit a Poisson distribution (again, using maximum likelihood). If it was a 0-1 variable, we might fit a Bernoulli distribution. If it was a discrete variable, then we might use a multinomial model. Even if the $x^{(j)}$ is continuous, we can use a multinomial model by quantizing to some fixed set of values; this can be quite effective.

A naive bayes classifier that has poorly fitting models for each feature could classify data very well. This (reliably confusing property) occurs because classification doesn't require a good model of $p(\mathbf{x}|y)$, or even of $p(y|\mathbf{x})$. All that needs to happen is that, at any $\mathbf{x}$, the score for the right class is higher than the score for all other classes. Figure 2.1 shows an example where a normal model of the class-conditional histograms is poor, but the normal model will result in a good naive bayes classifier. This works because a data item from (say) class one will reliably have a larger probability under the normal model for class one than it will for class two.

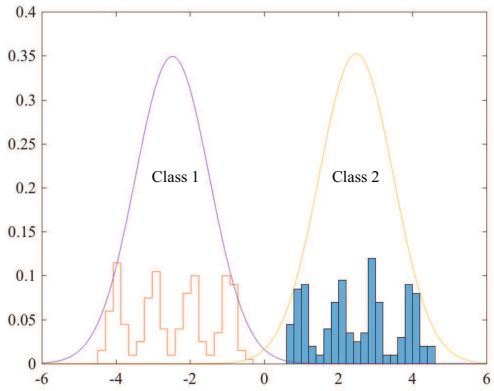


FIGURE 2.1: *The figure shows class conditional histograms of a feature x for two different classes. The histograms have been normalized so that the counts sum to one, so you can think of them as probability distributions. It should be fairly obvious that a normal model (superimposed) doesn't describe these histograms well. However, the normal model will result in a good naive bayes classifier.*

Worked example 2.2 *Classifying breast tissue samples*

The "breast tissue" dataset at https://archive.ics.uci.edu/ml/datasets/Breast+Tissue contains measurements of a variety of properties of six different classes of breast tissue. Build and evaluate a naive bayes classifier to distinguish between the classes automatically from the measurements.

Solution: I used R for this example, because I could then use packages easily. The main difficulty here is finding appropriate packages, understanding their documentation, and checking they're right (unless you want to write the source yourself, which really isn't all that hard). I used the R package `caret` to do train-test splits, cross-validation, etc. on the naive bayes classifier in the R package `klaR`. I separated out a test set randomly (approx 20% of the cases for each class, chosen at random), then trained with cross-validation on the remainder. I used a normal model for each feature. The class-confusion matrix on the test set was:

		Predict					
		adi	car	con	fad	gla	mas
True	adi	2	0	0	0	0	0
	car	0	3	0	0	0	1
	con	2	0	2	0	0	0
	fad	0	0	0	0	1	0
	gla	0	0	0	0	2	1
	mas	0	1	0	3	0	1

which is fairly good. The accuracy is 52%. In the training data, the classes are nearly balanced and there are six classes, meaning that chance is about 17%. These numbers, and the class-confusion matrix, will vary with test-train split. I have not averaged over splits, which would give a somewhat more accurate estimate of accuracy.

2.3.1 Cross-Validation to Choose a Model

Naive bayes presents us with a new problem. We can choose from several different types of model for $p(x^{(j)}|y)$ (eg normal models vs. Poisson models), and we need to know which one produces the best classifier. We also need to know how well that classifier will work. It is natural to use cross-validation to estimate how well each type of model works. You can't just look at every type of model for every variable, because that would yield too many models. Instead, choose M types of model that seem plausible (for example, by looking at histograms of feature components conditioned on class and using your judgement). Now compute a cross-validated error for each of M types of model, and choose the type of model with lowest cross-validated error. Computing the cross-validated error involves repeatedly splitting the training set into two pieces, fitting the model on one and computing the error on the other, then averaging the errors. Notice this means the model you fit to each fold will have slightly different parameter values, because each fold has slightly different has slightly different training data.

However, once we have chosen the type of model, we have two problems. First, we do not know the correct values for the parameters of the best type of model. For each fold in the cross-validation, we estimated slightly different parameters because we trained on slightly different data, and we don't know which estimate is right. Second, we do not have a good estimate of how well the best model works. This is because we chose the type of model with the smallest error estimate, which is likely smaller than the true error estimate for that type of model.

This problem is easily dealt with if you have a reasonably sized dataset. Split the labelled dataset into two pieces. One (call it the training set) is used for training and for choosing a model type, the other (call it the test set) is used only for evaluating the final model. Now for each type of model, compute the cross-validated error on the training set.

Now use the cross-validated error to choose the type of model. Very often this just means you choose the type that produces the lowest cross-validated error, but there might be cases where two types produce about the same error and one is a lot faster to evaluate, etc. Take the entire training set, and use this to estimate the parameters for that type of model. This estimate should be (a little) better than any of the estimates produced in the cross-validation, because it uses (slightly) more data. Finally, evaluate the resulting model on the test set.

This procedure is rather harder to describe than to do (there's a pretty natural set of nested loops here). There are some strong advantages. First, the estimate of how well a particular model type works is unbiased, because we evaluated on data not used on training. Second, once you have chosen a type of model, the parameter estimate you make is the best you can because you used all the training set to obtain it. Finally, your estimate of how well that particular model works is unbiased, too, because you obtained it using data that wasn't used to train or to select a model.

2.3.2 Missing Data

Missing data occurs when some values in the training data are unknown. This can happen in a variety of ways. Someone didn't record the value; someone recorded

it incorrectly, and you know the value is wrong but you don't know what the right one is; the dataset was damaged in storage or transmission; instruments failed; and so on. This is quite typical of data where the feature values are obtained by measuring effects in the real world. It's much less common where the feature values are computed from signals – for example, when one tries to classify digital images, or sound recordings.

Missing data can be a serious nuisance in classification problems, because many methods cannot handle incomplete feature vectors. For example, nearest neighbors has no real way of proceeding if some components of the feature vector are unknown. If there are relatively few incomplete feature vectors, one could just drop them from the dataset and proceed, but this should strike you as inefficient.

Naive bayes is rather good at handling data where there are many incomplete feature vectors in quite a simple way. For example, assume for some i, we wish to fit $p(x_i|y)$ with a normal distribution. We need to estimate the mean and standard deviation of that normal distribution (which we do with maximum likelihood, as one should). If not every example has a known value of x_i, this really doesn't matter; we simply omit the unknown number from the estimate. Write $x_{i,j}$ for the value of x_i for the j'th example. To estimate the mean, we form

$$\frac{\sum_{j \in \text{cases with known values}} x_{i,j}}{\text{number of cases with known values}}$$

and so on.

Dealing with missing data during classification is easy, too. We need to look for the y that produces the largest value of $\sum_i \log p(x_i|y)$. We can't evaluate $p(x_i|y)$ if the value of that feature is missing - but it is missing for each class. We can just leave that term out of the sum, and proceed. This procedure is fine if data is missing as a result of "noise" (meaning that the missing terms are independent of class). If the missing terms depend on the class, there is much more we could do — for example, we might build a model of the class-conditional density of missing terms.

Notice that if some values of a discrete feature x_i don't appear for some class, you could end up with a model of $p(x_i|y)$ that had zeros for some values. This almost inevitably leads to serious trouble, because it means your model states you cannot ever observe that value for a data item of that class. This isn't a safe property: it is hardly ever the case that not observing something means you cannot observe it. A simple, but useful, fix is to add one to all small counts. More sophisticated methods are available, but well beyond our scope.

Remember this: *Naive bayes classifiers are straightforward to build, and very effective. Dealing with missing data is easy. Experience has shown they are particularly effective at high dimensional data. A straightforward variant of cross-validation helps select which particular model to use.*

2.4 YOU SHOULD

2.4.1 remember these definitions:

2.4.2 remember these terms:

2.4.3 remember these facts:

2.4.4 use these procedures:

2.4.5 be able to:

- build a nearest neighbors classifier using your preferred software package, and produce a cross-validated estimate of its error rate or its accuracy;
- build a naive bayes classifier using your preferred software package, and produce a cross-validated estimate of its error rate or its accuracy;

PROGRAMMING EXERCISES

2.1. The UC Irvine machine learning data repository hosts a famous collection of data on whether a patient has diabetes (the Pima Indians dataset), originally owned by the National Institute of Diabetes and Digestive and Kidney Diseases and donated by Vincent Sigillito. This can be found at http://archive.ics.uci.edu/ml/datasets/Pima+Indians+Diabetes. This data has a set of attributes of patients, and a categorical variable telling whether the patient is diabetic or not. This is an exercise oriented to users of R, because you can use some packages to help.

- **(a)** Build a simple naive Bayes classifier to classify this data set. You should hold out 20% of the data for evaluation, and use the other 80% for training. You should use a normal distribution to model each of the class-conditional distributions. You should write this classifier yourself.
- **(b)** Now use the `caret` and `klaR` packages to build a naive bayes classifier for this data. The `caret` package does cross-validation (look at `train`) and can be used to hold out data. The `klaR` package can estimate class-conditional densities using a density estimation procedure that I will describe much later in the course. Use the cross-validation mechanisms in `caret` to estimate the accuracy of your classifier.
- **(c)** Now install SVMLight, which you can find at http://svmlight.joachims.org, via the interface in `klaR` (look for `svmlight` in the manual) to train and evaluate an SVM to classify this data. You don't need to understand much about SVM's to do this — we'll do that in following exercises. You should hold out 20% of the data for evaluation, and use the other 80% for training.

2.2. The UC Irvine machine learning data repository hosts a collection of data on student performance in Portugal, donated by Paulo Cortez, University of Minho, in Portugal. You can find this data at https://archive.ics.uci.edu/ml/datasets/Student+Performance. It is described in P. Cortez and A. Silva. "Using Data Mining to Predict Secondary School Student Performance," In A. Brito and J. Teixeira Eds., *Proceedings of 5th FUture BUsiness TEChnology Conference (FUBUTEC 2008)* pp. 5-12, Porto, Portugal, April, 2008,
There are two datasets (for grades in mathematics and for grades in Portugese). There are 30 attributes each for 649 students, and 3 values that can be predicted (G1, G2 and G3). Of these, ignore G1 and G2.

- **(a)** Use the mathematics dataset. Take the G3 attribute, and quantize this into two classes, $G3 > 12$ and $G3 \leq 12$. Build and evaluate a naive bayes classifier that predicts G3 from all attributes except G1 and G2. You should build this classifier from scratch (i.e. DON'T use the packages described in the code snippets). For binary attributes, you should use a binomial model. For the attributes described as "numeric", which take a small set of values, you should use a multinomial model. For the attributes described as "nominal", which take a small set of values, you should again use a multinomial model. Ignore the "absence" attribute. Estimate accuracy by cross-validation. You should use at least 10 folds, excluding 15% of the data at random to serve as test data, and average the accuracy over those folds. Report the mean and standard deviation of the accuracy over the folds.
- **(b)** Now revise your classifier of the previous part so that, for the attributes described as "numeric", which take a small set of values, you use a multinomial model. For the attributes described as "nominal", which take a

small set of values, you should still use a multinomial model. Ignore the "absence" attribute. Estimate accuracy by cross-validation. You should use at least 10 folds, excluding 15% of the data at random to serve as test data, and average the accuracy over those folds. Report the mean and standard deviation of the accuracy over the folds.

(c) Which classifier do you believe is more accurate and why?

2.3. The UC Irvine machine learning data repository hosts a collection of data on heart disease. The data was collected and supplied by Andras Janosi, M.D., of the Hungarian Institute of Cardiology, Budapest; William Steinbrunn, M.D., of the University Hospital, Zurich, Switzerland; Matthias Pfisterer, M.D., of the University Hospital, Basel, Switzerland; and Robert Detrano, M.D., Ph.D., of the V.A. Medical Center, Long Beach and Cleveland Clinic Foundation. You can find this data at https://archive.ics.uci.edu/ml/datasets/Heart+Disease.

Use the processed Cleveland dataset, where there are a total of 303 instances with 14 attributes each. The irrelevant attributes described in the text have been removed in these. The 14'th attribute is the disease diagnosis. There are records with missing attributes, and you should drop these.

(a) Take the disease attribute, and quantize this into two classes, num = 0 and num > 0. Build and evaluate a naive bayes classifier that predicts the class from all other attributes Estimate accuracy by cross-validation. You should use at least 10 folds, excluding 15% of the data at random to serve as test data, and average the accuracy over those folds. Report the mean and standard deviation of the accuracy over the folds.

(b) Now revise your classifier to predict each of the possible values of the disease attribute (0-4 as I recall). Estimate accuracy by cross-validation. You should use at least 10 folds, excluding 15% of the data at random to serve as test data, and average the accuracy over those folds. Report the mean and standard deviation of the accuracy over the folds.

2.4. The UC Irvine machine learning data repository hosts a collection of data on breast cancer diagnostics, donated by Olvi Mangasarian, Nick Street, and William H. Wolberg. You can find this data at http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic). For each record, there is an id number, 10 continuous variables, and a class (benign or malignant). There are 569 examples. Separate this dataset randomly into 100 validation, 100 test, and 369 training examples.

Write a program to train a support vector machine on this data using stochastic gradient descent. You should not use a package to train the classifier (you don't really need one), but your own code. You should ignore the id number, and use the continuous variables as a feature vector. You should scale these variables so each has unit variance. You should search for an appropriate value of the regularization constant, trying at least the values $\lambda = [1e-3, 1e-2, 1e-1, 1]$. Use the validation set for this search.

You should use at least 50 epochs of at least 100 steps each. In each epoch, you should separate out 50 training examples at random for evaluation. You should compute the accuracy of the current classifier on the set held out for the epoch every 10 steps. You should produce:

(a) A plot of the accuracy every 10 steps, for each value of the regularization constant.

(b) Your estimate of the best value of the regularization constant, together with a brief description of why you believe that is a good value.

(c) Your estimate of the accuracy of the best classifier on held out data

CHAPTER 3

SVM's and Random Forests

3.1 THE SUPPORT VECTOR MACHINE

Assume we have a labelled dataset consisting of N pairs $(\mathbf{x}_i, y_i)$. Here $\mathbf{x}_i$ is the i'th feature vector, and y_i is the i'th class label. We will assume that there are two classes, and that y_i is either 1 or -1. We wish to predict the sign of y for any point $\mathbf{x}$. We will use a linear classifier, so that for a new data item $\mathbf{x}$, we will predict

$$\mathsf{sign}\left(\mathbf{a}^T\mathbf{x} + b\right)$$

and the particular classifier we use is given by our choice of $\mathbf{a}$ and b.

You should think of $\mathbf{a}$ and b as representing a hyperplane, given by the points where $\mathbf{a}^T\mathbf{x} + b = 0$. Notice that the magnitude of $\mathbf{a}^T\mathbf{x} + b$ grows as the point $\mathbf{x}$ moves further away from the hyperplane. This hyperplane separates the positive data from the negative data, and is an example of a **decision boundary**. When a point crosses the decision boundary, the label predicted for that point changes. All classifiers have decision boundaries. Searching for the decision boundary that yields the best behavior is a fruitful strategy for building classifiers.

Example: 3.1 *A linear model with a single feature*

Assume we use a linear model with one feature. For an example with feature value x, predicts $\mathsf{sign}\,(ax + b)$. Equivalently, the model tests x against the threshold $-b/a$.

Example: 3.2 *A linear model with two features*

Assume we use a linear model with two features. For an example with feature vector $\mathbf{x}$, the model predicts $\mathsf{sign}\left(\mathbf{a}^T\mathbf{x} + b\right)$. The sign changes along the line $\mathbf{a}^T\mathbf{x} + b = 0$. You should check that this is, indeed, a line. On one side of this line, the model makes positive predictions; on the other, negative. Which side is which can be swapped by multiplying $\mathbf{a}$ and b by -1.

This family of classifiers may look bad to you, and it is easy to come up with examples that it misclassifies badly. In fact, the family is extremely strong. First, it is easy to estimate the best choice of rule for very large datasets. Second, linear

classifiers have a long history of working very well in practice on real data. Third, linear classifiers are fast to evaluate.

In practice, examples that are classified badly by the linear rule usually are classified badly because there are too few features. Remember the case of the alien who classified humans into male and female by looking at their heights; if that alien had looked at their chromosomes as well as height, the error rate would have been smaller. In practical examples, experience shows that the error rate of a poorly performing linear classifier can usually be improved by adding features to the vector $\mathbf{x}$.

We will choose $\mathbf{a}$ and b by choosing values that minimize a cost function. The cost function must achieve two goals. First, the cost function needs a term that ensures each training example should be on the right side of the decision boundary (or, at least, not be too far on the wrong side). Second, the cost function needs a term that should penalize errors on query examples. The appropriate cost function has the form:

$$\text{Training error cost} + \lambda \text{ penalty term}$$

where λ is an unknown weight that balances these two goals. We will eventually set the value of λ by a search process.

3.1.1 The Hinge Loss

Write

$$\gamma_i = \mathbf{a}^T\mathbf{x}_i + b$$

for the value that the linear function takes on example i. Write $C(\gamma_i, y_i)$ for a function that compares γ_i with y_i. The training error cost will be of the form

$$(1/N)\sum_{i=1}^{N} C(\gamma_i, y_i).$$

A good choice of C should have some important properties.

- If γ_i and y_i have different signs, then C should be large, because the classifier will make the wrong prediction for this training example. Furthermore, if γ_i and y_i have different signs *and* γ_i has large magnitude, then the classifier will very likely make the wrong prediction for test examples that are close to $\mathbf{x}_i$. This is because the magnitude of $(\mathbf{a}^T\mathbf{x} + b)$ grows as $\mathbf{x}$ gets further from the decision boundary. So C should get larger as the magnitude of γ_i gets larger in this case.

- If γ_i and y_i have the same signs, but γ_i has small magnitude, then the classifier will classify $\mathbf{x}_i$ correctly, but might not classify points that are nearby correctly. This is because a small magnitude of γ_i means that $\mathbf{x}_i$ is close to the decision boundary, so there will be points nearby that are on the other side of the decision boundary. We want to discourage this, so C should not be zero in this case.

- Finally, if γ_i and y_i have the same signs and γ_i has large magnitude, then C can be zero because $\mathbf{x}_i$ is on the right side of the decision boundary and so are all the points near to $\mathbf{x}_i$.

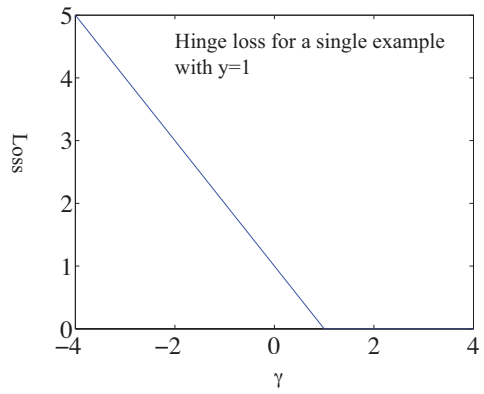


FIGURE 3.1: *The hinge loss, plotted for the case $y_i = 1$. The horizontal variable is the $\gamma_i = \mathbf{a}^T\mathbf{x}_i + b$ of the text. Notice that giving a strong negative response to this positive example causes a loss that grows linearly as the magnitude of the response grows. Notice also that giving an insufficiently positive response also causes a loss. Giving a strongly positive response is free.*

The **hinge loss**, which takes the form

$$C(y_i, \gamma_i) = \max(0, 1 - y_i\gamma_i),$$

has these properties.

- If γ_i and y_i have different signs, then C will be large. Furthermore, the cost grows linearly as $\mathbf{x}_i$ moves further away from the boundary on the wrong side.
- If γ_i and y_i have the same sign, but $y_i\gamma_i < 1$ (which means that $\mathbf{x}_i$ is close to the decision boundary), there is some cost, which gets larger as $\mathbf{x}_i$ gets closer to the boundary.
- If $y_i\gamma_i > 1$ (so the classifier predicts the sign correctly *and* $\mathbf{x}_i$ is far from the boundary) there is no cost.

A classifier trained to minimize this loss is encouraged to (a) make strong positive (or negative) predictions for positive (or negative) examples and (b) for examples it gets wrong, make predictions with the smallest magnitude that it can. A linear classifier trained with the hinge loss is known as a **support vector machine** or **SVM**.

3.1.2 Regularization

The penalty term is needed, because the hinge loss has one odd property. Assume that the pair $\mathbf{a}$, b correctly classifies all training examples, so that $y_i(\mathbf{a}^T x_i + b) > 0$. Then we can always ensure that the hinge loss for the dataset is zero, by scaling $\mathbf{a}$ and b, because you can choose a scale so that $y_j(\mathbf{a}^T x_j + b) > 1$ for *every* example index j. This scale hasn't changed the result of the classification rule on the training

data. Now if $\mathbf{a}$ and b result in a hinge loss of zero, then so do $2\mathbf{a}$ and $2b$. This should worry you, because it means we can't choose the classifier parameters uniquely.

Now think about future examples. We don't know what their feature values will be, and we don't know their labels. But we do know that the hinge loss for an example with feature vector $\mathbf{x}$ and unknown label y will be $\max(0, 1 - y\left[\mathbf{a}^T\mathbf{x} + b\right])$. Now imagine the hinge loss for this example *isn't* zero. If the example is classified correctly, then it is close to the decision boundary. We expect that there are fewer of these examples than examples that are far from the decision boundary and on the wrong side, so we concentrate on examples that are misclassified. For misclassified examples, if $\|\mathbf{a}\|$ is small, then at least the hinge loss will be small. By this argument, we would like to achieve a small value of the hinge loss on the training examples using a $\mathbf{a}$ that has small length.

We can do so by adding a penalty term to the hinge loss to favor solutions where $\|\mathbf{a}\|$ is small. To obtain a $\mathbf{a}$ of small length, it is enough to ensure that $(1/2)\mathbf{a}^T\mathbf{a}$ is small (the factor of $1/2$ makes the gradient cleaner). This penalty term will ensure that there is a unique choice of classifier parameters in the case the hinge loss is zero. Experience (and some theory we can't go into here) shows that having a small $\|\mathbf{a}\|$ helps even if there is no pair that classifies all training examples correctly. Doing so improves the error on future examples. Adding a penalty term to improve the solution of a learning problem is sometimes referred to as **regularization**. The penalty term is often referred to as a **regularizer**, because it tends to discourage solutions that are large (and so have possible high loss on future test data) but are not strongly supported by the training data. The parameter λ is often referred to as the **regularization parameter**.

Using the hinge loss to form the training cost, and regularizing with a penalty term $(1/2)\mathbf{a}^T\mathbf{a}$ means our cost function is:

$$S(\mathbf{a}, b; \lambda) = \left[(1/N)\sum_{i=1}^{N} \max(0, 1 - y_i\left(\mathbf{a}^T\mathbf{x}_i + b\right))\right] + \lambda\left(\frac{\mathbf{a}^T\mathbf{a}}{2}\right).$$

There are now two problems to solve. First, assume we know λ; we will need to find $\mathbf{a}$ and b that minimize $S(\mathbf{a}, b; \lambda)$. Second, we have no theory that tells us how to choose λ, so we will need to search for a good value.

3.1.3 Finding a Classifier with Stochastic Gradient Descent

The usual recipes for finding a minimum are ineffective for our cost function. First, write $\mathbf{u} = [\mathbf{a}, b]$ for the vector obtained by stacking the vector $\mathbf{a}$ together with b. We have a function $g(\mathbf{u})$, and we wish to obtain a value of $\mathbf{u}$ that achieves the minimum for that function. Sometimes we can solve a problem like this by constructing the gradient and finding a value of $\mathbf{u}$ the makes the gradient zero, but not this time (try it; the max creates problems). We must use a numerical method.

Typical numerical methods take a point $\mathbf{u}^{(n)}$, update it to $\mathbf{u}^{(n+1)}$, then check to see whether the result is a minimum. This process is started from a start point. The choice of start point may or may not matter for general problems, but for our problem a random start point is fine. The update is usually obtained by computing a direction $\mathbf{p}^{(n)}$ such that for small values of η, $g(\mathbf{u}^{(n)} + \eta\mathbf{p}^{(n)})$ is smaller than $g(\mathbf{u}^{(n)})$.

Such a direction is known as a **descent direction**. We must then determine how far to go along the descent direction, a process known as **line search**.

Obtaining a descent direction: One method to choose a descent direction is **gradient descent**, which uses the negative gradient of the function. Recall our notation that

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ \dots \\ u_d \end{pmatrix}$$

and that

$$\nabla g = \begin{pmatrix} \frac{\partial g}{\partial u_1} \\ \frac{\partial g}{\partial u_2} \\ \dots \\ \frac{\partial g}{\partial u_d} \end{pmatrix}.$$

We can write a Taylor series expansion for the function $g(\mathbf{u}^{(n)} + \eta\mathbf{p}^{(n)})$. We have that

$$g(\mathbf{u}^{(n)} + \eta\mathbf{p}^{(n)}) = g(\mathbf{u}^{(n)}) + \eta\left[(\nabla g)^T\mathbf{p}^{(n)}\right] + O(\eta^2)$$

This means that we can expect that if

$$\mathbf{p}^{(n)} = -\nabla g(\mathbf{u}^{(n)}),$$

we expect that, at least for small values of h, $g(\mathbf{u}^{(n)} + \eta\mathbf{p}^{(n)})$ will be less than $g(\mathbf{u}^{(n)})$. This works (as long as g is differentiable, and quite often when it isn't) because g must go down for at least small steps in this direction.

But recall that our cost function is a sum of a penalty term and one error cost per example. This means the cost function looks like

$$g(\mathbf{u}) = \left[(1/N)\sum_{i=1}^{N} g_i(\mathbf{u})\right] + g_0(\mathbf{u}),$$

as a function of $\mathbf{u}$. Gradient descent would require us to form

$$-\nabla g(\mathbf{u}) = -\left(\left[(1/N)\sum_{i=1}^{N} \nabla g_i(\mathbf{u})\right] + \nabla g_0(\mathbf{u})\right)$$

and then take a small step in this direction. But if N is large, this is unattractive, as we might have to sum a lot of terms. This happens a lot in building classifiers, where you might quite reasonably expect to deal with millions (billions; perhaps trillions) of examples. Touching each example at each step really is impractical.

Stochastic gradient descent is an algorithm that replaces the exact gradient with an approximation that has a random error, but is simple and quick to compute. The term

$$(\frac{1}{N})\sum_{i=1}^{N} \nabla g_i(\mathbf{u}).$$

is a population mean, and we know how to deal with those. We can estimate this term by drawing a random sample (a **batch**) of N_b (the **batch size**) examples, with replacement, from the population of N examples, then computing the mean for that sample. We approximate the population mean by

$$(\frac{1}{N_b}) \sum_{j \in \text{batch}} \nabla g_j(\mathbf{u}).$$

The batch size is usually determined using considerations of computer architecture (how many examples fit neatly into cache?) or of database design (how many examples are recovered in one disk cycle?). One common choice is $N_b = 1$, which is the same as choosing one example uniformly and at random. We form

$$\mathbf{p}_{N_b}^{(n)} = -\left(\left[(1/N_b) \sum_{j \in \text{batch}} \nabla g_i(\mathbf{u}) \right] + \nabla g_0(\mathbf{u}) \right)$$

and then take a small step along $\mathbf{p}_{N_b}^{(n)}$. Our new point becomes

$$\mathbf{u}^{(n+1)} = \mathbf{u}^{(n)} + \eta \mathbf{p}_{N_b}^{(n)},$$

where η is called the **steplength** (or sometimes **step size** or **learning rate**, even though it isn't the size or the length of the step we take, or a rate!).

Because the expected value of the sample mean is the population mean, if we take many small steps along $\mathbf{p}_{N_b}$, they should average out to a step backwards along the gradient. This approach is known as stochastic gradient descent because we're not going along the gradient, but along a random vector which is the gradient only in expectation. It isn't obvious that stochastic gradient descent is a good idea. Although each step is easy to take, we may need to take more steps. The question is then whether we gain in the increased speed of the step what we lose by having to take more steps. Not much is known theoretically, but in practice the approach is hugely successful for training classifiers.

Choosing a steplength: Choosing a steplength η takes some work. We can't search for the step that gives us the best value of g, because we don't want to evaluate the function g (doing so involves looking at each of the g_i terms). Instead, we use an η that is large at the start — so that the method can explore large changes in the values of the classifier parameters — and small steps later — so that it settles down. The choice of how η gets smaller is often known as a **steplength schedule**.

Here are useful examples of steplength schedules. Often, you can tell how many steps are required to have seen the whole dataset; this is called an **epoch**. A common steplength schedule sets the steplength in the e'th epoch to be

$$\eta^{(e)} = \frac{m}{e+n},$$

where m and n are constants chosen by experiment with small subsets of the dataset. When there are a lot of examples, an epoch is a long time to fix the steplength, and

this approach can reduce the steplength too slowly. Instead, you can divide training into what I shall call **seasons** (blocks of a fixed number of iterations, smaller than epochs), and make the steplength a function of the season number.

There is no good test for whether stochastic gradient descent has converged to the right answer, because natural tests involve evaluating the gradient and the function, and doing so is expensive. More usual is to plot the error as a function of iteration on the validation set, and interrupt or stop training when the error has reached an acceptable level. The error (resp. accuracy) should vary randomly (because the steps are taken in directions that only approximate the gradient) but should decrease (resp. increase) overall as training proceeds (because the steps do approximate the gradient). Figures 3.2 and 3.3 show examples of these curves, which are sometimes known as **learning curves**.

3.1.4 Searching for λ

We do not know a good value for λ. We will obtain a value by choosing a set of different values, fitting an SVM using each value, and taking the λ value that will yield the best SVM. Experience has shown that the performance of a method is not profoundly sensitive to the value of λ, so that we can look at values spaced quite far apart. It is usual to take some small number (say, $1e-4$), then multiply by powers of 10 (or 3, if you're feeling fussy and have a fast computer). So, for example, we might look at $\lambda \in \{1e-4, 1e-3, 1e-2, 1e-1\}$. We know how to fit an SVM to a particular value of λ (Section 3.1.3). The problem is to choose the value that yields the best SVM, and to use that to get the best classifier.

We have seen a version of this problem before (Section 2.3.1). There, we chose from several different types of model to obtain the best naive bayes classifier. The recipe from that section is easily adapted to the current problem. We regard each different λ value as representing a different model. We split the data into two pieces: one is a training set, used for fitting and choosing models; the other is a test set, used for evaluating the final chosen model.

Now for each value of λ, compute the cross-validated error of an SVM using that λ on the training set. Do this by repeatedly splitting the training set into two pieces (training and validation); fitting the SVM with that λ to the training piece using stochastic gradient descent; evaluating the error on the validation piece; and averaging these errors. Now use the cross-validated error to choose the best λ value. Very often this just means you choose the value that produces the lowest cross-validated error, but there might be cases where two values produce about the same error and one is preferred for some other reason. Notice that you can compute the standard deviation of the cross-validated error as well as the mean, so you can tell whether differences between cross-validated errors are significant.

Now take the entire training set, and use this to fit an SVM for the chosen λ value. This should be (a little) better than any of the SVMs obtained in the cross-validation, because it uses (slightly) more data. Finally, evaluate the resulting SVM on the test set.

This procedure is rather harder to describe than to do (there's a pretty natural set of nested loops here). There are some strong advantages. First, the estimate of how well a particular SVM type works is unbiased, because we evaluated on data

not used on training. Second, once you have chosen the cross-validation parameter, the SVM you fit is the best you can fit because you used all the training set to obtain it. Finally, your estimate of how well that particular SVM works is unbiased, too, because you obtained it using data that wasn't used to train or to select a model.

3.1.5 Example: Training an SVM with Stochastic Gradient Descent

I have summarized the SVM training procedure in a set of boxes, below. You should be aware that the recipe there admits many useful variations, though. One useful practical trick is to rescale the feature vector components so each has unit variance. This doesn't change anything conceptual as the best choice of decision boundary for rescaled data is easily derived from the best choice for unscaled, and vice versa. Rescaling very often makes stochastic gradient descent perform better because the method takes steps that are even in each component.

It is quite usual to use packages to fit SVM's, and good packages may use a variety of tricks which we can't go into to make training more efficient. Nonetheless, you should have a grasp of the overall process, because it follows a pattern that is useful for training other models (among other things, most deep networks are trained using this pattern).

Procedure: 3.1 *Training an SVM: Overall*

Start with a dataset containing N pairs $(\mathbf{x}_i, y_i)$. Each $\mathbf{x}_i$ is a d-dimensional feature vector, and each y_i is a label, either 1 or -1. Optionally, rescale the $\mathbf{x}_i$ so that each component has unit variance. Choose a set of possible values of the regularization weight λ. Separate the dataset into two sets: test and training. Reserve the test set. For each value of the regularization weight, use the training set to estimate the accuracy of an SVM with that λ value, using cross-validation as in procedure 3.2 and stochastic gradient descent. Use this information to choose λ_0, the best value of λ (usually, the one that yields the highest accuracy). Now use the training set to fit the best SVM using λ_0 as the regularization constant. Finally, use the test set to compute the accuracy or error rate of that SVM, and report that

To train an SVM

Procedure: 3.2 *Training an SVM: estimating the accuracy*

Repeatedly: split the training dataset into two components (training and validation), at random; use the training component to train an SVM; and compute the accuracy on the validation component. Now average the resulting accuracy values.

To estimate accuracy of an SVM with known λ

Procedure: 3.3 *Training an SVM: stochastic gradient descent*

Obtain $\mathbf{u} = (\mathbf{a}, b)$ by stochastic gradient descent on the cost function

$$g(\mathbf{u}) = \left[(1/N) \sum_{i=1}^{N} g_i(\mathbf{u}) \right] + g_0(\mathbf{u})$$

where $g_0(\mathbf{u}) = \lambda(\mathbf{a}^T\mathbf{a})/2$ and $g_i(\mathbf{u}) = \max(0, 1 - y_i\left(\mathbf{a}^T\mathbf{x}_i + b\right))$.
Do so by first choosing a fixed number of items per batch N_b, the number of steps per season N_s, and the number of steps k to take before evaluating the model (this is usually a lot smaller than N_s). Choose a random start point. Now iterate:

- Update the stepsize. In the s'th season, the step size is typically $\eta^{(s)} = \frac{m}{s+n}$ for constants m and n chosen by small-scale experiments.
- Split the training dataset into a training part and a validation part. This split changes each season. Use the validation set to get an unbiased estimate of error during that season's training.
- Now, until the end of the season (i.e. until you have taken N_s steps):
  - Take k steps. Each step is taken by selecting a batch of N_b data items uniformly and at random from the training part for that season. Write $\mathcal{D}$ for this set. Now compute

$$\mathbf{p}^{(n)} = -\frac{1}{N_b}\left(\sum_{i \in \mathcal{D}} \nabla g_i(\mathbf{u}^{(n)})\right) - \lambda\mathbf{u}^{(n)},$$

    and update the model by computing

$$\mathbf{u}^{(n+1)} = \mathbf{u}^{(n)} + \eta\mathbf{p}^{(n)}$$

  - Evaluate the current model $\mathbf{u}^{(n)}$ by computing the accuracy on the validation part for that season. Plot the accuracy as a function of step number.

There are two ways to stop. You can choose a fixed number of seasons (or of epochs) and stop when that is done. Alternatively, you can watch the error plot and stop when the error reaches some level or meets some criterion.

To fit an SVM with stochastic gradient descent

Here is an example in some detail. I downloaded the dataset at http://archive.ics.uci.edu/ml/datasets/Adult. This dataset apparently contains 48, 842 data items,

but I worked with only the first 32, 000. Each consists of a set of numeric and categorical features describing a person, together with whether their annual income is larger than or smaller than 50K$. I ignored the categorical features to prepare these figures. This isn't wise if you want a good classifier, but it's fine for an example. I used these features to predict whether income is over or under 50K$. I split the data into 5, 000 test examples, and 27,000 training examples. It's important to do so at random. There are 6 numerical features. I subtracted the mean (which doesn't usually make much difference) and rescaled each so that the variance was 1 (which is often very important).

Setting up stochastic gradient descent: We have estimates $\mathbf{a}^{(n)}$ and $b^{(n)}$ of the classifier parameters, and we want to improve the estimates. I used a batch size of $N_b = 1$. Pick the r'th example at random. The gradient is

$$\nabla \left(\max(0, 1 - y_r \left(\mathbf{a}^T \mathbf{x}_r + b \right)) + \frac{\lambda}{2} \mathbf{a}^T \mathbf{a} \right).$$

Assume that $y_k \left(\mathbf{a}^T \mathbf{x}_r + b \right) > 1$. In this case, the classifier predicts a score with the right sign, and a magnitude that is greater than one. Then the first term is zero, and the gradient of the second term is easy. Now if $y_k \left(\mathbf{a}^T \mathbf{x}_r + b \right) < 1$, we can ignore the max, and the first term is $1 - y_r \left(\mathbf{a}^T \mathbf{x}_r + b \right)$; the gradient is again easy. If $y_r \left(\mathbf{a}^T \mathbf{x}_r + b \right) = 1$, there are two distinct values we could choose for the gradient, because the max term isn't differentiable. It does not matter which value we choose because this situation hardly ever happens. We choose a steplength η, and update our estimates using this gradient. This yields:

$$\mathbf{a}^{(n+1)} = \mathbf{a}^{(n)} - \eta \begin{cases} \lambda \mathbf{a} & \text{if } y_k \left(\mathbf{a}^T \mathbf{x}_k + b \right) \geq 1 \\ \lambda \mathbf{a} - y_k \mathbf{x} & \text{otherwise} \end{cases}$$

and

$$b^{(n+1)} = b^{(n)} - \eta \begin{cases} 0 & \text{if } y_k \left(\mathbf{a}^T \mathbf{x}_k + b \right) \geq 1 \\ -y_k & \text{otherwise} \end{cases}.$$

Training: I used two different training regimes. In the first training regime, there were 100 seasons. In each season, I applied 426 steps. For each step, I selected one data item uniformly at random (sampling with replacement), then stepped down the gradient. This means the method sees a total of 42, 600 data items. This means that there is a high probability it has touched each data item once (27, 000 isn't enough, because we are sampling with replacement, so some items get seen more than once). I chose 5 different values for the regularization parameter and trained with a steplength of $1/(0.01 * s + 50)$, where s is the season. At the end of each season, I computed $\mathbf{a}^T \mathbf{a}$ and the accuracy (fraction of examples correctly classified) of the current classifier on the held out test examples. Figure 3.2 shows the results. You should notice that the accuracy changes slightly each season; that for larger regularizer values $\mathbf{a}^T \mathbf{a}$ is smaller; and that the accuracy settles down to about 0.8 very quickly.

In the second training regime, there were 100 seasons. In each season, I applied 50 steps. For each step, I selected one data item uniformly at random (sampling with replacement), then stepped down the gradient. This means the method sees a total of 5,000 data items, and about 3,000 unique data items — it hasn't seen the

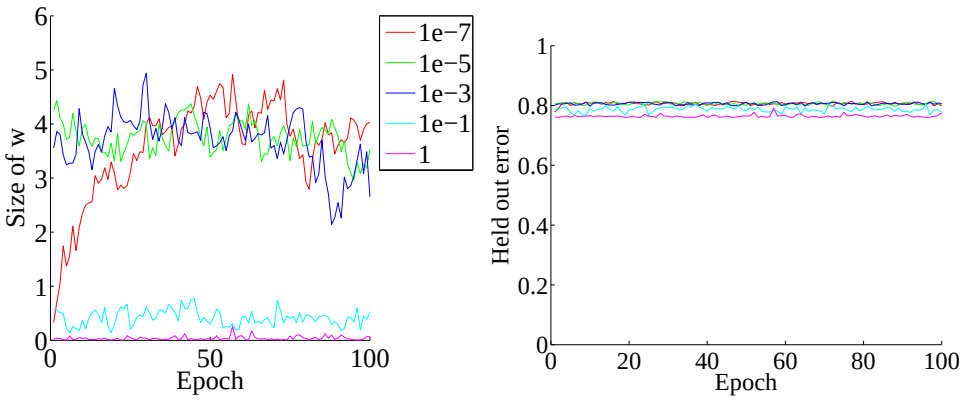


FIGURE 3.2: *On the* **left**, *the magnitude of the weight vector* $\mathbf{a}$ *at the end of each season for the first training regime described in the text. On the* **right**, *the accuracy on held out data at the end of each season. Notice how different choices of regularization parameter lead to different magnitudes of* $\mathbf{a}$*; how the method isn't particularly sensitive to choice of regularization parameter (they change by factors of 100); how the accuracy settles down fairly quickly; and how overlarge values of the regularization parameter do lead to a loss of accuracy.*

whole training set. I chose 5 different values for the regularization parameter and trained with a steplength of $1/(0.01 * s + 50)$, where s is the season. At the end of each season, I computed $\mathbf{a}^T\mathbf{a}$ and the accuracy (fraction of examples correctly classified) of the current classifier on the held out test examples. Figure 3.3 shows the results.

This is an easy classification example. Points worth noting are

- the accuracy makes large changes early, then settles down to make slight changes each season;
- quite large changes in regularization constant have small effects on the outcome, but there is a best choice;
- for larger values of the regularization constant, $\mathbf{a}^T\mathbf{a}$ is smaller;
- there isn't much difference between the two training regimes;
- and the method doesn't need to see all the training data to produce a classifier that is about as good as it would be if the method *had* seen all training data.

All of these points are relatively typical of SVM's trained using stochastic gradient descent with very large datasets.

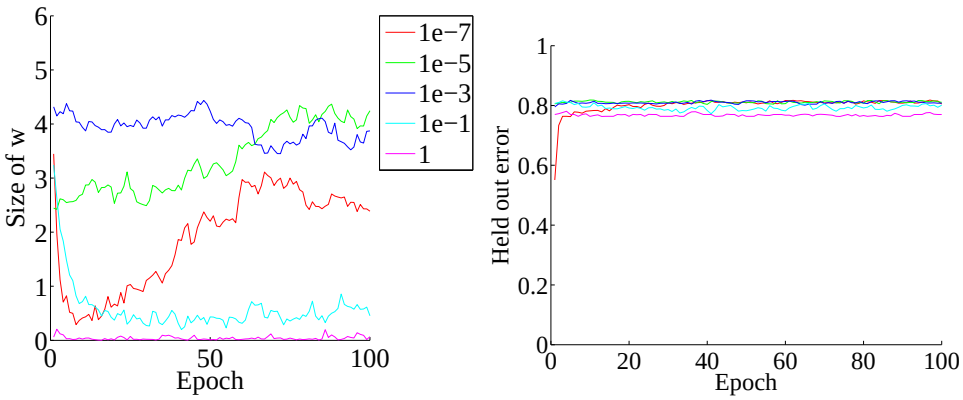


FIGURE 3.3: *On the* **left**, *the magnitude of the weight vector* $\mathbf{a}$ *at the end of each season for the second training regime described in the text. On the* **right**, *the accuracy on held out data at the end of each season. Notice how different choices of regularization parameter lead to different magnitudes of* $\mathbf{a}$*; how the method isn't particularly sensitive to choice of regularization parameter (they change by factors of 100); how the accuracy settles down fairly quickly; and how overlarge values of the regularization parameter do lead to a loss of accuracy.*

Remember this: *Linear SVM's are a go-to classifier. When you have a binary classification problem, the first step should be to try a linear SVM. Training with stochastic gradient descent is straightforward, and extremely effective. Finding an appropriate value of the regularization constant requires an easy search. There is an immense quantity of good software available.*

3.1.6 Multi-Class Classification with SVMs

I have shown how one trains a linear SVM to make a binary prediction (i.e. predict one of two outcomes). But what if there are three, or more, labels? In principle, you could write a binary code for each label, then use a different SVM to predict each bit of the code. It turns out that this doesn't work terribly well, because an error by one of the SVM's is usually catastrophic.

There are two methods that are widely used. In the **all-vs-all** approach, we train a binary classifier for each pair of classes. To classify an example, we present it to each of these classifiers. Each classifier decides which of two classes the example belongs to, then records a vote for that class. The example gets the class label with the most votes. This approach is simple, but scales very badly with the number of classes (you have to build $O(N^2)$ different SVM's for N classes).

In the **one-vs-all** approach, we build a binary classifier for each class. This

classifier must distinguish its class from all the other classes. We then take the class with the largest classifier score. One can think up quite good reasons this approach shouldn't work. For one thing, the classifier isn't told that you intend to use the score to tell similarity between classes. In practice, the approach works rather well and is quite widely used. This approach scales a bit better with the number of classes ($O(N)$).

Remember this: *It is straightforward to build a multi-class classifier out of binary classifiers. Any decent SVM package will do this for you.*

3.2 CLASSIFYING WITH RANDOM FORESTS

I described a classifier as a rule that takes a feature, and produces a class. One way to build such a rule is with a sequence of simple tests, where each test is allowed to use the results of all previous tests. This class of rule can be drawn as a tree (Figure **??**), where each node represents a test, and the edges represent the possible outcomes of the test. To classify a test item with such a tree, you present it to the first node; the outcome of the test determines which node it goes to next; and so on, until the example arrives at a leaf. When it does arrive at a leaf, we label the test item with the most common label in the leaf. This object is known as a **decision tree**. Notice one attractive feature of this decision tree: it deals with multiple class labels quite easily, because you just label the test item with the most common label in the leaf that it arrives at when you pass it down the tree.

Figure 3.5 shows a simple 2D dataset with four classes, next to a decision tree that will correctly classify at least the training data. Actually classifying data with a tree like this is straightforward. We take the data item, and pass it down the tree. Notice it can't go both left and right, because of the way the tests work. This means each data item arrives at a single leaf. We take the most common label at the leaf, and give that to the test item. In turn, this means we can build a geometric structure on the feature space that corresponds to the decision tree. I have illustrated that structure in figure 3.5, where the first decision splits the feature space in half (which is why the term split is used so often), and then the next decisions split each of those halves into two.

The important question is how to get the tree from data. It turns out that the best approach for building a tree incorporates a great deal of randomness. As a result, we will get a different tree each time we train a tree on a dataset. None of the individual trees will be particularly good (they are often referred to as "weak learners"). The natural thing to do is to produce many such trees (a **decision forest**), and allow each to vote; the class that gets the most votes, wins. This strategy is extremely effective.

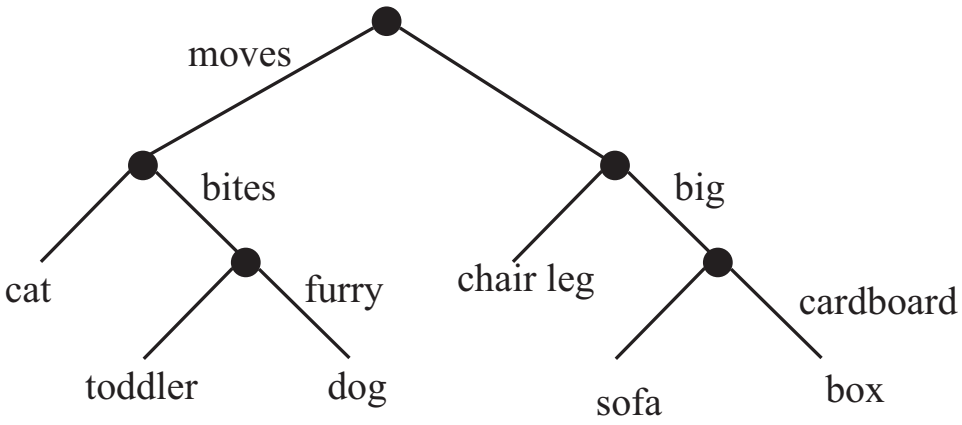


FIGURE 3.4: *This — the household robot's guide to obstacles — is a typical decision tree. I have labelled only one of the outgoing branches, because the other is the negation. So if the obstacle moves, bites, but isn't furry, then it's a toddler. In general, an item is passed down the tree until it hits a leaf. It is then labelled with the leaf's label.*

3.2.1 Building a Decision Tree

There are many algorithms for building decision trees. We will use an approach chosen for simplicity and effectiveness; be aware there are others. We will always use a binary tree, because it's easier to describe and because that's usual (it doesn't change anything important, though). Each node has a **decision function**, which takes data items and returns either 1 or -1.

We train the tree by thinking about its effect on the training data. We pass the whole pool of training data into the root. Any node splits its incoming data into two pools, left (all the data that the decision function labels 1) and right (ditto, -1). Finally, each leaf contains a pool of data, which it can't split because it is a leaf.

Training the tree uses a straightforward algorithm. First, we choose a class of decision functions to use at each node. It turns out that a very effective algorithm is to choose a single feature at random, then test whether its value is larger than, or smaller than a threshold. For this approach to work, one needs to be quite careful about the choice of threshold, which is what we describe in the next section. Some minor adjustments, described below, are required if the feature chosen isn't ordinal. Surprisingly, being clever about the choice of *feature* doesn't seem add a great deal of value. We won't spend more time on other kinds of decision function, though there are lots.

Now assume we use a decision function as described, and we know how to choose a threshold. We start with the root node, then recursively either split the pool of data at that node, passing the left pool left and the right pool right, or stop splitting and return. Splitting involves choosing a decision function from the class to give the "best" split for a leaf. The main questions are how to choose the best

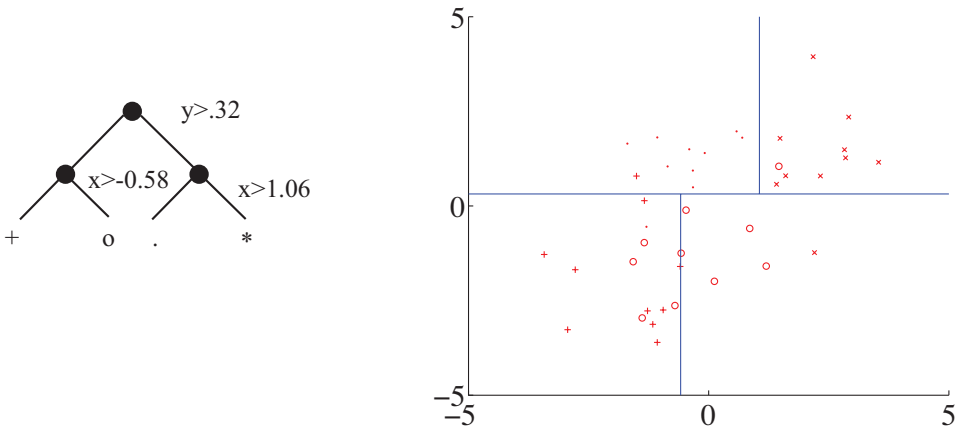


FIGURE 3.5: *A straightforward decision tree, illustrated in two ways. On the* **left**, *I have given the rules at each split; on the* **right**, *I have shown the data points in two dimensions, and the structure that the tree produces in the feature space.*

split (next section), and when to stop.

Stopping is relatively straightforward. Quite simple strategies for stopping are very good. It is hard to choose a decision function with very little data, so we must stop splitting when there is too little data at a node. We can tell this is the case by testing the amount of data against a threshold, chosen by experiment. If all the data at a node belongs to a single class, there is no point in splitting. Finally, constructing a tree that is too deep tends to result in generalization problems, so we usually allow no more than a fixed depth D of splits. Choosing the best splitting threshold is more complicated.

Figure 3.6 shows two possible splits of a pool of training data. One is quite obviously a lot better than the other. In the good case, the split separates the pool into positives and negatives. In the bad case, each side of the split has the same number of positives and negatives. We cannot usually produce splits as good as the good case here. What we are looking for is a split that will make the proper label more certain.

Figure 3.7 shows a more subtle case to illustrate this. The splits in this figure are obtained by testing the horizontal feature against a threshold. In one case, the left and the right pools contain about the same fraction of positive ('x') and negative ('o') examples. In the other, the left pool is all positive, and the right pool is mostly negative. This is the better choice of threshold. If we were to label any item on the left side positive and any item on the right side negative, the error rate would be fairly small. If you count, the best error rate for the informative split is 20% on the training data, and for the uninformative split it is 40% on the training data.

But we need some way to score the splits, so we can tell which threshold is best. Notice that, in the uninformative case, knowing that a data item is on the left (or the right) does not tell me much more about the data than I already knew.

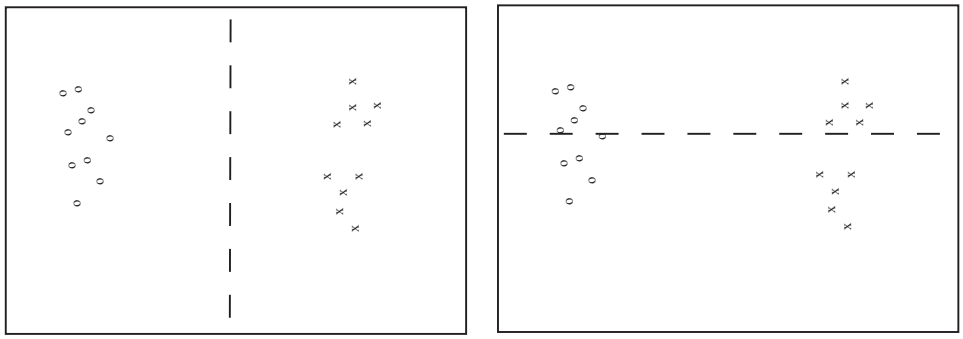


FIGURE 3.6: *Two possible splits of a pool of training data. Positive data is represented with an 'x', negative data with a 'o'. Notice that if we split this pool with the informative line, all the points on the left are 'o's, and all the points on the right are 'x's. This is an excellent choice of split — once we have arrived in a leaf, everything has the same label. Compare this with the less informative split. We started with a node that was half 'x' and half 'o', and now have two nodes each of which is half 'x' and half 'o' — this isn't an improvement, because we do not know more about the label as a result of the split.*

We have that $p(1|\text{left pool, uninformative}) = 2/3 \approx 3/5 = p(1|\text{parent pool})$ and $p(1|\text{right pool, uninformative}) = 1/2 \approx 3/5 = p(1|\text{parent pool})$. For the informative pool, knowing a data item is on the left classifies it completely, and knowing that it is on the right allows us to classify it an error rate of 1/3. The informative split means that my uncertainty about what class the data item belongs to is significantly reduced if I know whether it goes left or right. To choose a good threshold, we need to keep track of how informative the split is.

3.2.2 Choosing a Split with Information Gain

Write $\mathcal{P}$ for the set of all data at the node. Write $\mathcal{P}_l$ for the left pool, and $\mathcal{P}_r$ for the right pool. The entropy of a pool $\mathcal{C}$ scores how many bits would be required to represent the class of an item in that pool, on average. Write $n(i; \mathcal{C})$ for the number of items of class i in the pool, and $N(\mathcal{C})$ for the number of items in the pool. Then the entropy $H(\mathcal{C})$ of the pool $\mathcal{C}$ is

$$-\sum_i \frac{n(i;\mathcal{C})}{N(\mathcal{C})} \log_2 \frac{n(i;\mathcal{C})}{N(\mathcal{C}}.$$

It is straightforward that $H(\mathcal{P})$ bits are required to classify an item in the parent pool $\mathcal{P}$. For an item in the left pool, we need $H(\mathcal{P}_l)$ bits; for an item in the right pool, we need $H(\mathcal{P}_r)$ bits. If we split the parent pool, we expect to encounter items in the left pool with probability

$$\frac{N(\mathcal{P}_l)}{N(\mathcal{P})}$$

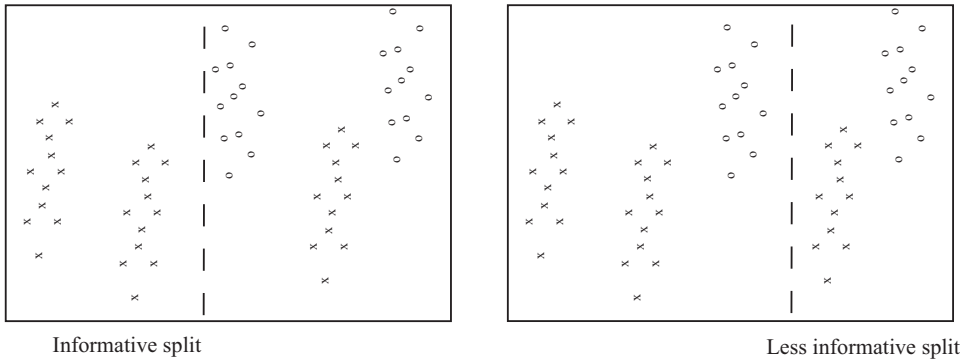


FIGURE 3.7: *Two possible splits of a pool of training data. Positive data is represented with an 'x', negative data with a 'o'. Notice that if we split this pool with the informative line, all the points on the left are 'x's, and two-thirds of the points on the right are 'o's. This means that knowing which side of the split a point lies would give us a good basis for estimating the label. In the less informative case, about two-thirds of the points on the left are 'x's and about half on the right are 'x's — knowing which side of the split a point lies is much less useful in deciding what the label is.*

and items in the right pool with probability

$$\frac{N(\mathcal{P}_r)}{N(\mathcal{P})}.$$

This means that, on average, we must supply

$$\frac{N(\mathcal{P}_l)}{N(\mathcal{P})}H(\mathcal{P}_l) + \frac{N(\mathcal{P}_r)}{N(\mathcal{P})}H(\mathcal{P}_r)$$

bits to classify data items if we split the parent pool. Now a good split is one that results in left and right pools that are informative. In turn, we should need fewer bits to classify once we have split than we need before the split. You can see the difference

$$I(\mathcal{P}_l, \mathcal{P}_r; \mathcal{P}) = H(\mathcal{P}) - \left(\frac{N(\mathcal{P}_l)}{N(\mathcal{P})}H(\mathcal{P}_l) + \frac{N(\mathcal{P}_r)}{N(\mathcal{P})}H(\mathcal{P}_r) \right)$$

as the **information gain** caused by the split. This is the average number of bits that you *don't* have to supply if you know which side of the split an example lies. Better splits have larger information gain.

Recall that our decision function is to choose a feature at random, then test its value against a threshold. Any data point where the value is larger goes to the left pool; where the value is smaller goes to the right. This may sound much too simple to work, but it is actually effective and popular. Assume that we are at a node, which we will label k. We have the pool of training examples that have reached that node. The i'th example has a feature vector $\mathbf{x}_i$, and each of these feature vectors is a d dimensional vector.

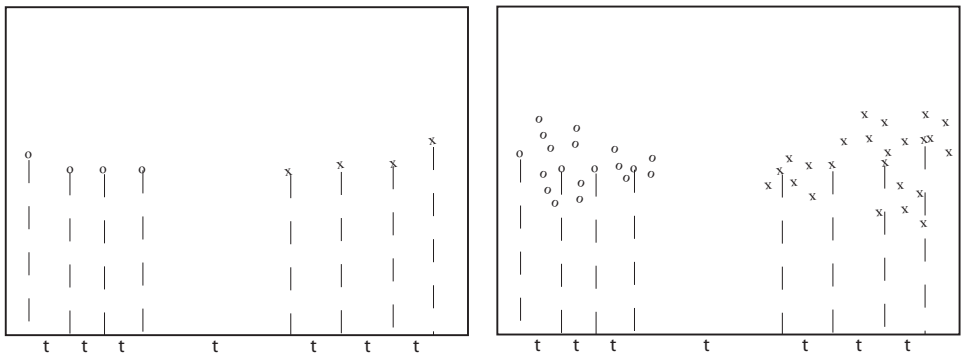


FIGURE 3.8: *We search for a good splitting threshold by looking at values of the chosen component that yield different splits. On the* **left***, I show a small dataset and its projection onto the chosen splitting component (the horizontal axis). For the 8 data points here, there are only 7 threshold values that produce interesting splits, and these are shown as 't's on the axis. On the* **right***, I show a larger dataset; in this case, I have projected only a subset of the data, which results in a small set of thresholds to search.*

We choose an integer j in the range $1 \dots d$ uniformly and at random. We will split on this feature, and we store j in the node. Recall we write $x_i^{(j)}$ for the value of the j'th component of the i'th feature vector. We will choose a threshold t_k, and split by testing the sign of $x_i^{(j)} - t_k$. Choosing the value of t_k is easy. Assume there are N_k examples in the pool. Then there are $N_k - 1$ possible values of t_k that lead to different splits. To see this, sort the N_k examples by $x^{(j)}$, then choose values of t_k halfway between example values (Figure 3.8). For each of these values, we compute the information gain of the split. We then keep the threshold with the best information gain.

We can elaborate this procedure in a useful way, by choosing m features at random, finding the best split for each, then keeping the feature and threshold value that is best. It is important that m is a lot smaller than the total number of features — a usual root of thumb is that m is about the square root of the total number of features. It is usual to choose a single m, and choose that for all the splits.

Now assume we happen to have chosen to work with a feature that isn't ordinal, and so can't be tested against a threshold. A natural, and effective, strategy is as follows. We can split such a feature into two pools by flipping an unbiased coin for each value — if the coin comes up H, any data point with that value goes left, and if it comes up T, any data point with that value goes right. We chose this split at random, so it might not be any good. We can come up with a good split by repeating this procedure F times, computing the information gain for each split, then keeping the one that has the best information gain. We choose F in advance, and it usually depends on the number of values the categorical variable can take.

We now have a relatively straightforward blueprint for an algorithm, which I have put in a box. It's a blueprint, because there are a variety of ways in which it

can be revised and changed.

Procedure: 3.4 *Building a decision tree: overall*

We have a dataset containing N pairs $(\mathbf{x}_i, y_i)$. Each x_i is a d-dimensional feature vector, and each y_i is a label. Call this dataset a **pool**. Now recursively apply the following procedure:

- If the pool is too small, or if all items in the pool have the same label, or if the depth of the recursion has reached a limit, stop.
- Otherwise, search the features for a good split that divides the pool into two, then apply this procedure to each child.

We search for a good split by the following procedure:

- Choose a subset of the feature components at random. Typically, one uses a subset whose size is about the square root of the feature dimension.
- For each component of this subset, search for a good split. If the component is ordinal, do so using the procedure of box 3.5, otherwise use the procedure of box 3.6.

Overall approach to build a decision tree

Procedure: 3.5 *Splitting an ordinal feature*

We search for a good split on a given ordinal feature by the following procedure:

- Select a set of possible values for the threshold.
- For each value split the dataset (every data item with a value of the component below the threshold goes left, others go right), and compue the information gain for the split.

Keep the threshold that has the largest information gain.
A good set of possible values for the threshold will contain values that separate the data "reasonably". If the pool of data is small, you can project the data onto the feature component (i.e. look at the values of that component alone), then choose the $N - 1$ distinct values that lie between two data points. If it is big, you can randomly select a subset of the data, then project that subset on the feature component and choose from the values between data points.

To split an ordinal feature in a decision tree

Procedure: 3.6 *Splitting a non-ordinal feature*

Split the values this feature takes into sets pools by flipping an unbiased coin for each value — if the coin comes up H, any data point with that value goes left, and if it comes up T, any data point with that value goes right. Repeating this procedure F times, computing the information gain for each split, then keep the split that has the best information gain. We choose F in advance, and it usually depends on the number of values the categorical variable can take.

To split a non-ordinal feature in a decision tree

3.2.3 Forests

A single decision tree tends to yield poor classifications. One reason is because the tree is not chosen to give the best classification of its training data. We used a random selection of splitting variables at each node, so the tree can't be the "best possible". Obtaining the best possible tree presents significant technical difficulties. It turns out that the tree that gives the best possible results on the training data can perform rather poorly on test data. The training data is a small subset of possible examples, and so must differ from the test data. The best possible tree on the training data might have a large number of small leaves, built using carefully chosen splits. But the choices that are best for training data might not be best for test data.

Rather than build the best possible tree, we have built a tree efficiently, but with number of random choices. If we were to rebuild the tree, we would obtain a different result. This suggests the following extremely effective strategy: build many trees, and classify by merging their results.

3.2.4 Building and Evaluating a Decision Forest

There are two important strategies for building and evaluating decision forests. I am not aware of evidence strongly favoring one over the other, but different software packages use different strategies, and you should be aware of the options. In one strategy, we separate labelled data into a training and a test set. We then build multiple decision trees, training each using the whole training set. Finally, we evaluate the forest on the test set. In this approach, the forest has not seen some fraction of the available labelled data, because we used it to test. However, each tree has seen every training data item.

Procedure: 3.7 *Building a decision forest*

We have a dataset containing N pairs $(\mathbf{x}_i, y_i)$. Each $\mathbf{x}_i$ is a d-dimensional feature vector, and each y_i is a label. Separate the dataset into a test set and a training set. Train multiple distinct decision trees on the training set, recalling that the use of a random set of components to find a good split means you will obtain a distinct tree each time.

In the other strategy, sometimes called **bagging**, each time we train a tree we randomly subsample the labelled data with replacement, to yield a training set the same size as the original set of labelled data. Notice that there will be duplicates in this training set, which is like a bootstrap replicate. This training set is often called a **bag**. We keep a record of the examples that do not appear in the bag (the "out of bag" examples). Now to evaluate the forest, we evaluate each tree on its out of bag examples, and average these error terms. In this approach, the entire forest has seen all labelled data, and we also get an estimate of error, but no tree has seen all the training data.

Procedure: 3.8 *Building a decision forest using bagging*

We have a dataset containing N pairs $(\mathbf{x}_i, y_i)$. Each $\mathbf{x}_i$ is a d-dimensional feature vector, and each y_i is a label. Now build k bootstrap replicates of the training data set. Train one decision tree on each replicate.

3.2.5 Classifying Data Items with a Decision Forest

Once we have a forest, we must classify test data items. There are two major strategies. The simplest is to classify the item with each tree in the forest, then take the class with the most votes. This is effective, but discounts some evidence that might be important. For example, imagine one of the trees in the forest has a leaf with many data items with the same class label; another tree has a leaf with exactly one data item in it. One might not want each leaf to have the same vote.

Procedure: 3.9 *Classification with a decision forest*

Given a test example $\mathbf{x}$, pass it down each tree of the forest. Now choose one of the following strategies.

- Each time the example arrives at a leaf, record one vote for the label that occurs most often at the leaf. Now choose the label with the most votes.
- Each time the example arrives at a leaf, record N_l votes for each of the labels that occur at the leaf, where N_l is the number of times the label appears in the training data at the leaf. Now choose the label with the most votes.

An alternative strategy that takes this observation into account is to pass the test data item down each tree. When it arrives at a leaf, we record one vote for each of the training data items in that leaf. The vote goes to the class of the training data item. Finally, we take the class with the most votes. This approach allows big, accurate leaves to dominate the voting process. Both strategies are in use, and I am not aware of compelling evidence that one is always better than the other. This may be because the randomness in the training process makes big, accurate leaves uncommon in practice.

Worked example 3.1 *Classifying heart disease data*

Build a random forest classifier to classify the "heart" dataset from the UC Irvine machine learning repository. The dataset is at http://archive.ics.uci.edu/ml/datasets/Heart+Disease. There are several versions. You should look at the processed Cleveland data, which is in the file "processed.cleveland.data.txt".

Solution: I used the R random forest package. This uses a bagging strategy. This package makes it quite simple to fit a random forest, as you can see. In this dataset, variable 14 (V14) takes the value 0, 1, 2, 3 or 4 depending on the severity of the narrowing of the arteries. Other variables are physiological and physical measurements pertaining to the patient (read the details on the website). I tried to predict all five levels of variable 14, using the random forest as a multivariate classifier. This works rather poorly, as the out-of-bag class confusion matrix below shows. The total out-of-bag error rate was 45%.

		Predict					
		0	1	2	3	4	Class error
True	0	151	7	2	3	1	7.9%
	1	32	5	9	9	0	91%
	2	10	9	7	9	1	81%
	3	6	13	9	5	2	86%
	4	2	3	2	6	0	100%

This is the example of a class confusion matrix from table 2.1. Fairly clearly, one can predict narrowing or no narrowing from the features, but not the degree of narrowing (at least, not with a random forest). So it is natural to quantize variable 14 to two levels, 0 (meaning no narrowing), and 1 (meaning any narrowing, so the original value could have been 1, 2, or 3). I then built a random forest to predict this quantized variable from the other variables. The total out-of-bag error rate was 19%, and I obtained the following out-of-bag class confusion matrix

		Predict		
		0	1	Class error
True	0	138	26	16%
	1	31	108	22%

Notice that the false positive rate (16%, from 26/164) is rather better than the false negative rate (22%). You might wonder whether it is better to train on and predict $0, \ldots, 4$, then quantize the predicted value. If you do this, you will find you get a false positive rate of 7.9%, but a false negative rate that is much higher (36%, from 50/139). In this application, a false negative is likely more of a problem than a false positive, so the tradeoff is unattractive.

Remember this: *Random forests are straightforward to build, and very effective. They can predict any kind of label. Good software implementations are easily available.*

3.3 YOU SHOULD

3.3.1 remember these definitions:

3.3.2 remember these terms:

3.3.3 remember these facts:

3.3.4 use these procedures:

3.3.5 be able to:

- build an SVM using your preferred software package, and produce a cross-validated estimate of its error rate or its accuracy;
- write code to train an SVM using stochastic gradient descent, and produce a cross-validated estimate of its error rate or its accuracy;
- and build a decision forest using your preferred software package, and produce a cross-validated estimate of its error rate or its accuracy.

PROGRAMMING EXERCISES

3.1. The UC Irvine machine learning data repository hosts a collection of data on breast cancer diagnostics, donated by Olvi Mangasarian, Nick Street, and William H. Wolberg. You can find this data at http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic). For each record, there is an id number, 10 continuous variables, and a class (benign or malignant). There are 569 examples. Separate this dataset randomly into 100 validation, 100 test, and 369 training examples.

Write a program to train a support vector machine on this data using stochastic gradient descent. You should not use a package to train the classifier (you don't really need one), but your own code. You should ignore the id number, and use the continuous variables as a feature vector. You should scale these variables so each has unit variance. You should search for an appropriate value of the regularization constant, trying at least the values $\lambda = [1e-3, 1e-2, 1e-1, 1]$. Use the validation set for this search.

You should use at least 50 epochs of at least 100 steps each. In each epoch, you should separate out 50 training examples at random for evaluation. You should compute the accuracy of the current classifier on the set held out for the epoch every 10 steps. You should produce:

(a) A plot of the accuracy every 10 steps, for each value of the regularization constant.

(b) Your estimate of the best value of the regularization constant, together with a brief description of why you believe that is a good value.

(c) Your estimate of the accuracy of the best classifier on held out data

3.2. The UC Irvine machine learning data repository hosts a collection of data on adult income, donated by Ronny Kohavi and Barry Becker. You can find this data at https://archive.ics.uci.edu/ml/datasets/Adult For each record, there is a set of continuous attributes, and a class $\geq 50K$ or $< 50K$. There are 48842 examples. You should use only the continous attributes (see the description on the web page) and drop examples where there are missing values of the continuous attributes. Separate the resulting dataset randomly into 10% validation, 10% test, and 80% training examples.

Write a program to train a support vector machine on this data using stochastic gradient descent. You should not use a package to train the classifier (you don't really need one), but your own code. You should ignore the id number, and use the continuous variables as a feature vector. You should scale these variables so that each has unit variance. You should search for an appropriate value of the regularization constant, trying at least the values $\lambda = [1e-3, 1e-2, 1e-1, 1]$. Use the validation set for this search

You should use at least 50 epochs of at least 300 steps each. In each epoch, you should separate out 50 training examples at random for evaluation. You should compute the accuracy of the current classifier on the set held out for the epoch every 30 steps. You should produce:

(a) A plot of the accuracy every 30 steps, for each value of the regularization constant.

(b) Your estimate of the best value of the regularization constant, together with a brief description of why you believe that is a good value.

(c) Your estimate of the accuracy of the best classifier on held out data

3.3. The UC Irvine machine learning data repository hosts a collection of data on the whether p53 expression is active or inactive. You can find out what this means, and more information about the dataset, by reading: Danziger, S.A.,

Baronio, R., Ho, L., Hall, L., Salmon, K., Hatfield, G.W., Kaiser, P., and Lathrop, R.H. "Predicting Positive p53 Cancer Rescue Regions Using Most Informative Positive (MIP) Active Learning," *PLOS Computational Biology*, 5(9), 2009; Danziger, S.A., Zeng, J., Wang, Y., Brachmann, R.K. and Lathrop, R.H. "Choosing where to look next in a mutation sequence space: Active Learning of informative p53 cancer rescue mutants", *Bioinformatics*, 23(13), 104-114, 2007; and Danziger, S.A., Swamidass, S.J., Zeng, J., Dearth, L.R., Lu, Q., Chen, J.H., Cheng, J., Hoang, V.P., Saigo, H., Luo, R., Baldi, P., Brachmann, R.K. and Lathrop, R.H. "Functional census of mutation sequence spaces: the example of p53 cancer rescue mutants," *IEEE/ACM transactions on computational biology and bioinformatics*, 3, 114-125, 2006.
You can find this data at https://archive.ics.uci.edu/ml/datasets/p53+Mutants. There are a total of 16772 instances, with 5409 attributes per instance. Attribute 5409 is the class attribute, which is either active or inactive. There are several versions of this dataset. You should use the version `K8.data`.

(a) Train an SVM to classify this data, using stochastic gradient descent. You will need to drop data items with missing values. You should estimate a regularization constant using cross-validation, trying at least 3 values. Your training method should touch at least 50% of the training set data. You should produce an estimate of the accuracy of this classifier on held out data consisting of 10% of the dataset, chosen at random.

(b) Now train a naive bayes classifier to classify this data. You should produce an estimate of the accuracy of this classifier on held out data consisting of 10% of the dataset, chosen at random.

(c) Compare your classifiers. Which one is better? why?

3.4. The UC Irvine machine learning data repository hosts a collection of data on whether a mushroom is edible, donated by Jeff Schlimmer and to be found at http://archive.ics.uci.edu/ml/datasets/Mushroom. This data has a set of categorical attributes of the mushroom, together with two labels (poisonous or edible). Use the R random forest package (as in the example in the chapter) to build a random forest to classify a mushroom as edible or poisonous based on its attributes.

(a) Produce a class-confusion matrix for this problem. If you eat a mushroom based on your classifier's prediction it is edible, what is the probability of being poisoned?

MNIST Exercises

The following exercises are elaborate, but rewarding. The MNIST dataset is a dataset of 60, 000 training and 10, 000 test examples of handwritten digits, originally constructed by Yann Lecun, Corinna Cortes, and Christopher J.C. Burges. It is very widely used to check simple methods. There are 10 classes in total ("0" to "9"). This dataset has been extensively studied, and there is a history of methods and feature constructions at https://en.wikipedia.org/wiki/MNIST_database and at http://yann.lecun.com/exdb/mnist/. You should notice that the best methods perform extremely well. The original dataset is at http://yann.lecun.com/exdb/mnist/. It is stored in an unusual format, described in detail on that website. Writing your own reader is pretty simple, but web search yields readers for standard packages. There is reader code in matlab available (at least) at http://ufldl.stanford.edu/wiki/index.php/Using_the_MNIST_Dataset. There is reader code for R available (at least) at https://stackoverflow.com/questions/21521571/how-to-read-mnist-database-in-r.

The dataset consists of 28×28 images. These were originally binary images, but appear to be grey level images as a result of some anti-aliasing. I will ignore mid grey pixels (there aren't many of them) and call dark pixels "ink pixels", and light pixels "paper pixels". The digit has been centered in the image by centering the center of gravity of the image pixels. Here are some options for re-centering the digits that I will refer to in the exercises.

- **Untouched:** do not re-center the digits, but use the images as is.
- **Bounding box:** construct an $b \times b$ bounding box so that the horizontal (resp. vertical) range of ink pixels is centered in the box.
- **Stretched bounding box:** construct an $b \times b$ bounding box so that the horizontal (resp. vertical) range of ink pixels runs the full horizontal (resp. vertical) range of the box. Obtaining this representation will involve rescaling image pixels: you find the horizontal and vertical ink range, cut that out of the original image, then resize the result to $b \times b$.

Once the image has been re-centered, you can compute features. Here are some options for constructing features that I will refer to in the exercises.

- **Raw pixels:** use the raw pixel values from images.
- **PCA:** project images onto the first d principal components computed for the entire dataset.
- **Local PCA:** first, compute the first d principal components for each digit class separately. Now for any image, compute a $10d$ dimensional feature vector by, for each class, subtracting that class mean from the image, then projecting the image onto the d principal components for that class. Finally, stack all 10 d dimensional features you get. This measures how much the difference between the image and the class mean looks like the difference between images of that class and the class mean.

3.5. Investigate classifying MNIST using naive bayes. Use the procedures of Section 2.3.1 to compare four cases on raw pixel image features. These cases are obtained by choosing either normal model or binomial model for every feature, and untouched images or stretched bounding box images.
(a) Which is the best case?
(b) How accurate is the best case? (remember, the answer to this is *not* obtained by taking the best accuracy from the previous subexercise — check Section 2.3.1 if you're vague on this point).

3.6. Investigate classifying MNIST using nearest neighbors. You will use approximate nearest neighbors. Obtain the FLANN package for approximate nearest neighbors from http://www.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN. To use this package, you should consider first using a function that builds an index for the training dataset (`flann_build_index()`, or variants), then querying with your test points (`flann_find_nearest_neighbors_index()`, or variants). The alternative (`flann_find_nearest_neighbors()`, etc.) builds the index then throws it away, which can be inefficient if you don't use it correctly.

(a) Compare untouched raw pixels with bounding box raw pixels and with stretched bounding box raw pixels. Which works better? Why? Is there a difference in query times?

(b) Does rescaling each feature (i.e. each pixel value) so that it has unit variance improve either classifier from the previous subexercise?

(c) Plot accuracy against d for a variety of d values for stretched bounding box PCA. You should use some large values of d, reasonably close to 784 ($= 28 \times 28$). Compare this to the accuracy of stretched bounding box raw pixels (equivalent to $d = 784$).

(d) Does rescaling each feature (i.e. each projected direction) so that it has unit variance improve results from the previous subexercise?

3.7. Investigate classifying MNIST using an SVM. Compare the following four cases: untouched raw pixels; stretched bounding box raw pixels; stretched bounding box PCA; and stretched bounding box local PCA. Which works best? Why?

3.8. Investigate classifying MNIST using a decision forest. Using the same parameters for your forest construction (i.e. same depth of tree; same number of trees; etc.), compare the following four cases: untouched raw pixels; stretched bounding box raw pixels; stretched bounding box PCA; and stretched bounding box local PCA. Which works best? Why?

3.9. If you've done all four previous exercises, you're likely tired of MNIST, but very well informed. Compare your methods to the table of methods at http://yann.lecun.com/exdb/mnist/. What improvements could you make?

TYPO - n-dimensional vector of ones should be d

CHAPTER 4

Extracting Important Relationships in High Dimensions

Chapter **??** described methods to explore the relationship between two elements in a dataset. We could extract a pair of elements and construct various plots. For vector data, we could also compute the correlation between different pairs of elements. But if each data item is d-dimensional, there could be a lot of pairs to deal with.

We will think of our dataset as a collection of d dimensional vectors. It turns out that there are easy generalizations of our summaries. However, is hard to plot d-dimensional vectors. We need to find some way to make them fit on a 2-dimensional plot. Some simple methods can offer insights, but to really get what is going on we need methods that can represent all relationships in a dataset in one go.

These methods visualize the dataset as a "blob" in a d-dimensional space. Many such blobs are flattened in some directions, because components of the data are strongly correlated. Finding the directions in which the blobs are flat yields methods to compute lower dimensional representations of the dataset.

4.1 SUMMARIES AND SIMPLE PLOTS

In this chapter, we assume that our data items are vectors. This means that we can add and subtract values and multiply values by a scalar without any distress. This is an important assumption, but it doesn't necessarily mean that data is continuous (for example, you can meaningfully add the number of children in one family to the number of children in another family). It does rule out a lot of discrete data. For example, you can't add "sports" to "grades" and expect a sensible answer.

When we plotted histograms, we saw that mean and variance were a very helpful description of data that had a unimodal histogram. If the histogram had more than one mode, one needed to be somewhat careful to interpret the mean and variance; in the pizza example, we plotted diameters for different manufacturers to try and see the data as a collection of unimodal histograms. In higher dimensions, the analogue of a unimodal histogram is a "blob" — a group of data points that clusters nicely together and should be understood together.

You might not believe that "blob" is a technical term, but it's quite widely used. This is because it is relatively easy to understand a single blob of data. There are good summary representations (mean and covariance, which I describe below). If a dataset forms multiple blobs, we can usually coerce it into a representation as a collection of blobs (using the methods of chapter 6). But many datasets really are single blobs, and we concentrate on such data here. There are quite useful tricks for understanding blobs of low dimension by plotting them, which I describe below.

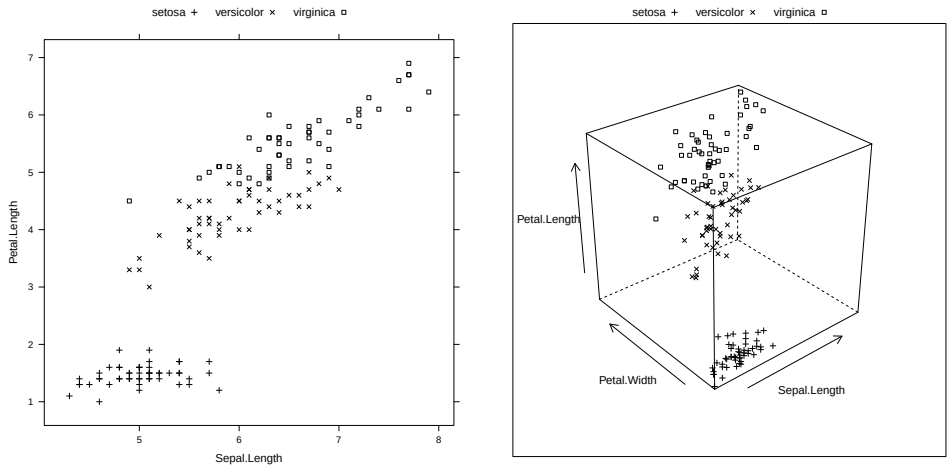


FIGURE 4.1: **Left:** *a 2D scatterplot for the famous Iris data. I have chosen two variables from the four, and have plotted each species with a different marker.* **Right:** *a 3D scatterplot for the same data. You can see from the plots that the species cluster quite tightly, and are different from one another. If you compare the two plots, you can see how suppressing a variable leads to a loss of structure. Notice that, on the left, some 'x's lie on top of boxes; you can see that this is an effect of projection by looking at the 3D picture (for each of these data points, the petal widths are quite different). You should worry that leaving out the last variable might have suppressed something important like this.*

To understand a high dimensional blob, we will need to think about the coordinate transformations that places it into a particularly convenient form.

Notation: Our data items are vectors, and we write a vector as $\mathbf{x}$. The data items are d-dimensional, and there are N of them. The entire data set is $\{\mathbf{x}\}$. When we need to refer to the i'th data item, we write $\mathbf{x}_i$. We write $\{\mathbf{x}_i\}$ for a new dataset made up of N items, where the i'th item is $\mathbf{x}_i$. If we need to refer to the j'th component of a vector $\mathbf{x}_i$, we will write $x_i^{(j)}$ (notice this isn't in bold, because it is a component not a vector, and the j is in parentheses because it isn't a power). Vectors are always column vectors.

4.1.1 The Mean

For one-dimensional data, we wrote

$$\text{mean}\left(\{x\}\right) = \frac{\sum_i x_i}{N}.$$

This expression is meaningful for vectors, too, because we can add vectors and divide by scalars. We write

$$\text{mean}\left(\{\mathbf{x}\}\right) = \frac{\sum_i \mathbf{x}_i}{N}$$

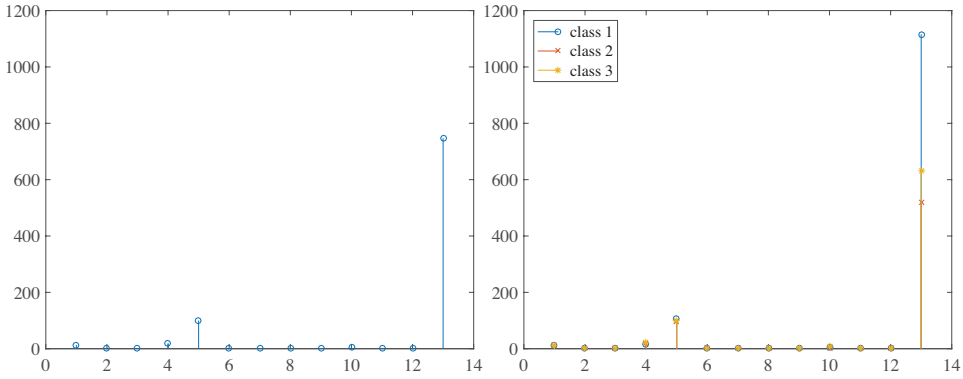


FIGURE 4.2: *On the* **left**, *a stem plot of the mean of all data items in the wine dataset, from* http://archive.ics.uci.edu/ml/datasets/Wine. *On the* **right**, *I have overlaid stem plots of each class mean from the wine dataset, from* http://archive.ics.uci.edu/ml/datasets/Wine, *so that you can see the differences between class means.*

and call this the mean of the data. Notice that each component of $\text{mean}(\{\mathbf{x}\})$ is the mean of that component of the data. There is not an easy analogue of the median, however (how do you order high dimensional data?) and this is a nuisance. Notice that, just as for the one-dimensional mean, we have

$$\text{mean}(\{\mathbf{x} - \text{mean}(\{\mathbf{x}\})\}) = 0$$

(i.e. if you subtract the mean from a data set, the resulting data set has zero mean).

4.1.2 Stem Plots and Scatterplot Matrices

Plotting high dimensional data is tricky. If there are relatively few dimensions, you could just choose two (or three) of them and produce a 2D (or 3D) scatterplot. Figure 4.1 shows such a scatterplot, for data that was originally four dimensional. This dataset is a famous dataset to do with the botanical classification of irises. I found a copy at the UC Irvine repository of datasets that are important in machine learning. You can find the repository at http://archive.ics.uci.edu/ml/index.html.

Another simple but useful plotting mechanism is the stem plot. This is can be a useful way to plot a few high dimensional data points. One plots each component of the vector as a vertical line, typically with a circle on the end (easier seen than said; look at figure 4.2). The dataset I used for this is the wine dataset, from the UC Irvine machine learning data repository. You can find this dataset at http://archive.ics.uci.edu/ml/datasets/Wine. For each of three types of wine, the data records the values of 13 different attributes. In the figure, I show the overall mean of the dataset, and also the mean of each type of wine (also known as the class means, or class conditional means). A natural way to compare class means is to plot them on top of one another in a stem plot (figure 4.2).

Another strategy that is very useful when there aren't too many dimensions is to use a scatterplot matrix. To build one, you lay out scatterplots for each pair

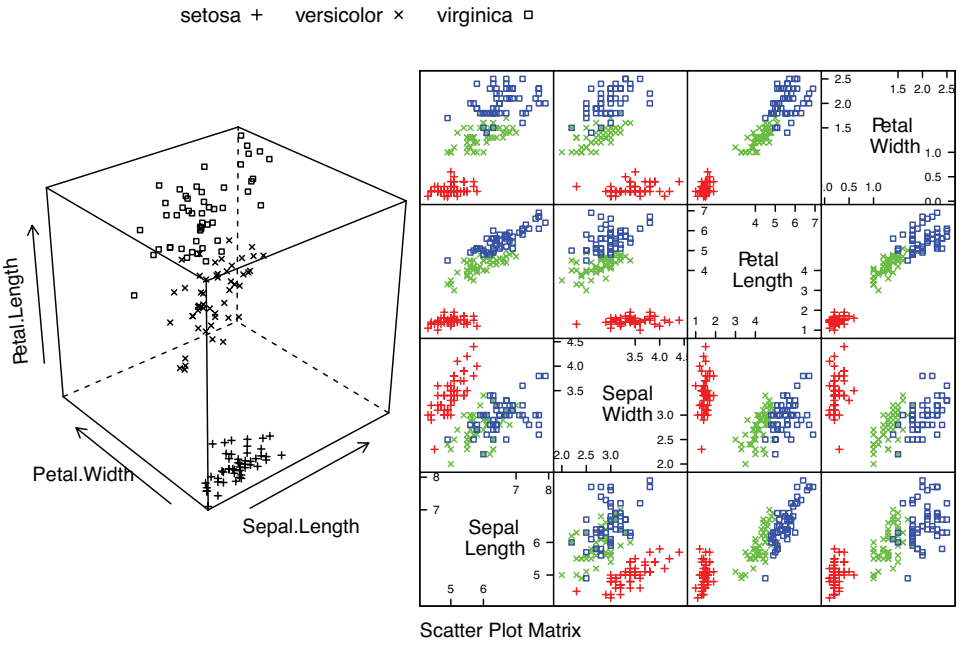


FIGURE 4.3: **Left:** *the 3D scatterplot of the iris data of Figure 4.1, for comparison.* **Right:** *a scatterplot matrix for the Iris data. There are four variables, measured for each of three species of iris. I have plotted each species with a different marker. You can see from the plot that the species cluster quite tightly, and are different from one another.*

of variables in a matrix. On the diagonal, you name the variable that is the vertical axis for each plot in the row, and the horizontal axis in the column. This sounds more complicated than it is; look at the example of figure 4.3, which shows both a 3D scatter plot and a scatterplot matrix for the same dataset. This is the famous Iris dataset, collected by Edgar Anderson in 1936, and made popular amongst statisticians by Ronald Fisher in that year.

Figure 4.4 shows a scatter plot matrix for four of the variables in the height weight dataset of http://www2.stetson.edu/~jrasp/data.htm; look for bodyfat.xls at that URL). This is originally a 16-dimensional dataset, but a 16 by 16 scatterplot matrix is squashed and hard to interpret. For figure 4.4, you can see that weight and adiposity appear to show quite strong correlations, but weight and age are pretty weakly correlated. Height and age seem to have a low correlation. It is also easy to visualize unusual data points. Usually one has an interactive process to do so — you can move a "brush" over the plot to change the color of data points under the brush.

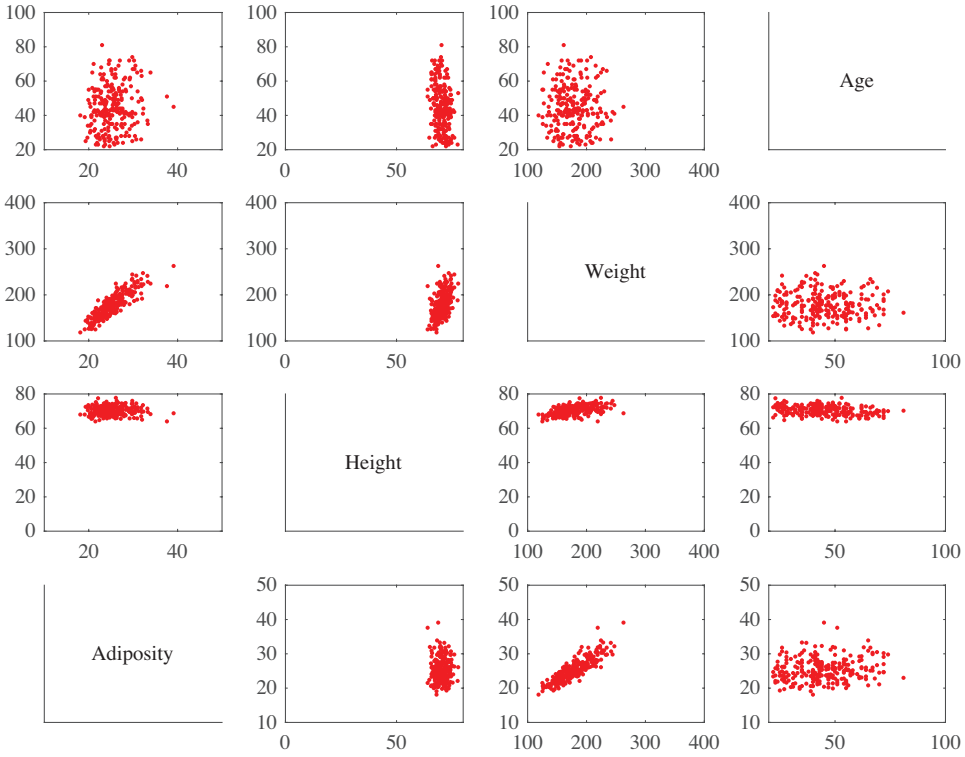


FIGURE 4.4: *This is a scatterplot matrix for four of the variables in the height weight dataset of http://www2.stetson.edu/~jrasp/data.htm. Each plot is a scatterplot of a pair of variables. The name of the variable for the horizontal axis is obtained by running your eye down the column; for the vertical axis, along the row. Although this plot is redundant (half of the plots are just flipped versions of the other half), that redundancy makes it easier to follow points by eye. You can look at a column, move down to a row, move across to a column, etc. Notice how you can spot correlations between variables and outliers (the arrows).*

4.1.3 Covariance

Variance, standard deviation and correlation can each be seen as an instance of a more general operation on data. Recall that I described correlation by extracting two components from each vector of a dataset of vectors. This gave me two datasets of N items; write $\{x\}$ for one and $\{y\}$ for the other. The i'th element of $\{x\}$ corresponds to the i'th element of $\{y\}$. This is because the i'th element of $\{x\}$ is one component of some bigger vector $\mathbf{x}_i$ and the i'th element of $\{y\}$ is another component of this vector. We can define the covariance of $\{x\}$ and $\{y\}$.

Definition: 4.1 *Covariance*

Assume we have two sets of N data items, $\{x\}$ and $\{y\}$. We compute the covariance by

$$\mathsf{cov}\left(\{x\},\{y\}\right) = \frac{\sum_i (x_i - \mathsf{mean}\left(\{x\}\right))(y_i - \mathsf{mean}\left(\{y\}\right))}{N}$$

Covariance measures the tendency of corresponding elements of $\{x\}$ and of $\{y\}$ to be larger than (resp. smaller than) the mean. Just like mean, standard deviation and variance, covariance can refer either to a property of a dataset (as in the definition here) or a particular expectation (as in chapter **??**). The correspondence is defined by the order of elements in the data set, so that x_1 corresponds to y_1, x_2 corresponds to y_2, and so on. If $\{x\}$ tends to be larger (resp. smaller) than its mean for data points where $\{y\}$ is also larger (resp. smaller) than its mean, then the covariance should be positive. If $\{x\}$ tends to be larger (resp. smaller) than its mean for data points where $\{y\}$ is smaller (resp. larger) than its mean, then the covariance should be negative.

From this description, it should be clear we have seen examples of covariance already. Notice that

$$\mathsf{std}\left(x\right)^2 = \mathsf{var}\left(\{x\}\right) = \mathsf{cov}\left(\{x\},\{x\}\right)$$

which you can prove by substituting the expressions. Recall that variance measures the tendency of a dataset to be different from the mean, so the covariance of a dataset with itself is a measure of its tendency not to be constant. More important is the relationship between covariance and correlation, in the box below.

Remember this:

$$\mathsf{corr}\left(\{(x,y)\}\right) = \frac{\mathsf{cov}\left(\{x\},\{y\}\right)}{\sqrt{\mathsf{cov}\left(\{x\},\{x\}\right)}\sqrt{\mathsf{cov}\left(\{y\},\{y\}\right)}}.$$

This is occasionally a useful way to think about correlation. It says that the correlation measures the tendency of $\{x\}$ and $\{y\}$ to be larger (resp. smaller) than their means for the same data points, *compared to* how much they change on their own.

4.1.4 The Covariance Matrix

Working with covariance (rather than correlation) allows us to unify some ideas. In particular, for data items which are d dimensional vectors, it is straightforward to compute a single matrix that captures all covariances between all pairs of components — this is the covariance matrix.

Definition: 4.2 *Covariance Matrix*

The covariance matrix is:

$$\mathsf{Covmat}\left(\{\mathbf{x}\}\right) = \frac{\sum_i (\mathbf{x}_i - \mathsf{mean}\left(\{\mathbf{x}\}\right))(\mathbf{x}_i - \mathsf{mean}\left(\{\mathbf{x}\}\right))^T}{N}$$

Notice that it is quite usual to write a covariance matrix as Σ, and we will follow this convention.

Covariance matrices are often written as Σ, whatever the dataset (you get to figure out precisely which dataset is intended, from context). Generally, when we want to refer to the j, k'th entry of a matrix $\mathcal{A}$, we will write $\mathcal{A}_{jk}$, so Σ_{jk} is the covariance between the j'th and k'th components of the data.

Useful Facts: 4.1 *Properties of the covariance matrix*

- The j, k'th entry of the covariance matrix is the covariance of the j'th and the k'th components of $\mathbf{x}$, which we write $\mathsf{cov}\left(\left\{x^{(j)}\right\}, \left\{x^{(k)}\right\}\right)$.
- The j, j'th entry of the covariance matrix is the variance of the j'th component of $\mathbf{x}$.
- The covariance matrix is symmetric.
- The covariance matrix is always positive semi-definite; it is positive definite, *unless* there is some vector $\mathbf{a}$ such that $\mathbf{a}^T(\mathbf{x}_i - \mathsf{mean}\left(\{\mathbf{x}_i\}\right) = 0$ for all i.

Proposition:

$$\mathsf{Covmat}\left(\{\mathbf{x}\}\right)_{jk} = \mathsf{cov}\left(\left\{x^{(j)}\right\}, \left\{x^{(k)}\right\}\right)$$

Proof: Recall

$$\mathsf{Covmat}\left(\{\mathbf{x}\}\right) = \frac{\sum_i (\mathbf{x}_i - \mathsf{mean}\left(\{\mathbf{x}\}\right))(\mathbf{x}_i - \mathsf{mean}\left(\{\mathbf{x}\}\right))^T}{N}$$

and the j, k'th entry in this matrix will be

$$\frac{\sum_i (x_i^{(j)} - \mathsf{mean}\left(\left\{x^{(j)}\right\}\right))(x_i^{(k)} - \mathsf{mean}\left(\left\{x^{(k)}\right\}\right))^T}{N}$$

which is $\mathsf{cov}\left(\left\{x^{(j)}\right\}, \left\{x^{(k)}\right\}\right)$.

Proposition:

$$\mathsf{Covmat}\left(\{\mathbf{x}_i\}\right)_{jj} = \Sigma_{jj} = \mathsf{var}\left(\left\{x^{(j)}\right\}\right)$$

Proof:

$$\begin{aligned} \mathsf{Covmat}\left(\{\mathbf{x}\}\right)_{jj} &= \mathsf{cov}\left(\left\{x^{(j)}\right\}, \left\{x^{(j)}\right\}\right) \\ &= \mathsf{var}\left(\left\{x^{(j)}\right\}\right) \end{aligned}$$

Proposition:

$$\mathsf{Covmat}\left(\{\mathbf{x}\}\right) = \mathsf{Covmat}\left(\{\mathbf{x}\}\right)^T$$

Proof: We have

$$\begin{aligned}\mathsf{Covmat}\left(\{\mathbf{x}\}\right)_{jk} &= \mathsf{cov}\left(\left\{x^{(j)}\right\}, \left\{x^{(k)}\right\}\right) \\ &= \mathsf{cov}\left(\left\{x^{(k)}\right\}, \left\{x^{(j)}\right\}\right) \\ &= \mathsf{Covmat}\left(\{\mathbf{x}\}\right)_{kj}\end{aligned}$$

Proposition: *Write* $\Sigma = \mathsf{Covmat}\left(\{\mathbf{x}\}\right)$. *If there is no vector* $\mathbf{a}$ *such that* $\mathbf{a}^T(\mathbf{x}_i - \mathsf{mean}\left(\{\mathbf{x}\}\right)) = 0$ *for all* i, *then for any vector* $\mathbf{u}$, *such that* $\|\mathbf{u}\| > 0$,

$$\mathbf{u}^T\Sigma\mathbf{u} > 0.$$

If there is such a vector $\mathbf{a}$, *then*

$$\mathbf{u}^T\Sigma\mathbf{u} \geq 0.$$

Proof: We have

$$\begin{aligned}\mathbf{u}^T\Sigma\mathbf{u} &= \frac{1}{N}\sum_i \left[\mathbf{u}^T(\mathbf{x}_i - \mathsf{mean}\left(\{\mathbf{x}\}\right))\right]\left[(\mathbf{x}_i - \mathsf{mean}\left(\{\mathbf{x}\}\right))^T\mathbf{u}\right] \\ &= \frac{1}{N}\sum_i \left[\mathbf{u}^T(\mathbf{x}_i - \mathsf{mean}\left(\{\mathbf{x}\}\right))\right]^2.\end{aligned}$$

Now this is a sum of squares. If there is some $\mathbf{a}$ such that $\mathbf{a}^T(\mathbf{x}_i - \mathsf{mean}\left(\{\mathbf{x}\}\right)) = 0$ for every i, then the covariance matrix must be positive semidefinite (because the sum of squares could be zero in this case). Otherwise, it is positive definite, because the sum of squares will always be positive.

4.2 USING MEAN AND COVARIANCE TO UNDERSTAND HIGH DIMENSIONAL DATA

The trick to interpreting high dimensional data is to use the mean and covariance to understand the blob. Figure 4.5 shows a two-dimensional data set. Notice that there is obviously some correlation between the x and y coordinates (it's a diagonal

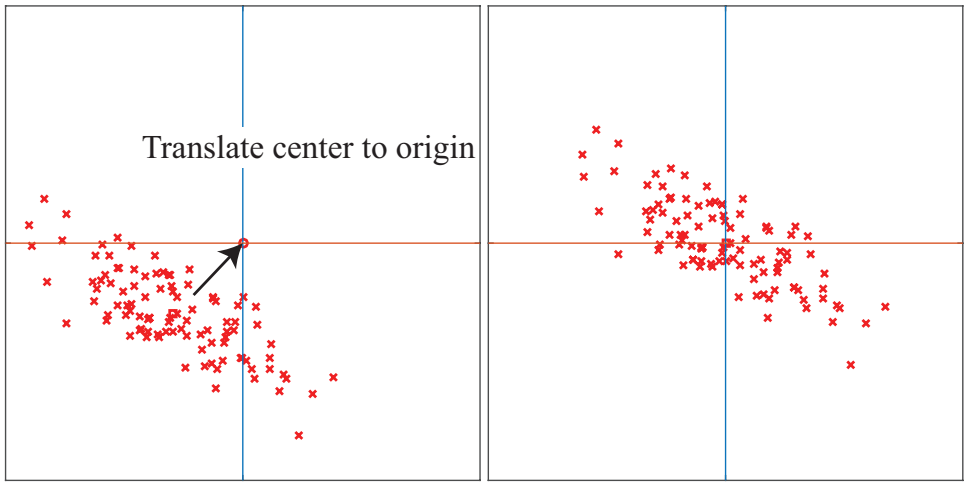


FIGURE 4.5: *On the* **left**, *a "blob" in two dimensions. This is a set of data points that lie somewhat clustered around a single center, given by the mean. I have plotted the mean of these data points with a hollow square (it's easier to see when there is a lot of data). To translate the blob to the origin, we just subtract the mean from each datapoint, yielding the blob on the* **right**.

blob), and that neither x nor y has zero mean. We can easily compute the mean and subtract it from the data points, and this translates the blob so that the origin is at the mean (Figure 4.5). The mean of the new, translated dataset is zero.

Notice this blob is diagonal. We know what that means from our study of correlation – the two measurements are correlated. Now consider *rotating* the blob of data about the origin. This doesn't change the distance between any pair of points, but it does change the overall appearance of the blob of data. We can choose a rotation that means the blob looks (roughly!) like an axis aligned ellipse. *In these coordinates* there is no correlation between the horizontal and vertical components. But one direction has more variance than the other.

It turns out we can extend this approach to high dimensional blobs. We will translate their mean to the origin, then rotate the blob so that there is no correlation between any pair of distinct components (this turns out to be straightforward, which may not be obvious to you). Now the blob looks like an axis-aligned ellipsoid, and we can reason about (a) what axes are "big" and (b) what that means about the original dataset.

4.2.1 Mean and Covariance under Affine Transformations

We have a d dimensional dataset $\{\mathbf{x}\}$. An affine transformation of this data is obtained by choosing some matrix $\mathcal{A}$ and vector $\mathbf{b}$, then forming a new dataset $\{\mathbf{m}\}$, where $\mathbf{m}_i = \mathcal{A}\mathbf{x}_i + \mathbf{b}$. Here $\mathcal{A}$ doesn't have to be square, or symmetric, or anything else; it just has to have second dimension d.

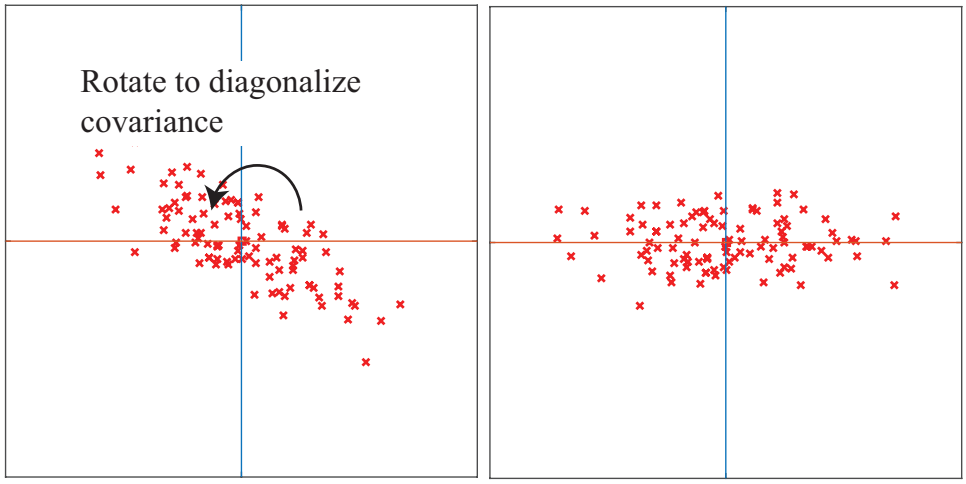


FIGURE 4.6: *On the* **left**, *the translated blob of figure 4.5. This blob lies somewhat diagonally, because the vertical and horizontal components are correlated. On the* **right**, *that blob of data rotated so that there is no correlation between these components. We can now describe the blob by the vertical and horizontal variances alone, as long as we do so in the new coordinate system. In this coordinate system, the vertical variance is significantly larger than the horizontal variance — the blob is short and wide.*

It is easy to compute the mean and covariance of $\{\mathbf{m}\}$. We have

$$\begin{aligned}\text{mean}\left(\{\mathbf{m}\}\right) &= \text{mean}\left(\{\mathcal{A}\mathbf{x}+\mathbf{b}\}\right)\\ &= \mathcal{A}\text{mean}\left(\{\mathbf{x}\}\right)+\mathbf{b},\end{aligned}$$

so you get the new mean by multiplying the original mean by $\mathcal{A}$ and adding $\mathbf{b}$.

The new covariance matrix is easy to compute as well. We have:

$$\begin{aligned}\text{Covmat}\left(\{\mathbf{m}\}\right) &= \text{Covmat}\left(\{\mathcal{A}\mathbf{x}+\mathbf{b}\}\right)\\ &= \frac{\sum_i(\mathbf{m}_i-\text{mean}\left(\{\mathbf{m}\}\right))(\mathbf{m}_i-\text{mean}\left(\{\mathbf{m}\}\right))^T}{N}\\ &= \frac{\sum_i(\mathcal{A}\mathbf{x}_i+\mathbf{b}-\mathcal{A}\text{mean}\left(\{\mathbf{x}\}\right)-\mathbf{b})(\mathcal{A}\mathbf{x}_i+\mathbf{b}-\mathcal{A}\text{mean}\left(\{\mathbf{x}\}\right)-\mathbf{b})^T}{N}\\ &= \frac{\mathcal{A}\left[\sum_i(\mathbf{x}_i-\text{mean}\left(\{\mathbf{x}\}\right))(\mathbf{x}_i-\text{mean}\left(\{\mathbf{x}\}\right))^T\right]\mathcal{A}^T}{N}\\ &= \mathcal{A}\text{Covmat}\left(\{\mathbf{x}\}\right)\mathcal{A}^T.\end{aligned}$$

All this means that we can try and choose affine transformations that yield "good" means and covariance matrices. It is natural to choose $\mathbf{b}$ so that the mean of the new dataset is zero. An appropriate choice of $\mathcal{A}$ can reveal a lot of information about the dataset.

4.2.2 Eigenvectors and Diagonalization

Recall a matrix $\mathcal{M}$ is **symmetric** if $\mathcal{M} = \mathcal{M}^T$. A symmetric matrix is necessarily square. Assume $\mathcal{S}$ is a $d \times d$ symmetric matrix, $\mathbf{u}$ is a $d \times 1$ vector, and λ is a scalar. If we have

$$\mathcal{S}\mathbf{u} = \lambda\mathbf{u}$$

then $\mathbf{u}$ is referred to as an **eigenvector** of $\mathcal{S}$ and λ is the corresponding **eigenvalue**. Matrices don't have to be symmetric to have eigenvectors and eigenvalues, but the symmetric case is the only one of interest to us.

In the case of a symmetric matrix, the eigenvalues are real numbers, and there are d distinct eigenvectors that are normal to one another, and can be scaled to have unit length. They can be stacked into a matrix $\mathcal{U} = [\mathbf{u}_1, \ldots, \mathbf{u}_d]$. This matrix is orthonormal, meaning that $\mathcal{U}^T\mathcal{U} = \mathcal{I}$.

This means that there is a diagonal matrix Λ and an orthonormal matrix $\mathcal{U}$ such that

$$\mathcal{S}\mathcal{U} = \mathcal{U}\Lambda.$$

In fact, there is a large number of such matrices, because we can reorder the eigenvectors in the matrix $\mathcal{U}$, and the equation still holds with a new Λ, obtained by reordering the diagonal elements of the original Λ. There is no reason to keep track of this complexity. Instead, we adopt the convention that the elements of $\mathcal{U}$ are always ordered so that the elements of Λ are sorted along the diagonal, with the largest value coming first. This gives us a particularly important procedure.

Procedure: 4.1 *Diagonalizing a symmetric matrix*

We can convert any symmetric matrix $\mathcal{S}$ to a diagonal form by computing

$$\mathcal{U}^T\mathcal{S}\mathcal{U} = \Lambda.$$

Numerical and statistical programming environments have procedures to compute $\mathcal{U}$ and Λ for you. We assume that the elements of $\mathcal{U}$ are always ordered so that the elements of Λ are sorted along the diagonal, with the largest value coming first.

Useful Facts: 4.2 *Orthonormal matrices are rotations*

You should think of orthonormal matrices as rotations, because they do not change lengths or angles. For $\mathbf{x}$ a vector, $\mathcal{R}$ an orthonormal matrix, and $\mathbf{m} = \mathcal{R}\mathbf{x}$, we have

$$\mathbf{u}^T\mathbf{u} = \mathbf{x}^T\mathcal{R}^T\mathcal{R}\mathbf{x} = \mathbf{x}^T\mathcal{I}\mathbf{x} = \mathbf{x}^T\mathbf{x}.$$

This means that $\mathcal{R}$ doesn't change lengths. For $\mathbf{y}$, $\mathbf{z}$ both unit vectors, we have that the cosine of the angle between them is

$$\mathbf{y}^T\mathbf{x}.$$

By the argument above, the inner product of $\mathcal{R}\mathbf{y}$ and $\mathcal{R}\mathbf{x}$ is the same as $\mathbf{y}^T\mathbf{x}$. This means that $\mathcal{R}$ doesn't change angles, either.

4.2.3 Diagonalizing Covariance by Rotating Blobs

We start with a dataset of N d-dimensional vectors $\{\mathbf{x}\}$. We can translate this dataset to have zero mean, forming a new dataset $\{\mathbf{m}\}$ where $\mathbf{m}_i = \mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\})$. Now recall that, if we were to form a new dataset $\{\mathbf{a}\}$ where

$$\mathbf{a}_i = \mathcal{A}\mathbf{m}_i$$

the covariance matrix of $\{\mathbf{a}\}$ would be

$$\mathsf{Covmat}(\{\mathbf{a}\}) = \mathcal{A}\mathsf{Covmat}(\{\mathbf{m}\})\mathcal{A}^T = \mathcal{A}\mathsf{Covmat}(\{\mathbf{x}\})\mathcal{A}^T.$$

Recall also we can diagonalize $\mathsf{Covmat}(\{\mathbf{m}\}) = \mathsf{Covmat}(\{\mathbf{x}\})$ to get

$$\mathcal{U}^T\mathsf{Covmat}(\{\mathbf{x}\})\mathcal{U} = \Lambda.$$

But this means we could form the dataset $\{\mathbf{r}\}$, using the rule

$$\mathbf{r}_i = \mathcal{U}^T\mathbf{m}_i = \mathcal{U}^T(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\})).$$

The mean of this new dataset is clearly $\mathbf{0}$. The covariance of this dataset is

$$\begin{aligned}\mathsf{Covmat}(\{\mathbf{r}\}) &= \mathsf{Covmat}\left(\{\mathcal{U}^T\mathbf{x}\}\right) \\ &= \mathcal{U}^T\mathsf{Covmat}(\{\mathbf{x}\})\mathcal{U} \\ &= \Lambda,\end{aligned}$$

where Λ is a diagonal matrix of eigenvalues of $\mathsf{Covmat}(\{\mathbf{x}\})$ that we obtained by diagonalization. We now have a very useful fact about $\{\mathbf{r}\}$: its covariance matrix is diagonal. This means that every pair of distinct components has covariance zero, and so has correlation zero. Remember that, in describing diagonalization, we

adopted the convention that the eigenvectors of the matrix being diagonalized were ordered so that the eigenvalues are sorted in descending order along the diagonal of Λ. Our choice of ordering means that the first component of $\mathbf{r}$ has the highest variance, the second component has the second highest variance, and so on.

The transformation from $\{\mathbf{x}\}$ to $\{\mathbf{r}\}$ is a translation followed by a rotation (remember $\mathcal{U}$ is orthonormal, and so a rotation). So this transformation is a high dimensional version of what I showed in Figures 4.5 and 4.6.

Useful Fact: 4.3 *You can transform data to zero mean and diagonal covariance*

We can translate and rotate *any* blob of data into a coordinate system where it has (a) zero mean and (b) diagonal covariance matrix.

4.2.4 Approximating Blobs

The covariance matrix of $\{\mathbf{r}\}$ is diagonal, and the values on the diagonal are interesting. It is quite usual for high dimensional datasets to have a small number of large values on the diagonal, and a lot of small values. This means that the blob of data is really a low dimensional blob in a high dimensional space. For example, think about a line segment (a 1D blob) in 3D. As another example, look at Figure 4.3; the scatterplot matrix strongly suggests that the blob of data is flattened (eg look at the petal width vs petal length plot).

The blob represented by $\{\mathbf{r}\}$ is low dimensional in a very strong sense. We need some notation to see this. The data set $\{\mathbf{r}\}$ is d-dimensional. We will try to represent it with an s dimensional dataset, and see what error we incur. Choose some $s < d$. Now take each data point $\mathbf{r}_i$ and replace the last $d - s$ components with 0. Call the resulting data item $\mathbf{p}_i$. We should like to know the average error in representing $\mathbf{r}_i$ with $\mathbf{p}_i$.

This error is

$$\frac{1}{N}\sum_i \left[(\mathbf{r}_i - \mathbf{p}_i)^T (\mathbf{r}_i - \mathbf{p}_i)^T\right].$$

Write $r_i^{(j)}$ for the j' component of $\mathbf{r}_i$, and so on. Remember that $\mathbf{p}_i$ is zero in the last $d - s$ components. The error is then

$$\frac{1}{N}\sum_i \left[\sum_{j>s}^{j=d} \left(r_i^{(j)}\right)^2\right].$$

But we know this number, because we know that $\{\mathbf{r}\}$ has zero mean. The error is

$$\sum_{j>s}^{j=d} \mathsf{var}\left(\left\{r^{(j)}\right\}\right)$$

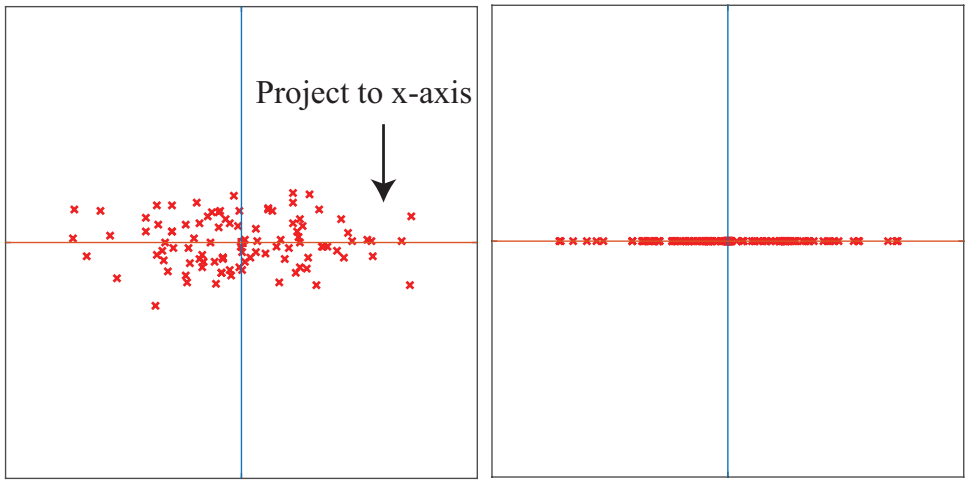


FIGURE 4.7: *On the* **left**, *the translated and rotated blob of figure 4.6. This blob is stretched — one direction has more variance than another. Setting the y coordinate to zero for each of these datapoints results in a representation that has relatively low error, because there isn't much variance in these values. This results in the blob on the* **right**. *The text shows how the error that results from this projection is computed.*

which is the sum of the diagonal elements of the covariance matrix from r, r to d, d. If this sum is small compared to the sum of the first r components, then dropping the last $d - r$ components results in a small error. In that case, we could think about the data as being r dimensional. Figure 4.7 shows the result of using this approach to represent the blob we've used as a running example as a 1D dataset.

This is an observation of great practical importance. As a matter of experimental fact, a great deal of high dimensional data produces relatively low dimensional blobs. We can identify the main directions of variation in these blobs, and use them to understand and to represent the dataset.

4.2.5 Example: Transforming the Height-Weight Blob

Translating a blob of data doesn't change the scatterplot matrix in any interesting way (the axes change, but the picture doesn't). Rotating a blob produces really interesting results, however. Figure 4.8 shows the dataset of figure 4.4, translated to the origin and rotated to diagonalize it. Now we do not have names for each component of the data (they're linear combinations of the original components), but each pair is now not correlated. This blob has some interesting shape features. Figure 4.8 shows the gross shape of the blob best. Each panel of this figure has the same scale in each direction. You can see the blob extends about 80 units in direction 1, but only about 15 units in direction 2, and much less in the other two directions. You should think of this blob as being rather cigar-shaped; it's long in one direction, but there isn't much in the others. The cigar metaphor isn't perfect because there aren't any 4 dimensional cigars, but it's helpful. You can think of

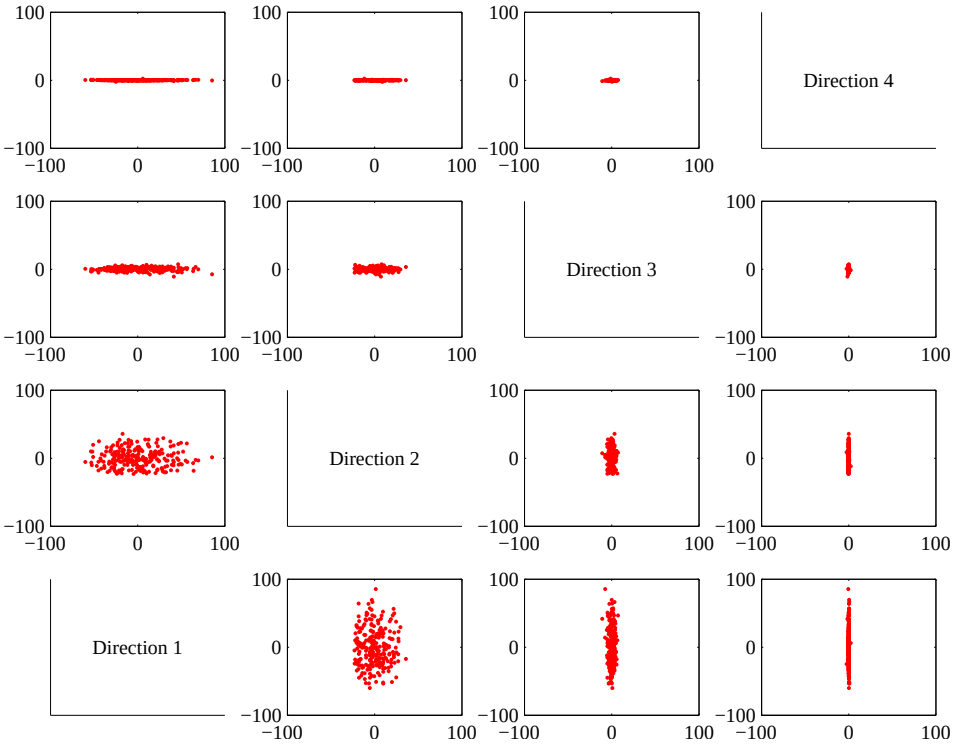


FIGURE 4.8: *A panel plot of the bodyfat dataset of figure 4.4, now rotated so that the covariance between all pairs of distinct dimensions is zero. Now we do not know names for the directions — they're linear combinations of the original variables. Each scatterplot is on the same set of axes, so you can see that the dataset extends more in some directions than in others.*

each panel of this figure as showing views down each of the four axes of the cigar.

Now look at figure 4.9. This shows the same rotation of the same blob of data, but now the scales on the axis have changed to get the best look at the detailed shape of the blob. First, you can see that blob is a little curved (look at the projection onto direction 2 and direction 4). There might be some effect here worth studying. Second, you can see that some points seem to lie away from the main blob. I have plotted each data point with a dot, and the interesting points with a number. These points are clearly special in some way.

The problem with these figures is that the axes are meaningless. The components are weighted combinations of components of the original data, so they don't have any units, etc. This is annoying, and often inconvenient. But I obtained Figure 4.8 by translating, rotating and projecting data. It's straightforward to undo the rotation and the translation – this takes the projected blob (which we know to be a good approximation of the rotated and translated blob) back to where the original blob was. Rotation and translation don't change distances, so the result is a good approximation of the original blob, but now in the original blob's co-

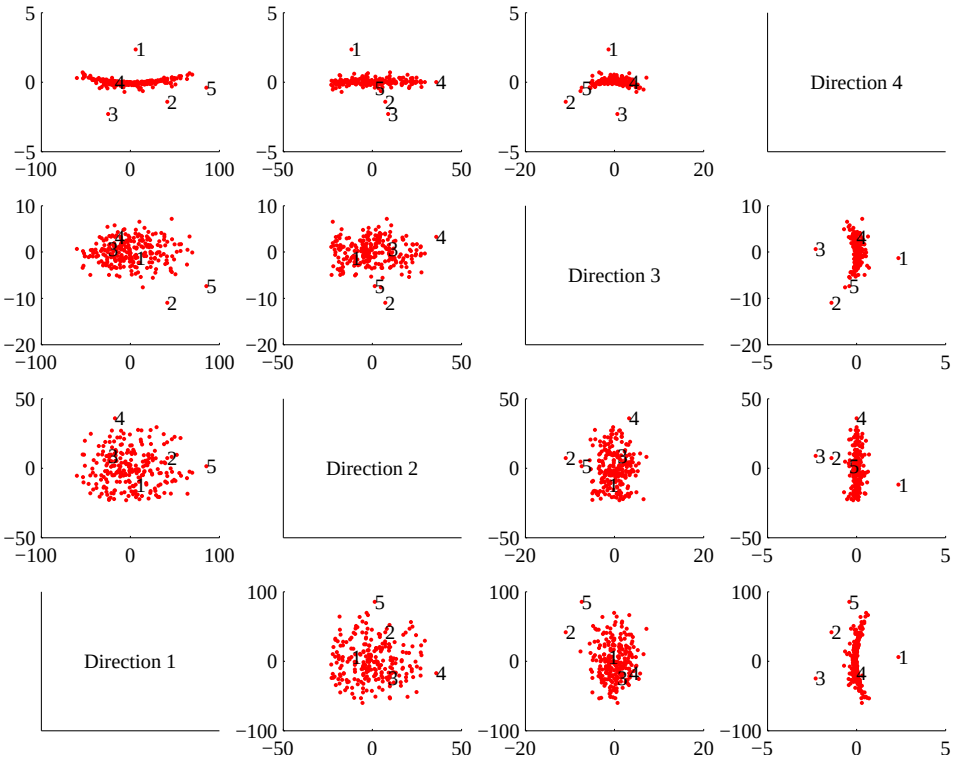


FIGURE 4.9: *A panel plot of the bodyfat dataset of figure 4.4, now rotated so that the covariance between all pairs of distinct dimensions is zero. Now we do not know names for the directions — they're linear combinations of the original variables. I have scaled the axes so you can see details; notice that the blob is a little curved, and there are several data points that seem to lie some way away from the blob, which I have numbered.*

ordinates. Figure 4.10 shows what happens to the data of Figure 4.4. This is a two dimensional version of the original dataset, embedded like a thin pancake of data in a four dimensional space. Crucially, it represents the original dataset quite accurately.

4.3 PRINCIPAL COMPONENTS ANALYSIS

We have seen that a blob of data can be translated so that it has zero mean, then rotated so the covariance matrix is diagonal. In this coordinate system, we can set some components to zero, and get a representation of the data that is still accurate. The rotation and translation can be undone, yielding a dataset that is in the same coordinates as the original, but lower dimensional. The new dataset is a good approximation to the old dataset. All this yields a really powerful idea: we can represent the original dataset with a small number of appropriately chosen vectors.

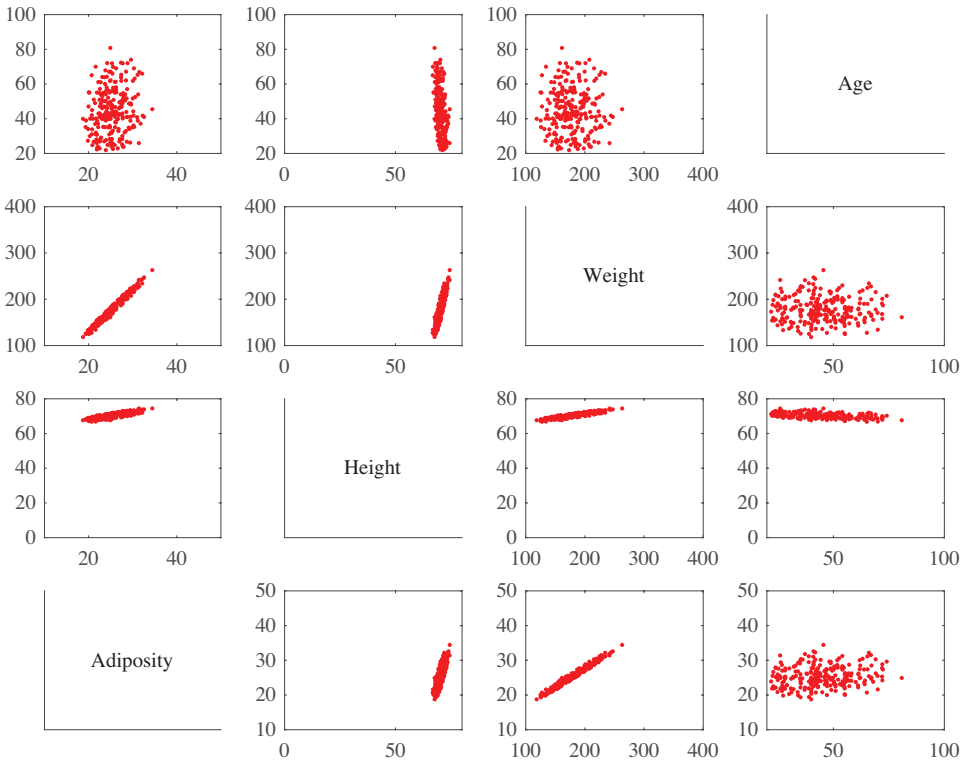


FIGURE 4.10: *The data of Figure 4.4, represented by translating and rotating so that the covariance is diagonal, projecting off the two smallest directions, then undoing the rotation and translation. This blob of data is two dimensional (because we projected off two dimensions), but is represented in a four dimensional space. You can think of it as a thin pancake of data in the four dimensional space (you should compare to Figure 4.4 on page 53). It is a good representation of the original data. Notice that it looks slightly thickened on edge, because it isn't aligned with the coordinate system – think of a view of a plate at a slight slant.*

We start with a dataset of N d-dimensional vectors $\{\mathbf{x}\}$. We translate this dataset to have zero mean, forming a new dataset $\{\mathbf{m}\}$ where $\mathbf{m}_i = \mathbf{x}_i - \mathsf{mean}\,(\{\mathbf{x}\})$. We diagonalize $\mathsf{Covmat}\,(\{\mathbf{m}\}) = \mathsf{Covmat}\,(\{\mathbf{x}\})$ to get

$$\mathcal{U}^T \mathsf{Covmat}\,(\{\mathbf{x}\})\mathcal{U} = \Lambda$$

and form the dataset $\{\mathbf{r}\}$, using the rule

$$\mathbf{r}_i = \mathcal{U}^T \mathbf{m}_i = \mathcal{U}^T(\mathbf{x}_i - \mathsf{mean}\,(\{\mathbf{x}\})).$$

We saw the mean of this dataset is zero, and the covariance is diagonal. We then represented the d-dimensional data set $\{\mathbf{r}\}$ with an s dimensional dataset, by choosing some $r < d$, then taking each data point $\mathbf{r}_i$ and replacing the last $d - s$ components with 0. We call the resulting data item $\mathbf{p}_i$.

Now consider undoing the rotation and translation. We would form a new dataset $\{\hat{\mathbf{x}}\}$, with the i'th element given by

$$\hat{\mathbf{x}}_i = \mathcal{U}\mathbf{p}_i + \mathsf{mean}\left(\{\mathbf{x}\}\right)$$

(you should check this expression). But this expression says that $\hat{\mathbf{x}}_i$ is constructed by forming a weighted sum of the first s columns of $\mathcal{U}$ (because all the other components of $\mathbf{p}_i$ are zero), then adding $\mathsf{mean}\left(\{\mathbf{x}\}\right)$. If we write $\mathbf{u}_j$ for the j'th column of $\mathcal{U}$, we have

$$\hat{\mathbf{x}}_i = \sum_{j=1}^{s} r_i^{(j)}\mathbf{u}_j + \mathsf{mean}\left(\{\mathbf{x}\}\right).$$

What is important about this sum is that s is usually a lot less than d. The $\mathbf{u}_j$ are known as **principal components** of the dataset.

Remember this: *Data items in a d dimensional data set can usually be represented with good accuracy as a weighted sum of a small number s of d dimensional vectors, together with the mean. This means that the dataset lies on an s-dimensional subspace of the d-dimensional space. The subspace is spanned by the principal components of the data.*

We can easily determine the error in approximating $\{\mathbf{x}\}$ with $\{\hat{\mathbf{x}}\}$. The error in representing $\{\mathbf{r}\}$ by $\{\mathbf{p}_s\}$ was easy to compute. We had

$$\frac{1}{N}\sum_i \left[\left(\mathbf{r}_i - \mathbf{p}_i\right)^T \left(\mathbf{r}_i - \mathbf{p}_i\right)^T\right] = \frac{1}{N}\sum_i \left[\sum_{j>s}^{j=d} \left(r_i^{(j)}\right)^2\right].$$

This was the sum of the diagonal elements of the covariance matrix of $\{\mathbf{r}\}$ from s, s to d, d. If this sum is small compared to the sum of the first s components, then dropping the last $d - s$ components results in a small error.

The error in representing $\{\mathbf{x}\}$ with $\{\hat{\mathbf{x}}\}$ is now easy to get. Rotations and translations do not change lengths. This means that

$$\left\|\mathbf{x}_i - \hat{\mathbf{x}}_i\right\|^2 = \left\|\mathbf{r}_i - \mathbf{p}_{r,i}\right\|^2 = \sum_{u=r+1}^{d} (\mathbf{r}_i^{(u)})^2$$

which is the sum of the diagonal elements of the covariance matrix of $\{\mathbf{r}\}$ from s, s to d, d which is easy to evaluate, because these are the values of the $d-s$ eigenvalues that we decided to ignore. Now we could choose s by identifying how much error we can tolerate. More usual is to plot the eigenvalues of the covariance matrix, and look for a "knee", like that in Figure 4.11. You can see that the sum of remaining eigenvalues is small.

Procedure: 4.2 *Principal Components Analysis*

Assume we have a general data set $\mathbf{x}_i$, consisting of N d-dimensional vectors. Now write $\Sigma = \mathsf{Covmat}(\{\mathbf{x}\})$ for the covariance matrix.
Form $\mathcal{U}$, Λ, such that

$$\Sigma\mathcal{U} = \mathcal{U}\Lambda$$

(these are the eigenvectors and eigenvalues of Σ). Ensure that the entries of Λ are sorted in decreasing order. Choose r, the number of dimensions you wish to represent. Typically, we do this by plotting the eigenvalues and looking for a "knee" (Figure 4.11). It is quite usual to do this by hand.
Constructing a low-dimensional representation: For $1 \leq j \leq r$, write $\mathbf{u}_i$ for the i'th column of $\mathcal{U}$. Represent the data point $\mathbf{x}_i$ as

$$\hat{\mathbf{x}_i} = \mathsf{mean}(\{\mathbf{x}\}) + \sum_{j=1}^{s} \left[\mathbf{u}_j^T (\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\}))\right] \mathbf{u}_j$$

4.3.1 Example: Representing Colors with Principal Components

Diffuse surfaces reflect light uniformly in all directions. Examples of diffuse surfaces include matte paint, many styles of cloth, many rough materials (bark, cement, stone, etc.). One way to tell a diffuse surface is that it does not look brighter (or darker) when you look at it along different directions. Diffuse surfaces can be colored, because the surface reflects different fractions of the light falling on it at different wavelengths. This effect can be represented by measuring the spectral reflectance of a surface, which is the fraction of light the surface reflects as a function of wavelength. This is usually measured in the visual range of wavelengths (about 380nm to about 770 nm). Typical measurements are every few nm, depending on the measurement device. I obtained data for 1995 different surfaces from http://www.cs.sfu.ca/~colour/data/ (there are a variety of great datasets here, from Kobus Barnard).

Each spectrum has 101 measurements, which are spaced 4nm apart. This represents surface properties to far greater precision than is really useful. Physical properties of surfaces suggest that the reflectance can't change too fast from wavelength to wavelength. It turns out that very few principal components are sufficient to describe almost any spectral reflectance function. Figure 4.11 shows the mean spectral reflectance of this dataset, and Figure 4.11 shows the eigenvalues of the covariance matrix.

This is tremendously useful in practice. One should think of a spectral reflectance as a function, usually written $\rho(\lambda)$. What the principal components analysis tells us is that we can represent this function rather accurately on a (really small) finite dimensional basis. This basis is shown in figure 4.11. This means that

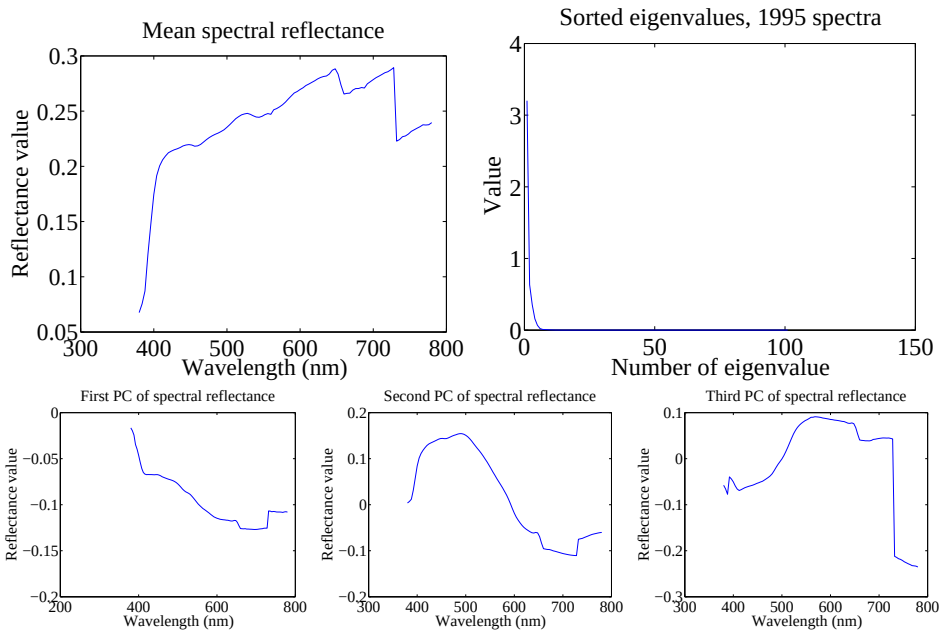


FIGURE 4.11: *On the* **top left**, *the mean spectral reflectance of a dataset of 1995 spectral reflectances, collected by Kobus Barnard (at* http://www.cs.sfu.ca/~colour/data/*). On the* **top right**, *eigenvalues of the covariance matrix of spectral reflectance data, from a dataset of 1995 spectral reflectances, collected by Kobus Barnard (at* http://www.cs.sfu.ca/~colour/data/*). Notice how the first few eigenvalues are large, but most are very small; this suggests that a good representation using few principal components is available. The* **bottom row** *shows the first three principal components. A linear combination of these, with appropriate weights, added to the mean of figure* **??**, *gives a good representation of the dataset.*

there is a mean function $r(\lambda)$ and k functions $\phi_m(\lambda)$ such that, for any $\rho(\lambda)$,

$$\rho(\lambda) = r(\lambda) + \sum_{i=1}^{k} c_i \phi_i(\lambda) + e(\lambda)$$

where $e(\lambda)$ is the error of the representation, which we know is small (because it consists of all the other principal components, which have tiny variance). In the case of spectral reflectances, using a value of k around 3-5 works fine for most applications (Figure 4.12). This is useful, because when we want to predict what a particular object will look like under a particular light, we don't need to use a detailed spectral reflectance model; instead, it's enough to know the c_i for that object. This comes in useful in a variety of rendering applications in computer graphics. It is also the key step in an important computer vision problem, called **color constancy**. In this problem, we see a picture of a world of colored objects under unknown colored lights, and must determine what color the objects are. Modern color constancy systems are quite accurate, even though the problem

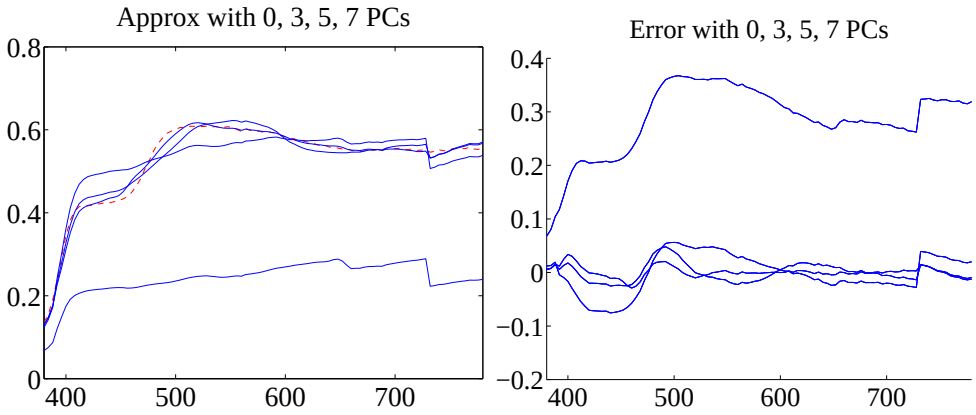


FIGURE 4.12: *On the* **left**, *a spectral reflectance curve (dashed) and approximations using the mean, the mean and 3 principal components, the mean and 5 principal components, and the mean and 7 principal components. Notice the mean is a relatively poor approximation, but as the number of principal components goes up, the error falls rather quickly. On the* **right** *is the error for these approximations. Figure plotted from a dataset of 1995 spectral reflectances, collected by Kobus Barnard (at http://www.cs.sfu.ca/~colour/data/).*

sounds underconstrained. This is because they are able to exploit the fact that relatively few c_i are enough to accurately describe a surface reflectance.

4.3.2 Example: Representing Faces with Principal Components

An image is usually represented as an array of values. We will consider intensity images, so there is a single intensity value in each cell. You can turn the image into a vector by rearranging it, for example stacking the columns onto one another. This means you can take the principal components of a set of images. Doing so was something of a fashionable pastime in computer vision for a while, though there are some reasons that this is not a great representation of pictures. However, the representation yields pictures that can give great intuition into a dataset.

Figure **??** shows the mean of a set of face images encoding facial expressions of Japanese women (available at http://www.kasrl.org/jaffe.html; there are tons of face datasets at http://www.face-rec.org/databases/). I reduced the images to 64x64, which gives a 4096 dimensional vector. The eigenvalues of the covariance of this dataset are shown in figure 4.13; there are 4096 of them, so it's hard to see a trend, but the zoomed figure suggests that the first couple of hundred contain most of the variance. Once we have constructed the principal components, they can be rearranged into images; these images are shown in figure 4.14. Principal components give quite good approximations to real images (figure 4.15).

The principal components sketch out the main kinds of variation in facial expression. Notice how the mean face in Figure 4.14 looks like a relaxed face, but with fuzzy boundaries. This is because the faces can't be precisely aligned, because each face has a slightly different shape. The way to interpret the components is to

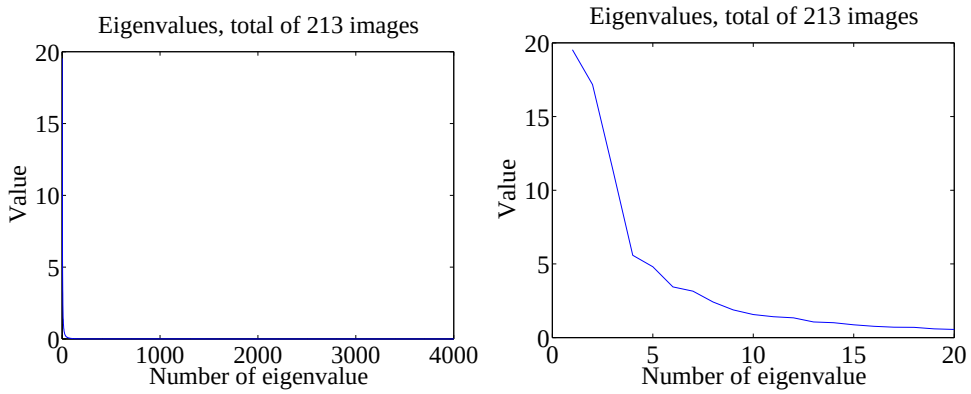


FIGURE 4.13: *On the* **left**,*the eigenvalues of the covariance of the Japanese facial expression dataset; there are 4096, so it's hard to see the curve (which is packed to the left). On the* **right**, *a zoomed version of the curve, showing how quickly the values of the eigenvalues get small.*

remember one adjusts the mean towards a data point by adding (or subtracting) some scale times the component. So the first few principal components have to do with the shape of the haircut; by the fourth, we are dealing with taller/shorter faces; then several components have to do with the height of the eyebrows, the shape of the chin, and the position of the mouth; and so on. These are all images of women who are not wearing spectacles. In face pictures taken from a wider set of models, moustaches, beards and spectacles all typically appear in the first couple of dozen principal components.

4.4 MULTI-DIMENSIONAL SCALING

One way to get insight into a dataset is to plot it. But choosing what to plot for a high dimensional dataset could be difficult. Assume we must plot the dataset in two dimensions (by far the most common choice). We wish to build a scatter plot in two dimensions — but where should we plot each data point? One natural requirement is that the points be laid out in two dimensions in a way that reflects how they sit in many dimensions. In particular, we would like points that are far apart in the high dimensional space to be far apart in the plot, and points that are close in the high dimensional space to be close in the plot.

4.4.1 Choosing Low D Points using High D Distances

We will plot the high dimensional point $\mathbf{x}_i$ at $\mathbf{v}_i$, which is a two-dimensional vector. Now the squared distance between points i and j in the high dimensional space is

$$D_{ij}^{(2)}(\mathbf{x}) = (\mathbf{x}_i - \mathbf{x}_j)^T(\mathbf{x}_i - \mathbf{x}_j)$$

(where the superscript is to remind you that this is a squared distance). We could build an $N \times N$ matrix of squared distances, which we write $\mathcal{D}^{(2)}(\mathbf{x})$. The i, j'th

FIGURE 4.14: *The mean and first 16 principal components of the Japanese facial expression dataset.*

entry in this matrix is $D_{ij}^{(2)}(\mathbf{x})$, and the $\mathbf{x}$ argument means that the distances are between points in the high-dimensional space. Now we could choose the $\mathbf{v}_i$ to make

$$\sum_{ij} \left(D_{ij}^{(2)}(\mathbf{x}) - D_{ij}^{(2)}(\mathbf{v}) \right)^2$$

as small as possible. Doing so should mean that points that are far apart in the high dimensional space are far apart in the plot, and that points that are close in the high dimensional space are close in the plot.

In its current form, the expression is difficult to deal with, but we can refine it. Because translation does not change the distances between points, it cannot change either of the $\mathcal{D}^{(2)}$ matrices. So it is enough to solve the case when the mean of the points $\mathbf{x}_i$ is zero. We can assume that

$$\frac{1}{N} \sum_i \mathbf{x}_i = \mathbf{0}.$$

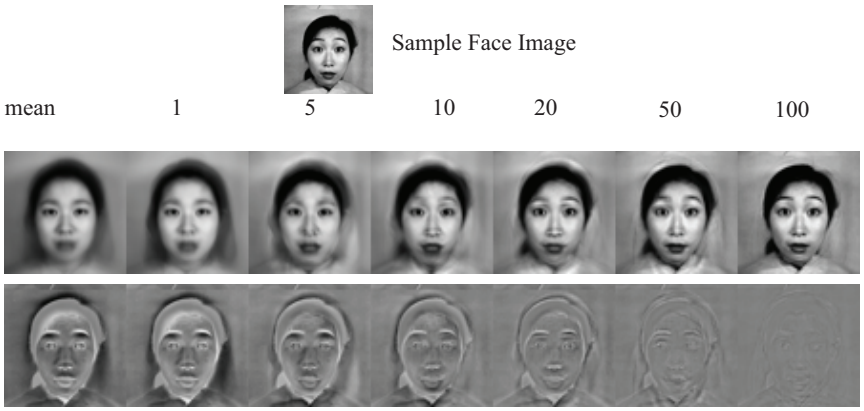


FIGURE 4.15: *Approximating a face image by the mean and some principal components; notice how good the approximation becomes with relatively few components.*

Now write $\mathbf{1}$ for the n-dimensional vector containing all ones, and $\mathcal{I}$ for the identity matrix. Notice that

$$D_{ij}^{(2)} = (\mathbf{x}_i - \mathbf{x}_j)^T(\mathbf{x}_i - \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_i - 2\mathbf{x}_i \cdot \mathbf{x}_j + \mathbf{x}_j \cdot \mathbf{x}_j.$$

Now write

$$\mathcal{A} = \left[\mathcal{I} - \frac{1}{N}\mathbf{1}\mathbf{1}^T\right].$$

Using this expression, you can show that the matrix $\mathcal{M}$, defined below,

$$\mathcal{M}(\mathbf{x}) = -\frac{1}{2}\mathcal{A}\mathcal{D}^{(2)}(\mathbf{x})\mathcal{A}^T$$

has i, jth entry $\mathbf{x}_i \cdot \mathbf{x}_j$ (exercises). I now argue that, to make $\mathcal{D}^{(2)}(\mathbf{v})$ is close to $\mathcal{D}^{(2)}(\mathbf{x})$, it is enough to make $\mathcal{M}(\mathbf{v})$ close to $\mathcal{M}(\mathbf{x})$. Proving this will take us out of our way unnecessarily, so I omit a proof.

We need some notation. Take the dataset of N d-dimensional column vectors $\mathbf{x}_i$, and form a matrix $\mathcal{X}$ by stacking the vectors, so

$$\mathcal{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \cdots \\ \mathbf{x}_N^T \end{bmatrix}.$$

In this notation, we have

$$\mathcal{M}(\mathbf{x}) = \mathcal{X}\mathcal{X}^T.$$

Notice $\mathcal{M}(\mathbf{x})$ is symmetric, and it is positive semidefinite. It can't be positive definite, because the data is zero mean, so $\mathcal{M}(\mathbf{x})\mathbf{1} = 0$.

We can choose a set of $\mathbf{v}_i$ that makes $\mathcal{D}^{(2)}(\mathbf{v})$ close to $\mathcal{D}^{(2)}(\mathbf{x})$ quite easily.

To obtain a $\mathcal{M}(\mathbf{v})$ that is close to $\mathcal{M}(\mathbf{x})$, we need to choose $\mathcal{V} = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_N]^T$ so that $\mathcal{V}\mathcal{V}^T$ is close to $\mathcal{M}(\mathbf{x})$. We are computing an approximate factorization of the matrix $\mathcal{M}(\mathbf{x})$.

4.4.2 Factoring a Dot-Product Matrix

We seek a set of k dimensional $\mathbf{v}$ that can be stacked into a matrix $\mathcal{V}$. This must produce a $\mathcal{M}(\mathbf{v}) = \mathcal{V}\mathcal{V}^T$ that must (a) be as close as possible to $\mathcal{M}(\mathbf{x})$ and (b) have rank at most k. It can't have rank larger than k because there must be some $\mathcal{V}$ which is $N \times k$ so that $\mathcal{M}(\mathbf{v}) = \mathcal{V}\mathcal{V}^T$. The rows of this $\mathcal{V}$ are our $\mathbf{v}_i^T$.

We can obtain the best factorization of $\mathcal{M}(\mathbf{x})$ from a diagonalization. Write write $\mathcal{U}$ for the matrix of eigenvectors of $\mathcal{M}(\mathbf{x})$ and Λ for the diagonal matrix of eigenvalues sorted in descending order, so we have

$$\mathcal{M}(\mathbf{x}) = \mathcal{U}\Lambda\mathcal{U}^T$$

and write $\Lambda^{(1/2)}$ for the matrix of positive square roots of the eigenvalues. Now we have

$$\mathcal{M}(\mathbf{x}) = \mathcal{U}\Lambda^{1/2}\Lambda^{1/2}\mathcal{U}^T = \left(\mathcal{U}\Lambda^{1/2}\right)\left(\mathcal{U}\Lambda^{1/2}\right)^T$$

which allows us to write

$$\mathcal{X} = \mathcal{U}\Lambda^{1/2}.$$

Now think about approximating $\mathcal{M}(\mathbf{x})$ by the matrix $\mathcal{M}(\mathbf{v})$. The error is a sum of squares of the entries,

$$\text{err}(\mathcal{M}(\mathbf{x}), \mathcal{A}) = \sum_{ij}(m_{ij} - a_{ij})^2.$$

Because $\mathcal{U}$ is a rotation, it is straightforward to show that

$$\text{err}(\mathcal{U}^T\mathcal{M}(\mathbf{x})\mathcal{U}, \mathcal{U}^T\mathcal{M}(\mathbf{v})\mathcal{U}) = \text{err}(\mathcal{M}(\mathbf{x}), \mathcal{M}(\mathbf{v})).$$

But

$$\mathcal{U}^T\mathcal{M}(\mathbf{x})\mathcal{U} = \Lambda$$

which means that we could find $\mathcal{M}(\mathbf{v})$ from the best rank k approximation to Λ. This is obtained by setting all but the k largest entries of Λ to zero. Call the resulting matrix Λ_k. Then we have

$$\mathcal{M}(\mathbf{v}) = \mathcal{U}\Lambda_k\mathcal{U}$$

and

$$\mathcal{V} = \mathcal{U}\Lambda_k^{(1/2)}.$$

The first k columns of $\mathcal{V}$ are non-zero. We drop the remaining $N - k$ columns of zeros. The rows of the resulting matrix are our $\mathbf{v}_i$, and we can plot these. This method for constructing a plot is known as **principal coordinate analysis**.

This plot might not be perfect, because reducing the dimension of the data points should cause some distortions. In many cases, the distortions are tolerable. In other cases, we might need to use a more sophisticated scoring system that penalizes some kinds of distortion more strongly than others. There are many ways to do this; the general problem is known as **multidimensional scaling**.

Procedure: 4.3 *Principal Coordinate Analysis*

Assume we have a matrix $D^{(2)}$ consisting of the squared differences between each pair of N points. We do not need to know the points. We wish to compute a set of points in r dimensions, such that the distances between these points are as similar as possible to the distances in $D^{(2)}$.

- Form $\mathcal{A} = \left[\mathcal{I} - \frac{1}{N}\mathbf{1}\mathbf{1}^T\right]$.
- Form $\mathcal{W} = \frac{1}{2}\mathcal{A}\mathcal{D}^{(2)}\mathcal{A}^T$.
- Form $\mathcal{U}$, Λ, such that $\mathcal{W}\mathcal{U} = \mathcal{U}\Lambda$ (these are the eigenvectors and eigenvalues of $\mathcal{W}$). Ensure that the entries of Λ are sorted in decreasing order.
- Choose r, the number of dimensions you wish to represent. Form Λ_r, the top left $r \times r$ block of Λ. Form $\Lambda_r^{(1/2)}$, whose entries are the positive square roots of Λ_r. Form $\mathcal{U}_r$, the matrix consisting of the first r columns of $\mathcal{U}$.

Then

$$\mathcal{V}^T = \Lambda_r^{(1/2)}\mathcal{U}_r^T = [\mathbf{v}_1, \ldots, \mathbf{v}_N]$$

is the set of points to plot.

4.4.3 Example: Mapping with Multidimensional Scaling

Multidimensional scaling gets positions (the $\mathcal{V}$ of section 4.4.1) from distances (the $\mathcal{D}^{(2)}(\mathbf{x})$ of section 4.4.1). This means we can use the method to build maps from distances alone. I collected distance information from the web (I used http://www.distancefromto.net, but a google search on "city distances" yields a wide range of possible sources), then applied multidimensional scaling. I obtained distances between the South African provincial capitals, in kilometers. I then used principal coordinate analysis to find positions for each capital, and rotated, translated and scaled the resulting plot to check it against a real map (Figure 4.16).

One natural use of principal coordinate analysis is to see if one can spot any structure in a dataset. Does the dataset form a blob, or is it clumpy? This isn't a perfect test, but it's a good way to look and see if anything interesting is happening. In figure 4.17, I show a 3D plot of the spectral data, reduced to three dimensions using principal coordinate analysis. The plot is quite interesting. You should notice that the data points are spread out in 3D, but actually seem to lie on a complicated curved surface — they very clearly don't form a uniform blob. To me, the structure looks somewhat like a butterfly. I don't know why this occurs (perhaps the universe is doodling), but it certainly suggests that something worth investigating is going on. Perhaps the choice of samples that were measured is funny; perhaps the

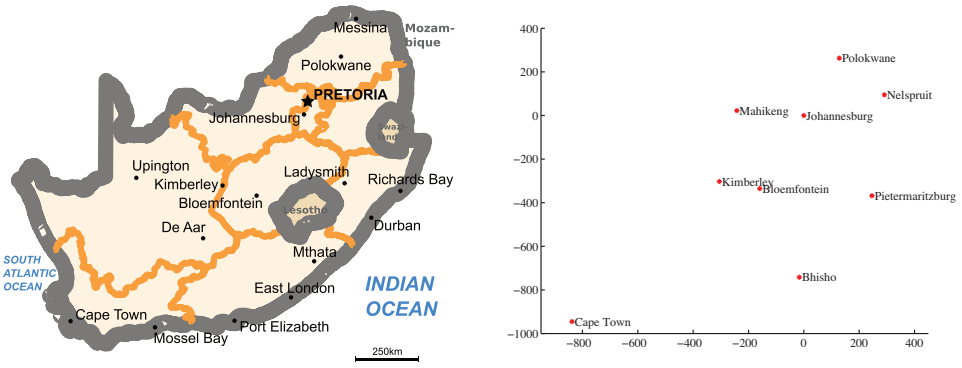


FIGURE 4.16: *On the* **left**, *a public domain map of South Africa, obtained from* http://commons.wikimedia.org/wiki/File:Map_of_South_Africa.svg, *and edited to remove surrounding countries. On the* **right**, *the locations of the cities inferred by multidimensional scaling, rotated, translated and scaled to allow a comparison to the map by eye. The map doesn't have all the provincial capitals on it, but it's easy to see that MDS has placed the ones that are there in the right places (use a piece of ruled tracing paper to check).*

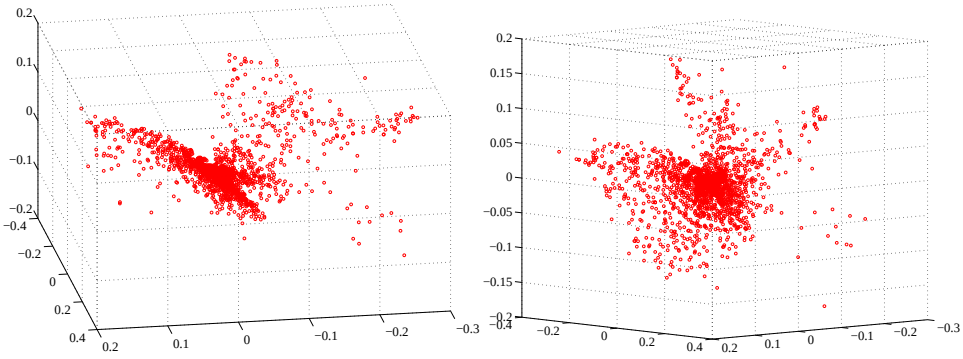


FIGURE 4.17: *Two views of the spectral data of section 4.3.1, plotted as a scatter plot by applying principal coordinate analysis to obtain a 3D set of points. Notice that the data spreads out in 3D, but seems to lie on some structure; it certainly isn't a single blob. This suggests that further investigation would be fruitful.*

measuring instrument doesn't make certain kinds of measurement; or perhaps there are physical processes that prevent the data from spreading out over the space.

Our algorithm has one really interesting property. In some cases, we do not actually know the datapoints as vectors. Instead, we *just* know distances between the datapoints. This happens often in the social sciences, but there are important cases in computer science as well. As a rather contrived example, one could survey people about breakfast foods (say, eggs, bacon, cereal, oatmeal, pancakes, toast, muffins, kippers and sausages for a total of 9 items). We ask each person to rate the similarity of each pair of distinct items on some scale. We advise people that similar

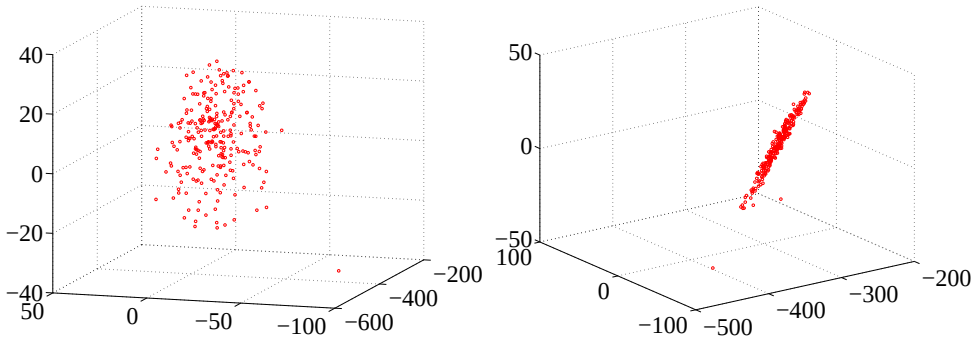


FIGURE 4.18: *Two views of a multidimensional scaling to three dimensions of the height-weight dataset. Notice how the data seems to lie in a flat structure in 3D, with one outlying data point. This means that the distances between data points can be (largely) explained by a 2D representation.*

items are ones where, if they were offered both, they would have no particular preference; but, for dissimilar items, they would have a strong preference for one over the other. The scale might be "very similar", "quite similar", "similar", "quite dissimilar", and "very dissimilar" (scales like this are often called **Likert scales**). We collect these similarities from many people for each pair of distinct items, and then average the similarity over all respondents. We compute distances from the similarities in a way that makes very similar items close and very dissimilar items distant. Now we have a table of distances between items, and can compute a $\mathcal{V}$ and produce a scatter plot. This plot is quite revealing, because items that most people think are easily substituted appear close together, and items that are hard to substitute are far apart. The neat trick here is that we did not start with a $\mathcal{X}$, but with just a set of distances; but we were able to associate a vector with "eggs", and produce a meaningful plot.

4.5 EXAMPLE: UNDERSTANDING HEIGHT AND WEIGHT

Recall the height-weight data set of section **??** (from http://www2.stetson.edu/~jrasp/data.htm; look for bodyfat.xls at that URL). This is, in fact, a 16-dimensional dataset. The entries are (in this order): *bodyfat; density; age; weight; height; adiposity; neck; chest; abdomen; hip; thigh; knee; ankle; biceps; forearm; wrist.* We know already that many of these entries are correlated, but it's hard to grasp a 16 dimensional dataset in one go. The first step is to investigate with a multidimensional scaling.

Figure **??** shows a multidimensional scaling of this dataset down to three dimensions. The dataset seems to lie on a (fairly) flat structure in 3D, meaning that inter-point distances are relatively well explained by a 2D representation. Two points seem to be special, and lie far away from the flat structure. The structure isn't perfectly flat, so there will be small errors in a 2D representation; but it's clear that a lot of dimensions are redundant. Figure 4.19 shows a 2D representation of these points. They form a blob that is stretched along one axis, and there is no sign

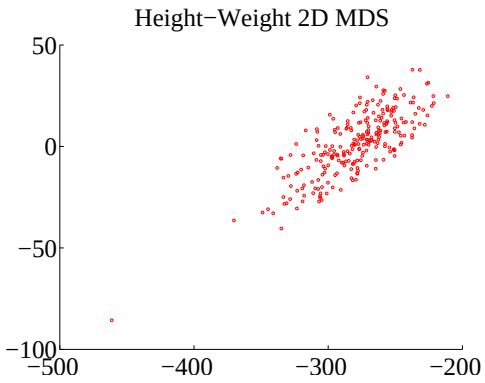


FIGURE 4.19: *A multidimensional scaling to two dimensions of the height-weight dataset. One data point is clearly special, and another looks pretty special. The data seems to form a blob, with one axis quite a lot more important than another.*

of multiple blobs. There's still at least one special point, which we shall ignore but might be worth investigating further. The distortions involved in squashing this dataset down to 2D seem to have made the second special point less obvious than it was in figure **??**.

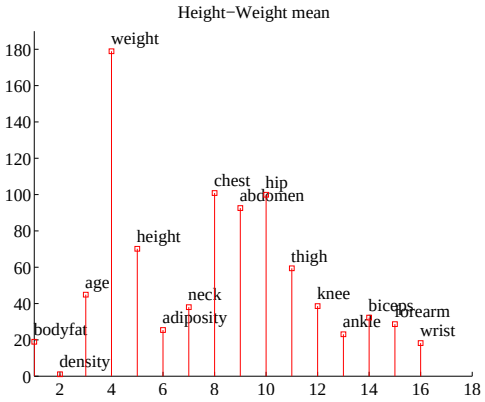


FIGURE 4.20: *The mean of the bodyfat.xls dataset. Each component is likely in a different unit (though I don't know the units), making it difficult to plot the data without being misleading. I've adopted one solution here, by plotting a stem plot. You shouldn't try to compare the values to one another. Instead, think of this plot as a compact version of a table.*

The next step is to try a principal component analysis. Figure 4.20 shows the mean of the dataset. The components of the dataset have different units, and shouldn't really be compared. But it is difficult to interpret a table of 16 numbers, so I have plotted the mean as a stem plot. Figure 4.21 shows the eigenvalues of the covariance for this dataset. Notice how one dimension is very important, and

after the third principal component, the contributions become small. Of course, I could have said "fourth", or "fifth", or whatever — the precise choice depends on how small a number you think is "small".

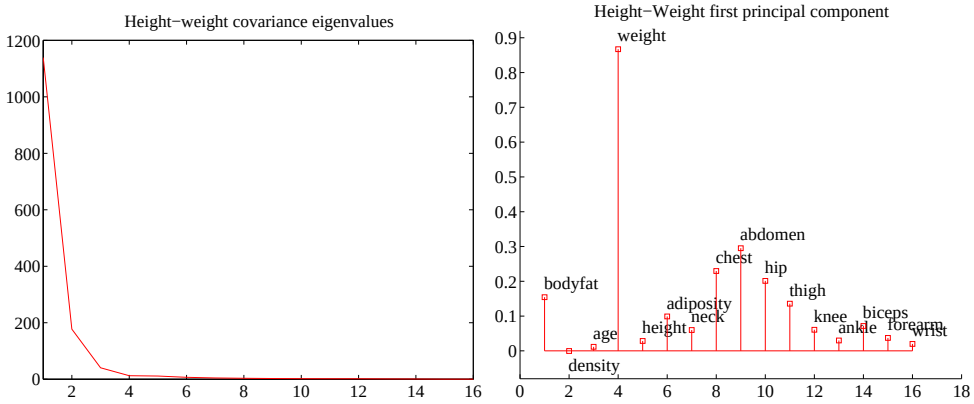


FIGURE 4.21: *On the* **left**, *the eigenvalues of the covariance matrix for the bodyfat data set. Notice how fast the eigenvalues fall off; this means that most principal components have very small variance, so that data can be represented well with a small number of principal components. On the* **right**, *the first principal component for this dataset, plotted using the same convention as for figure 4.20.*

Figure 4.21 also shows the first principal component. The eigenvalues justify thinking of each data item as (roughly) the mean plus some weight times this principal component. From this plot you can see that data items with a larger value of *weight* will also have larger values of most other measurements, except *age* and *density*. You can also see how much larger; if the weight goes up by 8.5 units, then the abdomen will go up by 3 units, and so on. This explains the main variation in the dataset.

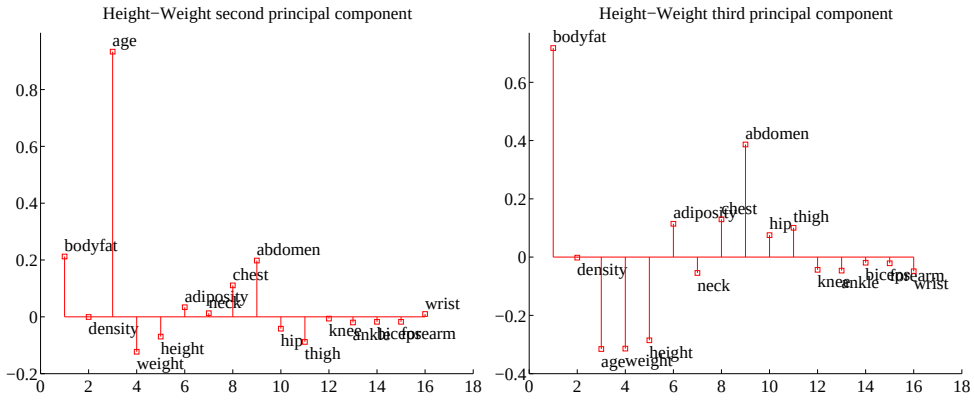


FIGURE 4.22: *On the* **left**, *the second principal component, and on the* **right** *the third principal component of the height-weight dataset.*

In the rotated coordinate system, the components are not correlated, and they have different variances (which are the eigenvalues of the covariance matrix). You can get some sense of the data by adding these variances; in this case, we get 1404. This means that, in the translated and rotated coordinate system, the average data point is about $37 = \sqrt{1404}$ units away from the center (the origin). Translations and rotations do not change distances, so the average data point is about 37 units from the center in the original dataset, too. If we represent a datapoint by using the mean and the first three principal components, there will be some error. We can estimate the average error from the component variances. In this case, the sum of the first three eigenvalues is 1357, so the mean square error in representing a datapoint by the first three principal components is $\sqrt{(1404 - 1357)}$, or 6.8. The relative error is $6.8/37 = 0.18$. Another way to represent this information, which is more widely used, is to say that the first three principal components explain all but $(1404 - 1357)/1404 = 0.034$, or 3.4% of the variance; notice that this is the square of the relative error, which will be a much smaller number.

All this means that explaining a data point as the mean and the first three principal components produces relatively small errors. Figure 4.23 shows the second and third principal component of the data. These two principal components suggest some further conclusions. As *age* gets larger, *height* and *weight* get slightly smaller, but the weight is redistributed; *abdomen* gets larger, whereas *thigh* gets smaller. A smaller effect (the third principal component) links *bodyfat* and *abdomen*. As *bodyfat* goes up, so does *abdomen*.

4.6 YOU SHOULD

4.6.1 remember these definitions:

4.6.2 remember these terms:

4.6.3 remember these facts:

4.6.4 remember these procedures:

4.6.5 be able to:

- Create, plot and interpret the first few principal components of a dataset.
- Compute the error resulting from ignoring some principal components.

PROBLEMS

Summaries

4.1. You have a dataset $\{\mathbf{x}\}$ of N vectors, $\mathbf{x}_i$, each of which is d-dimensional. We will consider a linear function of this dataset. Write $\mathbf{a}$ for a constant vector; then the value of this linear function evaluated on the i'th data item is $\mathbf{a}^T\mathbf{x}_i$. Write $f_i = \mathbf{a}^T\mathbf{x}_i$. We can make a new dataset $\{f\}$ out of the values of this linear function.

(a) Show that $\mathsf{mean}(\{f\}) = \mathbf{a}^T\mathsf{mean}(\{\mathbf{x}\})$ (easy).

(b) Show that $\mathsf{var}(\{f\}) = \mathbf{a}^T\mathsf{Covmat}(\{\mathbf{x}\})\mathbf{a}$ (harder, but just push it through the definition).

(c) Assume the dataset has the special property that there exists some $\mathbf{a}$ so that $\mathbf{a}^T\mathsf{Covmat}(\{\mathbf{x}\})\mathbf{a}$. Show that this means that the dataset lies on a hyperplane.

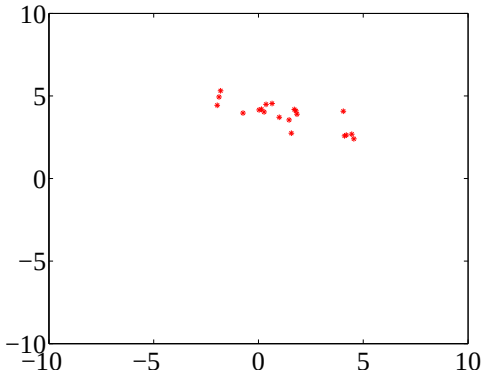


FIGURE 4.23: *Figure for the question*

4.2. On Figure 4.23, mark the mean of the dataset, the first principal component, and the second principal component.

4.3. You have a dataset $\{\mathbf{x}\}$ of N vectors, $\mathbf{x}_i$, each of which is d-dimensional. Assume that $\mathsf{Covmat}(\{\mathbf{x}\})$ has one non-zero eigenvalue. Assume that $\mathbf{x}_1$ and $\mathbf{x}_2$ do not have the same value.

(a) Show that you can choose a set of t_i so that you can represent *every* data item $\mathbf{x}_i$ *exactly*

$$\mathbf{x}_i = \mathbf{x}_1 + t_i(\mathbf{x}_2 - \mathbf{x}_1).$$

(b) Now consider the dataset of these t values. What is the relationship between (a) $\mathsf{std}(t)$ and (b) the non-zero eigenvalue of $\mathsf{Covmat}(\{\mathbf{x}\})$? Why?

PROGRAMMING EXERCISES

4.4. Obtain the iris dataset from the UC Irvine machine learning data repository at http://https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data.

(a) Plot a scatterplot matrix of this dataset, showing each species with a different marker.

(b) Now obtain the first two principal components of the data. Plot the data on those two principal components alone, again showing each species with a different marker. Has this plot introduced significant distortions? Explain

4.5. Take the wine dataset from the UC Irvine machine learning data repository at https://archive.ics.uci.edu/ml/datasets/Wine.

(a) Plot the eigenvalues of the covariance matrix in sorted order. How many principal components should be used to represent this dataset? Why?

(b) Construct a stem plot of each of the first 3 principal components (i.e. the eigenvectors of the covariance matrix with largest eigenvalues). What do you see?

(c) Compute the first two principal components of this dataset, and project it onto those components. Now produce a scatter plot of this two dimensional dataset, where data items of class 1 are plotted as a '1', class 2 as a '2', and so on.

4.6. Take the wheat kernel dataset from the UC Irvine machine learning data repository at http://archive.ics.uci.edu/ml/datasets/seeds. Compute the first two principal components of this dataset, and project it onto those components.

(a) Produce a scatterplot of this projection. Do you see any interesting phenomena?

(b) Plot the eigenvalues of the covariance matrix in sorted order. How many principal components should be used to represent this dataset? why?

4.7. The UC Irvine machine learning data repository hosts a collection of data on breast cancer diagnostics, donated by Olvi Mangasarian, Nick Street, and William H. Wolberg. You can find this data at http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic). For each record, there is an id number, 10 continuous variables, and a class (benign or malignant). There are 569 examples. Separate this dataset randomly into 100 validation, 100 test, and 369 training examples. Plot this dataset on the first three principal components, using different markers for benign and malignant cases. What do you see?

4.8. The UC Irvine Machine Learning data archive hosts a dataset of measurements of abalone at http://archive.ics.uci.edu/ml/datasets/Abalone. Compute the principal components of all variables except Sex. Now produce a scatter plot of the measurements projected onto the first two principal components, plotting an "m" for male abalone, an "f" for female abalone and an "i" for infants. What do you see?

4.9. Choose a state. For the 15 largest cities in your chosen state, find the distance between cities and the road mileage between cities. These differ because of the routes that roads take; you can find these distances by careful use of the internet. Prepare a map showing these cities on the plane using principal coordinate analysis for each of these two distances. How badly does using the road network distort to make a map distort the state? Does this differ from state to state? Why?

4.10. CIFAR-10 is a dataset of 32x32 images in 10 categories, collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. It is often used to evaluate machine learning algorithms. You can download this dataset from https://www.cs.toronto.edu/~kriz/cifar.html.

(a) For each category, compute the mean image and the first 20 principal components. Plot the error resulting from representing the images of each category using the first 20 principal components against the category.

(b) Compute the distances between mean images for each pair of classes. Use principal coordinate analysis to make a 2D map of the means of each categories. For this exercise, compute distances by thinking of the images as vectors.

(c) ***** EASE INTO THIS **** Here is another measure of the similarity of two classes. For class A and class B, define $E(A \to B)$ to be the average error obtained by representing all the images of class A using the mean of class A and the first 20 principal components of class B. Now define the similarity between classes to be $(1/2)(E(A \to B) + E(B \to A))$. Use principal coordinate analysis to make a 2D map of the classes. Compare this map to the map in the previous exercise – are they different? why?

CHAPTER 5

Applications and variants of PCA

5.1 PRINCIPAL COMPONENTS WITH THE SVD

If you remember the curse of dimension, you should have noticed something of a problem in my account of PCA. When I described the curse, I said one consequence was that forming a covariance matrix for high dimensional data is hard or impossible. Then I described PCA as a method to understand the important dimensions in high dimensional datasets. But PCA appears to rely on covariance, so I should not be able to form the principal components in the first place. In fact, we can form principal components without computing a covariance matrix.

5.1.1 Principal Components by SVD

I will now assume the dataset has zero mean, to simplify notation. This is easily achieved. You subtract the mean from each data item at the start, and add the mean back once you've finished smoothing. As usual, we have N data items, each a d dimensional column vector. We will now arrange these into a matrix,

$$\mathcal{X} = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \dots \mathbf{x}_N^T \end{pmatrix}$$

where each *row* of the matrix is a data vector. Now notice that the covariance matrix for this dataset can be formed by constructing $\mathcal{X}^T\mathcal{X}$, so that

$$\text{Covmat}\left(\{X\}\right) = \mathcal{X}^T\mathcal{X}$$

and if we form the SVD (see the math notes at the end if you don't remember this) of $\mathcal{X}$, we have $\mathcal{X} = \mathcal{U}\Sigma\mathcal{V}^T$. But we have $\mathcal{X}^T\mathcal{X} = \mathcal{V}\Sigma^T\Sigma\mathcal{V}^T$ so that

$$\mathcal{V}^T\mathcal{X}^T\mathcal{X}\mathcal{V} = \Sigma^T\Sigma$$

and $\Sigma^T\Sigma$ is diagonal. So we can recover the principal components of the dataset without actually forming the covariance matrix - we just form the SVD of $\mathcal{X}$, and the columns of $\mathcal{V}$ are the principal components.

5.1.2 Just a few Principal Components with NIPALS

For really big datasets, even taking the SVD is hard. Usually, we don't really want to recover all the principal components, because we want to recover a reasonably accurate low dimensional representation of the data. We continue to work with a data matrix $\mathcal{X}$, whose *rows* are data items. Now assume we wish to recover the first principal component. This means we are seeking a vector $\mathbf{u}$ and a set of N numbers w_i such that $w_i\mathbf{u}$ is a good approximation to $\mathbf{x}_i$. Now we can stack the w_i into a

column vector $\mathbf{w}$. We are asking that the matrix $\mathbf{w}\mathbf{u}^T$ be a good approximation to $\mathcal{X}$, in the sense that $\mathbf{w}\mathbf{u}^T$ encodes as much of the variance of $\mathcal{X}$ as possible.

The **Frobenius norm** is a term for the matrix norm obtained by summing squared entries of the matrix. We write

$$\|\mathcal{A}\|_F = \sum_{i,j} a_{ij}^2.$$

In the exercises, you will show that the right choice of $\mathbf{w}$ and $\mathbf{u}$ minimizes the cost

$$\|\mathcal{X} - \mathbf{w}\mathbf{u}^T\|_F$$

which we can write as

$$C(\mathbf{w}, \mathbf{u}) = \sum_{ij} \left(x_{ij} - w_i u_j\right)^2.$$

Now we need to *find* the relevant $\mathbf{w}$ and $\mathbf{u}$. Notice there is not a unique choice, because the pair $(s\mathbf{w}, (1/s)\mathbf{u})$ works as well as the pair $(\mathbf{w}, \mathbf{u})$. We will choose $\mathbf{u}$ such that $\|\mathbf{u}\| = 1$. There is still not a unique choice, because you can flip the signs in $\mathbf{u}$ and $\mathbf{w}$, but this doesn't matter. The gradient of the cost function is a set of partial derivatives with respect to components of $\mathbf{w}$ and $\mathbf{u}$. The partial with respect to w_k is

$$\frac{\partial C}{\partial w_k} = \sum_j \left(x_{kj} - w_k u_j\right) u_j$$

which can be written in matrix vector form as

$$\nabla_{\mathbf{w}} C = (\mathcal{X} - \mathbf{w}\mathbf{u}^T)\mathbf{u}.$$

Similarly, the partial with respect to u_l is

$$\frac{\partial C}{\partial u_l} = \sum_i \left(x_{il} - w_i u_l\right) w_i$$

which can be written in matrix vector form as

$$\nabla_{\mathbf{u}} C = (\mathcal{X}^T - \mathbf{u}\mathbf{w}^T)\mathbf{w}.$$

At the solution, these partial derivatives are zero. This suggests an algorithm. First, assume we have an estimate of $\mathbf{u}$, say $\mathbf{u}^{(n)}$. Then we could choose the $\mathbf{w}$ that makes the partial wrt $\mathbf{w}$ zero, so

$$\mathbf{w}^* = \frac{\mathcal{X}\mathbf{u}^{(n)}}{(\mathbf{u}^{(n)})^T\mathbf{u}^{(n)}}.$$

Now we can update the estimate of $\mathbf{u}$ by choosing a value that makes the partial wrt $\mathbf{u}$ zero, using our estimate $\mathbf{w}^*$, to get

$$\mathbf{u}^* = \frac{\mathcal{X}^T\mathbf{w}^*}{(\mathbf{w}^*)^T\mathbf{w}^*}.$$

We need to rescale to ensure that our estimate of $\mathbf{u}$ has unit length. Write $s = \sqrt{(\mathbf{u}^*)^T\mathbf{u}^*}$ We get

$$\mathbf{u}^{(n+1)} = \frac{\mathbf{u}^*}{s}$$

and

$$\mathbf{w}^{(n+1)} = s\mathbf{w}^*.$$

This iteration can be started by choosing some row of $\mathcal{X}$ as $\mathbf{u}^{(0)}$. You can test for convergence by checking $\|\mathbf{u}^{(n+1)} - \mathbf{u}^{(n)}\|$. If this is small enough, then the algorithm has converged.

To obtain a second principal component, you form $\mathcal{X}^{(1)} = \mathcal{X} - \mathbf{w}\mathbf{u}^T$ and apply the algorithm to that. You can get many principal components like this, but it's not a good way to get all of them (eventually numerical issues mean the estimates are poor). The algorithm is widely known as NIPALS (for Non-linear Iterative Partial Least Squares).

5.1.3 Principal Components and Missing Values

Now imagine our dataset has missing values. We assume that the values are not missing in inconvenient patterns — if, for example, the k'th component was missing for every vector then we'd have to drop it — but don't go into what precise kind of pattern is a problem. Your intuition should suggest that we can estimate a few principal components of the dataset without particular problems. The argument is as follows. Each entry of a covariance matrix is a form of average; estimating averages in the presence of missing values is straightforward; and, when we estimate a few principal components, we are estimating far fewer numbers than when we are estimating a whole covariance matrix, so we should be able to make something work. This argument is sound (if vague).

The whole point of NIPALS is that, if you want a few principal components, you don't need to use either a covariance matrix or an SVD. This simplifies thinking about missing values. NIPALS is quite forgiving of missing values, though missing values make it hard to use matrix notation. Recall I wrote the cost function as $C(\mathbf{w}, \mathbf{u}) = \sum_{ij}(x_{ij} - w_i u_j)^2$. Notice that missing data occurs in $\mathcal{X}$ because there are x_{ij} whose values we don't know, but there is no missing data in $\mathbf{w}$ or $\mathbf{u}$ (we're estimating the values, and we always have *some* estimate). We change the sum so that it ranges over only the known values, to get

$$C(\mathbf{w}, \mathbf{u}) = \sum_{ij \in \text{known values}} (x_{ij} - w_i u_j)^2$$

then write

$$\frac{\partial C}{\partial w_k} = \sum_{j \in \text{known values for given } k} \left[(x_{kj} - w_k u_j)\, u_j\right]$$

and

$$\frac{\partial C}{\partial u_l} = \sum_{i \in \text{known values for given } l} \left[(x_{il} - w_i u_l)\, w_i\right].$$

These partial derivatives must be zero at the solution, so we can estimate

$$w_k^* = \frac{\sum_{j \in \text{known values for given } k} [x_{kj} u_j]}{\sum_j \left[u_j^{(n)} u_j^{(n)} \right]}$$

and

$$u_l^* = \frac{\sum_{i \in \text{known values for given } l} x_{il} w_l}{\sum_i w_i^* w_i^*}$$

We then normalize as before.

Procedure: 5.1 *Obtaining some principal components with NIPALS*

We assume that $\mathcal{X}$ has zero mean. Each row is a data item. Start with $\mathbf{u}^0$ as some row of $\mathcal{X}$. Now iterate

- compute

$$w_k^* = \frac{\sum_{j \in \text{known values for given } k} [x_{kj} u_j]}{\sum_j \left[u_j^{(n)} u_j^{(n)} \right]}$$

  and

$$u_l^* = \frac{\sum_{i \in \text{known values for given } l} x_{il} w_l}{\sum_i w_i^* w_i^*};$$

- compute $s = \sqrt{(\mathbf{u}^*)^T \mathbf{u}^*}$, and

$$\mathbf{u}^{(n+1)} = \frac{\mathbf{u}^*}{s}$$

  and

$$\mathbf{w}^{(n+1)} = s\mathbf{w}^*;$$

- Check for convergence by checking that $\|\mathbf{u}^{(n+1)} - \mathbf{u}^{(n)}\|$ is small.

This procedure yields a single principal component representing the highest variance in the dataset. To obtain the next principal component, replace $\mathcal{X}$ with $\mathcal{X} - \mathbf{w}\mathbf{u}^T$ and repeat the procedure. This process will yield good estimates of the first few principal components, but as you generate more principal components, numerical errors will become more significant.

5.2 TEXT MODELS AND LATENT SEMANTIC ANALYSIS

It is really useful to be able to cluster together documents that are "similar". We cannot do so with k-means as we currently understand it, because we do not have

a distance between documents. But the k-means recipe (i.e. iterate: allocate each data point to the closest cluster center; re-estimate cluster centers from their data points) is spectacularly flexible and powerful. I will demonstrate an application of this recipe to text clustering here.

For many kinds of document, we obtain a good representation by (a) choosing a vocabulary (a list of different words) then (b) representing the document by a vector of word counts, where we simply ignore every word outside the vocabulary. This is a viable representation for many applications because quite often, most of the words people actually use come from a relatively short list (typically 100s to 1000s, depending on the particular application). The vector has one component for each word in the list, and that component contains the number of times that particular word is used. This model is sometimes known as a **bag-of-words** model.

We could try to cluster on the distance between word vectors, but this turns out to be a poor idea. This is because quite small changes in word use might lead to large differences between count vectors. For example, some authors might write "car" when others write "auto". In turn, two documents might have a large (resp. small) count for "car" and a small (resp. large) count for "auto". Just looking at the counts would significantly overstate the difference between the vectors. However, the counts are informative: a document that uses the word "car" often, and the word "lipstick" seldom, is likely quite different from a document that uses "lipstick" often and "car" seldom.

Details of how you put the vocabulary together can be quite important. It is not a good idea to count extremely common words, sometimes known as **stop words**, because every document has lots of them and the counts don't tell you very much. Typical stop words include "and", "the", "he", "she", and so on. These are left out of the vocabulary. Notice that the choice of stop words can be quite important, and depends somewhat on the application. It's often, but not always, helpful to stem words – a process that takes "winning" to "win", "hugely" to "huge", and so on. This can create confusion (for example, a search for "stock" may be looking for quite different things than a search for "stocking"). We will always use datasets that have been preprocessed to produce word counts, but you should be aware that pre-processing this data is hard and involves choices that can have significant effects on the application.

5.2.1 Document Clustering with a Simple Topic Model

We get a useful notion of the differences between documents by pretending that the count vector for each document comes from one of a small set of underlying topics. Each topic generates words as independent, identically distributed samples from a multinomial distribution, with one probability per word in the vocabulary. Each topic will be a cluster center. If two documents come from the same topic, they should have "similar" word distributions. Topics are one way to deal with changes in word use. For example, one topic might have quite high probability of generating the word "car" *and* a high probability of generating the word "auto"; another might have low probability of generating those words, but a high probability of generating "lipstick".

Now think about the key elements of the k-means algorithm. First, we need to

be able to tell the distance between any data item and any cluster center. Second, we need to be able to come up with a cluster center for any collection of data items. We can do each of these with our topic model, by thinking about it as a probabilistic model of how a document is generated. We will have t topics. Each document will be a vector of word counts. We will have N vectors of word counts (i.e. documents), and write $\mathbf{x}_i$ for the i'th such vector. To generate a document, we first choose a topic, choosing the j'th topic with probability π_j. Then we will obtain a set of words for the document by repeatedly drawing IID samples from that topic, and recording the count of each word in a count vector.

Each topic is a multinomial probability distribution. Write $\mathbf{p}_j$ for the vector of word probabilities for the j'th topic. We assume that words are generated independently, conditioned on the topic. Write x_{ik} for the k'th component of $\mathbf{x}_i$, and so on. Notice that $\mathbf{x}_i^T\mathbf{1}$ is the sum of entries in $\mathbf{x}_i$, and so the number of words in document i.

Distance from document to topic: The probability of observing the counts in $\mathbf{x}_i$ when the document was generated by topic j is

$$p(\mathbf{x}_i|\mathbf{p}_j) = \left(\frac{(\mathbf{x}_i^T\mathbf{1})!}{\prod_v x_{iv}!}\right)\prod_u p_{ju}^{x_{iu}}.$$

This number will be non-negative, but less than one. It will be big when the document is close to the topic, and small otherwise. We can obtain a distance from this expression by taking the negative log of the probability. This will be small when the document is close to the topic, and big otherwise.

Turning a set of documents into topic probabilities: Now assume we have a set of documents that we assert belong to topic j. We must estimate $\mathbf{p}_j$. We can do this by counting, because we assumed that each word was generated independent of all others, given the topic. One difficulty we will run into is that some (likely, most) counts will be zero, because most words are rare. But the fact you don't see a word in all the documents in the topic doesn't mean it never occurs. We've seen this issue before (section **??**). You should not allow any element of $\mathbf{p}_j$ to be zero; one way to deal with this problem is to add some small number (likely less than one) to each count.

Initialization: One possible initialization is to take a subset of documents, allocate elements randomly to clusters, then compute initial cluster centers.

We now have a k-means algorithm adapted to clustering documents. This isn't the state of the art, by any manner of means. The next step would be to compute soft weights linking a document to every cluster, then re-estimating the cluster centers taking the soft weights into account. This involves some fairly alarming equations (sections 10.2.2, 10.3.3 and 10.3.4).

5.2.2 Latent Semantic Analysis

Assume we have a set of N documents we wish to deal with. We have removed stop words, chosen a d dimensional vocabulary, and counted the number of times each word appears in each document. The result is a collection of N d dimensional vectors. Write the i'th vector $\mathbf{x}_i$ (these are usually called **word vectors**). There is one minor irritation here; I have used d for the dimension of the vector $\mathbf{x}_i$ for

consistency with the rest of the text, but d is the number of terms in the vocabulary *not* the number of documents.

The distance between two word vectors is usually a poor guide to the similarity of two documents. One reason is that the number of words in a document isn't particularly informative. As an extreme example, we could append a document to itself to produce a new document. The new document would have twice as many copies of each word as the old one, yet its meaning wouldn't have changed. But the distance between its word vector and other word vectors would be very different. Another reason is that two words that appear different may mean the same thing ("car" and "auto", above).

There are two tricks to making things better. First, normalizing the vector of word counts is really helpful. Experience has shown that a very effective measure of the similarity of documents i and j is the **cosine distance**

$$d_{ij} = \frac{\mathbf{x}_i^T \mathbf{x}_j}{\|\mathbf{x}_i\| \|\mathbf{x}_j\|}.$$

While this is widely referred to as a distance, it isn't really. If two documents are very similar, their cosine distance will be close to 1; if they are really different, their cosine distance will be close to -1. But notice that

$$\| \frac{\mathbf{x}_i}{\|\mathbf{x}_i\|} - \frac{\mathbf{x}_j}{\|\mathbf{x}_j\|} \|^2 = 2 - 2d_{ij}^2$$

so looking at the distance between normalized word vectors is enough.

Second, experience shows that when we project word vectors to a lower dimensional space, the distances between projected (and then normalized) word vectors are a better measure of the similarity of documents. In turn, this will allow us to cluster documents. There is another, equally useful, way to use the reduced dimensional space. Two apparently different words that have a similar meaning will tend to appear with the same words in a document. So, for example, either "car" or "auto" are likely to have "spanner", "grease", "oil", "gasoline", and so on in the same document. As I shall show, this will allow us to come up with a measure of the similarity of the meaning of two words.

Now arrange word vectors into a matrix in the usual way, to obtain

$$\mathcal{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \dots \\ \mathbf{x}_N^T \end{bmatrix}.$$

This matrix is widely called a **document-term matrix** (its transpose is called a **term-document matrix**). This is because you can think of it as a table of counts; each row represents a document, each column represents a term from the vocabulary. We will use this object to produce a reduced dimension representation of the words in each document.

To form a low dimensional representation, we take an SVD of $\mathcal{X}$, yielding

$$\mathcal{X} = \mathcal{U}\Sigma\mathcal{V}^T.$$

Now write u_{ik} for the i, k'th element of $\mathcal{U}$, σ_k for the k'th diagonal element of Σ, where the elements are ordered by size (largest first) and $\mathbf{v}_k^T$ for the k'th row of $\mathcal{V}$. You should read this equation in terms of documents. The equation says the i'th document is represented as

$$\mathbf{x}_i^T = \sum_{k=1}^{d} u_{ik}\sigma_k \mathbf{v}_k^T.$$

Now $\mathcal{U}$ and $\mathcal{V}$ are orthonormal, so $\sum_{k=1}^{N} u_{ik}^2 = 1$ and $\mathbf{v}_k^T\mathbf{v}_k = 1$. You should look closely at the limits of the sums. There are no more than d non-zero values of σ_k, which is why I wrote the expression for $\mathbf{x}_i$ with d terms. But $\mathcal{U}$ is $N \times N$. Now choose r, and approximate $\mathbf{x}_i^T$ by

$$\mathbf{x}_i^{(r)T} = \sum_{k=1}^{r} u_{ik}\sigma_k \mathbf{v}_k^T.$$

(you can think about this as setting all σ_k for $k > r$ to zero). It is straightforward to show that

$$\left\|\mathbf{x}_i^{(r)T} - \mathbf{x}_i^{(r)T}\right\|^2 \leq \sum_{k=r+1}^{d} \sigma_k^2$$

so if these singular values are small, so is the error in the approximation. Now stack these $\mathbf{x}_i^{(r)}$ into a matrix $\mathcal{X}^{(r)}$, and write Σ_r for the matrix obtained by setting all but the r largest singular values in Σ to 0. We have

$$\mathcal{X}^{(r)} = \mathcal{U}\Sigma_r\mathcal{V}^T$$

and so it is straightforward to recover each $\mathbf{x}_i^{(r)}$ by an SVD on $\mathcal{X}$. You should think of $\mathcal{X}^{(r)}$ as a smoothing of $\mathcal{X}$. For example, each document that contains the word "car" should also have a non-zero count for the word "automobile" (and vice versa) because the two words mean about the same thing. The original matrix of word counts $\mathcal{X}$ doesn't have this information, because it relies on counting actual words. But because word vectors in $\mathcal{X}^{(r)}$ are compelled to occupy a low dimensional space, counts "leak" between words with similar meanings. In turn, the counts in $\mathcal{X}^{(r)}$ are better estimates of what true word counts should be than one can obtain by simply countinng words. Recovering information from the SVD of $\mathcal{X}$ is referred to as **latent semantic analysis**. There are several ways to proceed.

Inter document distances: We could represent the i'th document by

$$\mathbf{d}_i = \frac{\mathbf{x}_i^{(r)}}{\left\|\mathbf{x}_i^{(r)}\right\|}.$$

The distance between $\mathbf{d}_i$ and $\mathbf{d}_j$ is a good representation of the differences in meaning of document i and document j. We can use this to cluster documents using k-means on this representation.

Querying for documents: Assume you have a few query words, and you need to find documents that are suggested by those words. You can represent the query words as a word vector $\mathbf{q}$, which you can think of as a very small document.

We will find nearby documents by: computing a $\mathbf{d}$ for the query word vector, then finding nearby $\mathbf{d}_i$. Computing a $\mathbf{d}$ for the query word vector is straightforward. We find the best representation of $\mathbf{q}$ on the space spanned by $\{\mathbf{v}_1, \ldots, \mathbf{v}_r\}$, then scale that to have unit norm.

Inter-word distances: Each row of $\mathcal{X}^{(r)}$ is a smoothed count of the number of times each word appears in a single document. In contrast, each column counts the number of times a single word appears in each document. Imagine we wish to know the similarity in meaning between two words. Represent the i'th word by the i'th *column* of $\mathcal{X}^{(r)}$, which I shall write as $\mathbf{w}_i$, so that

$$\mathcal{X}^{(r)} = [\mathbf{w}_1, \ldots, \mathbf{w}_d] .$$

Using a word more often (or less often) should likely not change its meaning. In turn, this means we should represent the i'th word by

$$\mathbf{n}_i = \frac{\mathbf{w}_i}{\|\mathbf{w}_i\|}$$

and the distance between the i'th and j'th words is the distance between $\mathbf{n}_i$ and $\mathbf{n}_j$.

Word embeddings: Representing a word as a set of characters can be extremely inconvenient, because the representation does not expose anything about the meaning of the word. Instead, we would like to use some feature vector of fixed length describing the word, where the features give some information about the meaning. Such a representation might be used to classify or cluster words. Another example application is finding words with a meaning that is similar to that of an unknown word. The construction of $\mathbf{n}_i$ above is an example of a representation with these properties. Such representations are known as **word embeddings**.

5.2.3 Example: Clustering NIPS Words

At https://archive.ics.uci.edu/ml/datasets/NIPS+Conference+Papers+1987-2015, you can find a dataset giving word counts for each word that appears at least 50 times in the NIPS conference proceedings from 1987-2015, by paper. It's big. There are 11463 distinct words in the vocabulary, and 5811 total documents. We will use LSA to cluster words and try to expose word similarities.

First, we need to deal with practicalities. Taking the SVD of a matrix this size will present problems, and storing the result will present quite serious problems. Storing $\mathcal{X}$ is quite easy, because most of the entries are zero, and a sparse matrix representation will work. But the whole point of the exercise is that $\mathcal{X}^{(r)}$ is *not* sparse, and this will have about 10^7 entries. Nonetheless, I was able to form an SVD in R, though it took about 30 minutes on my laptop. I give some strategies for larger matrices below.

Figure 5.1 gives a fan plot of the dendrogram for the 100 most common words, for a word embedding dimension of 2 and 50. Clustering was by a simple agglomerative clusterer, using the distance in the latent space. Each word is represented by a vector that summarizes the words that co-occur in the document — the representation says nothing about *where* a word appears in the document. So words that are "close" are words that tend to be used often together *in the same document*.

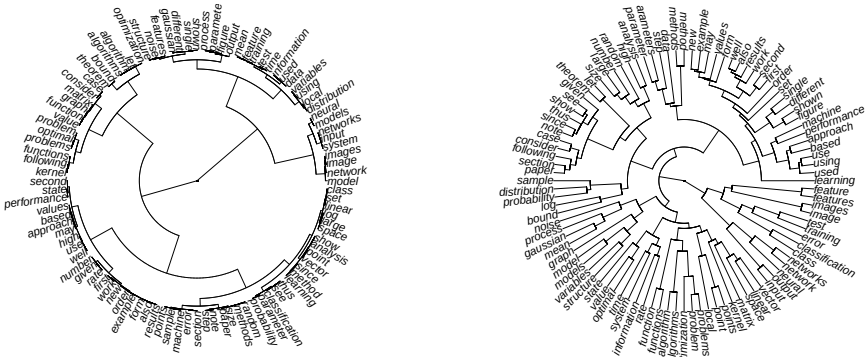


FIGURE 5.1: *A fan plot of the dendrogram obtained by clustering the most common 100 words in the NIPS word dataset in a word embedding space of dimension 2* **left** *and 50* **right**.

In turn, this gives a very rough estimate of whether words mean the same thing. For example, notice that each clustering regards "networks" and "network" as very similar (which is good), but "input" and "output" are also very similar (less good, but predictable; an author who uses one in a document is likely to use another). Look also at "use", "using" and "used"; and at "consider" and "case".

Recall for this example $\mathcal{X}$ is 5811×11463. The reason we could form and work with this matrix is that it is extremely sparse. But the whole point of our construction is that $\mathcal{X}^{(r)}$ isn't sparse. This means that forming it may be very difficult. In fact, we don't need to form $\mathcal{X}^{(r)}$ to compute the distance between two word embeddings. Because $\|\mathbf{n}_i\| = 1$ for each i, it is enough to be able to compute $\mathbf{n}_i^T \mathbf{n}_j$ for any pair of i, j. In turn, this is

$$\frac{\mathbf{w}_i^T \mathbf{w}_j}{\sqrt{\mathbf{w}_i^T \mathbf{w}_i}\sqrt{\mathbf{w}_j^T \mathbf{w}_j}}.$$

Now recall there are a lot of zeros in Σ_r, and they render most of the columns of $\mathcal{U}$ and rows of $\mathcal{V}^T$ irrelevant. In particular, write $\mathcal{U}_r$ for the $m \times r$ matrix consisting of the first r columns of $\mathcal{U}$, and so on; and write $\Sigma_r^{(s)}$ for the $r \times r$ submatrix of Σ_r with non-zero diagonal. Then we have

$$\mathcal{X}_r = \mathcal{U}\Sigma_r \mathcal{V}^T = \mathcal{U}_r \Sigma_r^{(s)} (\mathcal{V}_r)^T.$$

Now notice that

$$\begin{aligned}(\mathcal{X}^{(r)})^T\mathcal{X}^{(r)} &= \begin{bmatrix} \mathbf{w}_1^T\mathbf{w}_1 & \dots & \mathbf{w}_1^T\mathbf{w}_d \\ \dots & \dots & \dots \\ \mathbf{w}_d^T\mathbf{w}_1 & \dots & \mathbf{w}_d^T\mathbf{w}_d \end{bmatrix} \\ &= \mathcal{V}_r(\Sigma_r^{(s)})^2\mathcal{V}_r^T \\ &= (\Sigma_r^{(s)}\mathcal{V}_r^T)^T(\Sigma_r^{(s)}\mathcal{V}_r^T)\end{aligned}$$

(because $\mathcal{U}_r^T\mathcal{U}_r$ is the r dimensional identity matrix). Write

$$\mathcal{W} = (\Sigma_r^{(s)}\mathcal{V}_r^T) = [\mathbf{s}_1, \dots, \mathbf{s}_d]\,.$$

We have that

$$\mathbf{n}_i^T\mathbf{n}_j = \frac{\mathbf{w}_i^T\mathbf{w}_j}{\sqrt{\mathbf{w}_i^T\mathbf{w}_i}\sqrt{\mathbf{w}_j^T\mathbf{w}_j}} = \frac{\mathbf{s}_i^T\mathbf{s}_j}{\sqrt{\mathbf{s}_i^T\mathbf{s}_i}\sqrt{\mathbf{s}_j^T\mathbf{s}_j}}$$

so we can represent the i'th word with $\mathbf{s}_i/\|\mathbf{s}_i\|$. This is a significant improvement, because $\mathcal{W}$ has dimension $r \times 11463$ where r is the dimension of word embedding space. Furthermore, if r is a lot smaller than either N or d (which is the useful case), we could extract $\Sigma_r^{(s)}$ and $\mathcal{V}_r$ using the NIPALS strategy above.

5.3 CANONICAL CORRELATION ANALYSIS

One common kind of high dimensional dataset has two kinds of data. For example, you might have a representation of a sound, and a video sequence where the sound occurred. This happens rather often for a variety of reasons. One case involves road crossings, where accidents are rather easier to detect using sound than using video. You might want to detect accidents with sound, then preserve the video for evidence. Another case occurs when you want to choose sound clips to go with video clips automatically. Yet another case occurs when you want to learn how to read the (very small) motion cues in a video that result from sounds in a scene (so you could, say, read a conversation off the tiny wiggles in the curtain caused by the sound waves). As another example, you might have a representation of the text in a caption, and of the image that goes with the caption. Again, there are many cases. In the most common, you would predict words from pictures to label the pictures, or predict pictures from words to support image search. The important question here is: what aspects of the one kind of data can be predicted from the other?

In each case, we deal with a dataset of N pairs, $\mathbf{p}_i = [\mathbf{x}_i, \mathbf{y}_i]^T$, where $\mathbf{x}_i$ is a d_x dimensional vector representing one kind of data (eg words; sound; image; video) and $\mathbf{y}_i$ is a d_y dimensional vector representing the other kind. I will write $\{\mathbf{x}\}$ for the $\mathbf{x}$ part, etc., but notice that our agenda of prediction assumes that the pairing is significant — if you could shuffle one of the parts without affecting the outcome of the algorithm, then you couldn't predict one from the other.

We could do a principal components analysis on $\{\mathbf{p}\}$, but that approach misses the point. We are primarily interested in the relationship between $\{\mathbf{x}\}$ and $\{\mathbf{y}\}$ and the principal components capture only the major components of variance. For

example, imagine the $\mathbf{x}_i$ all have a very large scale, and the $\mathbf{y}_i$ all have a very small scale. Then the principal components will be determined by the $\mathbf{x}_i$. You could resolve this by scaling each in advance, but there is still no recipe to predict aspects of $\{\mathbf{y}\}$ from $\{\mathbf{x}\}$ (or vice versa). We assume that $\{\mathbf{x}\}$ and $\{\mathbf{y}\}$ have zero mean, because it will simplify the equations and is easy to achieve.

Canonical correlation analysis (or CCA) seeks linear projections of $\{\mathbf{x}\}$ and $\{\mathbf{y}\}$ such that one is easily predicted from the other. A projection of $\{\mathbf{x}\}$ onto one dimension can be represented by a vector $\mathbf{u}$. The projection yields a dataset $\left\{\mathbf{u}^T\mathbf{x}\right\}$ whose i'th element is $\mathbf{u}^T\mathbf{x}_i$. Assume we project $\{\mathbf{x}\}$ onto $\mathbf{u}$ and $\{\mathbf{y}\}$ onto $\mathbf{v}$. Our ability to predict one from the other is measured by the correlation of these two datasets. So we should look for $\mathbf{u}$, $\mathbf{v}$ so that

$$\mathsf{corr}\left(\left\{\mathbf{u}^T\mathbf{x}, \mathbf{v}^T\mathbf{y}\right\}\right)$$

is maximized. If you are worried that a negative correlation with a large absolute value also allows good prediction, and this isn't accounted for by the expression, you should remember that we choose the sign of $\mathbf{v}$.

We need some more notation. Write Σ for the covariance matrix of $\{\mathbf{p}\}$. Recall $\mathbf{p}_i = [\mathbf{x}_i, \mathbf{y}_i]^t$. This means the covariance matrix has a block structure, where one block is covariance of x components of $\{\mathbf{p}\}$ with each other, another is covariance of y components with each other, and the third is covariance of x components with y components. We write

$$\Sigma = \begin{bmatrix} \Sigma_{xx} & \Sigma_{xy} \\ \Sigma_{yx} & \Sigma_{yy} \end{bmatrix} = \begin{bmatrix} x-x \text{ covariance} & x-y \text{ covariance} \\ y-x \text{ covariance} & y-y \text{ covariance} \end{bmatrix}.$$

We have that

$$\mathsf{corr}\left(\left\{\mathbf{u}^T\mathbf{x}, \mathbf{v}^T\mathbf{y}\right\}\right) = \frac{\mathbf{u}^T\Sigma_{xy}\mathbf{v}}{\sqrt{\mathbf{u}^T\Sigma_{xx}\mathbf{u}}\sqrt{\mathbf{v}^T\Sigma_{yy}\mathbf{v}}}$$

and maximizing this ratio will be hard (think about what the derivatives look like). There is a useful trick. Assume $\mathbf{u}^*$, $\mathbf{v}^*$ are values at a maximum. Then they must also be solutions of the problem

$$\text{Max}\mathbf{u}^T\Sigma_{xy}\mathbf{v} \quad \text{Subject to} \quad \mathbf{u}^T\Sigma_{xx}\mathbf{u} = c_1 \text{ and } \mathbf{v}^T\Sigma_{yy}\mathbf{v} = c_2$$

(where c_1, c_2 are positive constants of no particular interest). This second problem is quite easy to solve. The Lagrangian is

$$\mathbf{u}^T\Sigma_{xy}\mathbf{v} - \lambda_1(\mathbf{u}^T\Sigma_{xx}\mathbf{u} - c_1) - \lambda_2(\mathbf{v}^T\Sigma_{yy}\mathbf{v} - c_2)$$

so we must solve

$$\begin{aligned} \Sigma_{xy}\mathbf{v} - \lambda_1\Sigma_{xx}\mathbf{u} &= 0 \\ \Sigma_{xy}^T\mathbf{u} - \lambda_2\Sigma_{yy}\mathbf{v} &= 0 \end{aligned}$$

For simplicity, we assume that there are no redundant variables in $\mathbf{x}$ or $\mathbf{y}$, so that Σ_{xx} and Σ_{yy} are both invertible. We substitute $(1/\lambda_1)\Sigma_{xx}^{-1}\Sigma_{xy}\mathbf{v} = \mathbf{u}$ and $\mathbf{w} = \Sigma_{yy}\mathbf{v}$ to get

$$\Sigma_{yy}^{-1}\Sigma_{xy}^T\Sigma_{xx}^{-1}\Sigma_{xy}\mathbf{v} = (\lambda_1\lambda_2)\mathbf{v}.$$

Similar reasoning yields

$$\Sigma_{xx}^{-1}\Sigma_{xy}\Sigma_{yy}^{-1}\Sigma_{xy}^{T}\mathbf{u} = (\lambda_1\lambda_2)\mathbf{u}.$$

So $\mathbf{u}$ and $\mathbf{v}$ are eigenvectors of the relevant matrices. But which eigenvectors? Notice that

$$\mathbf{u}^T\Sigma_{xy}\mathbf{v} = \mathbf{u}^T\left(\lambda_1\Sigma_{xx}\mathbf{u}\right) = \left(\lambda_2\mathbf{v}^T\Sigma_{yy}\right)\mathbf{v}$$

so that

$$\mathsf{corr}\left(\left\{\mathbf{u}^T\mathbf{x}, \mathbf{v}^T\mathbf{y}\right\}\right) = \frac{\mathbf{u}^T\Sigma_{xy}\mathbf{v}}{\sqrt{\mathbf{u}^T\Sigma_{xx}\mathbf{u}}\sqrt{\mathbf{v}^T\Sigma_{yy}\mathbf{v}}} = \sqrt{\lambda_1}\sqrt{\lambda_2}$$

meaning that the eigenvectors corresponding to the largest eigenvalues give the largest correlation directions, to the second largest give the second largest correlation directions, and so on. There are $\min(d_x, d_y)$ directions in total.

Worked example 5.1 *Anxiety and wildness in mice*

Compute the canonical correlations between indicators of anxiety and of wildness in mice, using the dataset at http://phenome.jax.org/db/q?rtn=projects/details&sym=Jaxpheno7

Solution: You should read the details on the web page that publishes the data. The anxiety indicators are: `transfer_arousal, freeze, activity, tremor, twitch, defecation_jar, urination_jar, defecation_arena, urination_arena`, and the wildness indicators are: `biting, irritability, aggression, vocal, finger_approach`. After this, it's just a question of finding a package and putting the data in it. I used R's `cancor`, and found the following five canonical correlations: 0.62, 0.53, 0.40, 0.35, 0.30. You shouldn't find the presence of strong correlations shocking (anxious mice should be bitey), but we don't have any evidence this isn't an accident. The example in the subsection below goes into this question in more detail.
This data was collected by The Jackson Laboratory, who ask it be cited as: Neuromuscular and behavioral testing in males of 6 inbred strains of mice. MPD:Jaxpheno7. Mouse Phenome Database web site, The Jackson Laboratory, Bar Harbor, Maine USA. http://phenome.jax.org

5.3.1 Example: CCA of Albedo and Shading

Here is a classical computer vision problem. The brightness of a diffuse (=dull, not shiny or glossy) surface in an image is the product of two effects: the **albedo** (the percentage of incident light that it reflects) and the **shading** (the amount of light incident on the surface). We will observe the brightness in an image, and the problem is to recover the albedo and the shading separately. This is a problem that dates back to the early 70's, but still gets regular and significant attention in the computer vision literature, because it's hard, and because it seems to be important.

We will confine our discussion to smooth (=not rough) surfaces, to prevent the complexity spiralling out of control. Albedo changes at marks on the surface, so a dark surface has low albedo and a light surface has high albedo. Shading is a property of the geometry of the light sources with respect to the surface. When you move an object around in a room, its shading may change a lot (though people are surprisingly bad at noticing), but its albedo doesn't change at all. To change an object's albedo, you need something like a marker (or paint, etc.). All this suggests that a CCA of albedo against shading will suggest there is no correlation.

Because this is a classical problem, there are datasets one can download. There is a very good dataset giving the albedo and shading for images, collected by Roger Grosse, Micah K. Johnson, Edward H. Adelson, and William T. Freeman at http://www.cs.toronto.edu/~rgrosse/intrinsic/. These images show individual objects on black backgrounds, and there are masks identifying object pixels. I constructed random 11×11 tiles of albedo and shading for each of the 20 objects depicted. I chose 20 tiles per image (so 400 in total), centered at random locations, but chosen so that every pixel in a tile lies on an object pixel. I then reshaped each tile into a 121 dimensional vector, and computed a CCA. The top 10 values of canonical correlations I obtained were: 0.96, 0.94, 0.93, 0.93, 0.92, 0.92, 0.91, 0.91, 0.90, 0.88.

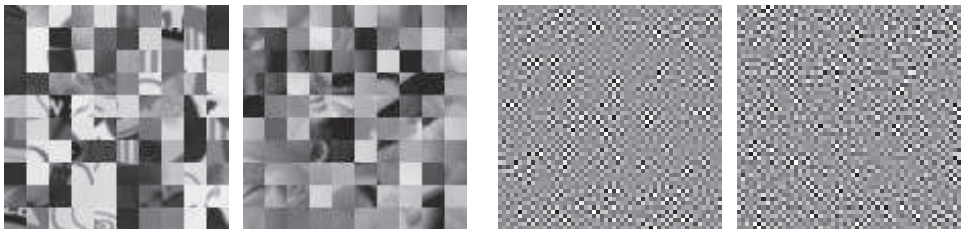
This should strike you as deeply alarming: how could albedo and shading be correlated? The correct answer is that they are not, but careless analysis might suggest they are. The difficulty is that the objective function we are maximizing is a ratio

$$\text{corr}\left(\left\{\mathbf{u}^T\mathbf{x}, \mathbf{v}^T\mathbf{y}\right\}\right) = \frac{\mathbf{u}^T\Sigma_{xy}\mathbf{v}}{\sqrt{\mathbf{u}^T\Sigma_{xx}\mathbf{u}}\sqrt{\mathbf{v}^T\Sigma_{yy}\mathbf{v}}}.$$

Now look at the denominator of this fraction, and recall our work on PCA. The whole point of PCA is that there are many directions $\mathbf{u}$ such that $\mathbf{u}^T\Sigma_{xx}\mathbf{u}$ is small — these are the directions that we can drop in building low dimensional models. But now they have a potential to be a significant nuisance. We could have the objective function take a large value simply because the terms in the denominator are very small. This is what happens in the case of albedo and shading. You can check this by looking at Figure 5.2, or by actually looking at the size of the canonical correlation directions (the $\mathbf{u}$'s and $\mathbf{v}$'s). You will find that, if you compute $\mathbf{u}$ and $\mathbf{v}$ using the procedure I described, these vectors have large magnitude (I found magnitudes of the order of 100). This suggests, correctly, that they're associated with small eigenvalues in Σ_{xx} and Σ_{yy}.

Just a quick check with intuition and an image tells us that these canonical correlations don't mean what we think. But this works only for a situation where we have intuition, etc. We need a test that tells whether the large correlation values have arisen by accident.

There is an easy and useful strategy for testing this. If there really are meaningful correlations between the $\{\mathbf{x}\}$ and $\{\mathbf{y}\}$, they should be disrupted if we reorder the datasets. So if something important is changed by permuting one dataset, there is evidence that there is a meaningful correlation. The recipe is straightforward. We choose a method to summarize the canonical correlations in a number (this is a statistic; if you don't remember the term, it's in the backup material). In the



Albedo tiles (10x10) Shading tiles (10x10) Albedo CC's (5x5) Shading CC's (5x5)

FIGURE 5.2: *On the* **left**, *a* 10×10 *grid of tiles of albedo (***far left***) and shading (***center left***), taken from Grosse et al's data set. The position of the tiles is keyed, so (for example) the albedo tile at 3, 5 corresponds to the shading tile at 3, 5. On the* **right**, *the first 25 canonical correlation directions for albedo (***center right***) and shading (***far right***). I have reshaped these into tiles and zoomed them. The scale is smallest value is black, and largest white. These are ordered so the pair with highest correlation is at the top left, next highest is one step to the right, etc. You should notice that these directions do not look even slightly like the patterns in the original tiles, or like any pattern you expect to encounter in a real image. This is because they're not: these are directions that have very small variance.*

case of canonical correlations, the usual choice is **Wilks' lambda** (or **Wilks'** λ if you're fussy). Write ρ_i for the i'th canonical correlation. Wilks' lambda is

$$\prod_{i=1}^{i=\min(d_x,d_y)} (1-\rho_i^2).$$

Notice if there are a lot of strong correlations, we should get a small number. We now compute that number for the dataset we have. We then construct a collection of new datasets by randomly reordering the items in $\{\mathbf{y}\}$, and for each we compute the value of the statistic. This gives an estimate of the distribution of values of Wilks' lambda available *if there is no correlation*. We then ask what fraction of the reordered datasets have an even smaller value of Wilk's lambda than the observed value. If this fraction is small, then it is unlikely that the correlations we observed arose by accident. All this is fairly easily done using a package (I used CCP in R).

Figure 5.3 shows what happens for the mouse canonical correlation of example 5.1. You should notice that this is a significance test, and follows the usual recipe for such tests except that we estimate the distribution of the statistic empirically. Here about 97% of random permutations have a larger value of the Wilks' lambda than that of the original data, which means that we would see the canonical correlation values we see only about once in 30 experiments if they were purely a chance effect. You should read this as quite good evidence there is a correlation.

The approach involves a fair amount of computation for a large dataset. Figure 5.4 shows what happens for albedo and shading. The figure is annoying to interpret, because the value of the Wilks' lambda is extremely small; the big point is that almost every permutation of the data has an even smaller value of the

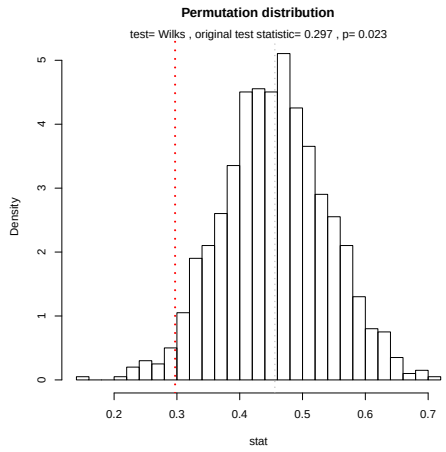


FIGURE 5.3: *A histogram of values of Wilks' lambda obtained from permuted versions of the mouse dataset of example 5.1. The value obtained for the original dataset is shown by the vertical line. Notice that most values are larger (about* 97% *of values), meaning that we would see the canonical correlation values we see only about once in 30 experiments if they were purely a chance effect. There is very likely a real effect here.*

Wilks' lambda — the correlations are entirely an accident, and are of no statistical significance.

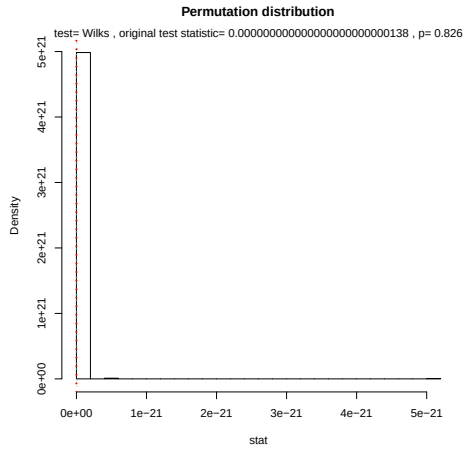


FIGURE 5.4: *A histogram of values of Wilks' lambda obtained from permuted versions of the 400 tile albedo shading dataset discussed in the text. The value obtained for the original dataset is shown by the vertical line, and is really tiny (rather less than 1e-21). But rather more than half of the values obtained by permuting the data are even tinier, meaning that we would see the canonical correlation values we see or smaller about 4 in every 5 experiments if they were purely a chance effect. There is no reason to believe the two have a correlation.*

CHAPTER 6

Clustering: Models of High Dimensional Data

High-dimensional data comes with problems. Data points tend not to be where you think; they can scattered quite far apart, and can be quite far from the mean. Except in special cases, the only really reliable probability model is the Gaussian (or Gaussian blob, or blob).

There is an important rule of thumb for coping with high dimensional data: **Use simple models**. A blob is a good simple model. Modelling data as a blob involves computing its mean and its covariance. Sometimes, as we shall see, the covariance can be hard to compute. Even so, a blob model is really useful. It is natural to try and extend this model to cover datasets that don't obviously consist of a single blob.

One very good, very simple, model for high dimensional data is to assume that it consists of multiple blobs. To build models like this, we must determine (a) what the blob parameters are and (b) which datapoints belong to which blob. Generally, we will collect together data points that are close and form blobs out of them. This process is known as **clustering**.

Clustering is a somewhat puzzling activity. It is extremely useful to cluster data, and it seems to be quite important to do it reasonably well. But it surprisingly hard to give crisp criteria for a good (resp. bad) clustering of a dataset. Usually, clustering is part of building a model, and the main way to know that the clustering algorithm is bad is that the model is bad.

6.1 THE CURSE OF DIMENSION

High dimensional models display uninituitive behavior (or, rather, it can take years to make your intuition see the true behavior of high-dimensional models as natural). In these models, most data lies in places you don't expect. We will do several simple calculations with an easy high-dimensional distribution to build some intuition.

6.1.1 The Curse: Data isn't Where You Think it is

Assume our data lies within a cube, with edge length two, centered on the origin. This means that each component of $\mathbf{x}_i$ lies in the range $[-1, 1]$. One simple model for such data is to assume that each dimension has uniform probability density in this range. In turn, this means that $P(x) = \frac{1}{2^d}$. The mean of this model is at the origin, which we write as $\mathbf{0}$.

The first surprising fact about high dimensional data is that most of the data can lie quite far away from the mean. For example, we can divide our dataset into two pieces. $\mathcal{A}(\epsilon)$ consists of all data items where *every* component of the data has

a value in the range $[-(1-\epsilon),(1-\epsilon)]$. $\mathcal{B}(\epsilon)$ consists of all the rest of the data. If you think of the data set as forming a cubical orange, then $\mathcal{B}(\epsilon)$ is the rind (which has thickness ϵ) and $\mathcal{A}(\epsilon)$ is the fruit.

Your intuition will tell you that there is more fruit than rind. This is true, for three dimensional oranges, but not true in high dimensions. The fact that the orange is cubical just simplifies the calculations, but has nothing to do with the real problem.

We can compute $P(\{\mathbf{x} \in \mathcal{A}(\epsilon)\})$ and $P(\{\mathbf{x} \in \mathcal{A}(\epsilon)\})$. These probabilities tell us the probability a data item lies in the fruit (resp. rind). $P(\{\mathbf{x} \in \mathcal{A}(\epsilon)\})$ is easy to compute as

$$P(\{\mathbf{x} \in \mathcal{A}(\epsilon)\}) = (2(1-\epsilon)))^d \left(\frac{1}{2^d}\right) = (1-\epsilon)^d$$

and

$$P(\{\mathbf{x} \in \mathcal{B}(\epsilon)\}) = 1 - P(\{\mathbf{x} \in \mathcal{A}(\epsilon)\}) = 1 - (1-\epsilon)^d.$$

But notice that, as $d \to \infty$,

$$P(\{\mathbf{x} \in \mathcal{A}(\epsilon)\}) \to 0.$$

This means that, for large d, we expect most of the data to be in $\mathcal{B}(\epsilon)$. Equivalently, for large d, we expect that at least one component of each data item is close to either 1 or -1.

This suggests (correctly) that much data is quite far from the origin. It is easy to compute the average of the squared distance of data from the origin. We want

$$\mathbb{E}\left[\mathbf{x}^T\mathbf{x}\right] = \int_{\text{box}} \left(\sum_i x_i^2\right) P(\mathbf{x}) d\mathbf{x}$$

but we can rearrange, so that

$$\mathbb{E}\left[\mathbf{x}^T\mathbf{x}\right] = \sum_i \mathbb{E}\left[x_i^2\right] = \sum_i \int_{\text{box}} x_i^2 P(\mathbf{x}) d\mathbf{x}.$$

Now each component of $\mathbf{x}$ is independent, so that $P(\mathbf{x}) = P(x_1)P(x_2)\ldots P(x_d)$. Now we substitute, to get

$$\mathbb{E}\left[\mathbf{x}^T\mathbf{x}\right] = \sum_i \mathbb{E}\left[x_i^2\right] = \sum_i \int_{-1}^{1} x_i^2 P(x_i) dx_i = \sum_i \frac{1}{2}\int_{-1}^{1} x_i^2 dx_i = \frac{d}{3},$$

so as d gets bigger, most data points will be further and further from the origin. Worse, as d gets bigger, data points tend to get further and further from one another. We can see this by computing the average of the squared distance of data points from one another. Write $\mathbf{u}$ for one data point and $\mathbf{v}$; we can compute

$$\mathbb{E}\left[d(\mathbf{u},\mathbf{v})^2\right] = \int_{\text{box}}\int_{\text{box}} \sum_i (u_i - v_i)^2 d\mathbf{u} d\mathbf{v} = \mathbb{E}\left[\mathbf{u}^T\mathbf{u}\right] + \mathbb{E}\left[\mathbf{v}^T\mathbf{v}\right] - 2\mathbb{E}\left[\mathbf{u}^T\mathbf{v}\right]$$

but since $\mathbf{u}$ and $\mathbf{v}$ are independent, we have $\mathbb{E}\left[\mathbf{u}^T\mathbf{v}\right] = \mathbb{E}[\mathbf{u}]^T\mathbb{E}[\mathbf{v}] = 0$. This yields

$$\mathbb{E}\left[d(\mathbf{u},\mathbf{v})^2\right] = 2\frac{d}{3}.$$

This means that, for large d, we expect our data points to be quite far apart.

6.1.2 Minor Banes of Dimension

High dimensional data presents a variety of important practical nuisances which follow from the curse of dimension. It is hard to estimate covariance matrices, and it is hard to build histograms.

Covariance matrices are hard to work with for two reasons. The number of entries in the matrix grows as the square of the dimension, so the matrix can get big and so difficult to store. More important, the amount of data we need to get an accurate estimate of all the entries in the matrix grows fast. As we are estimating more numbers, we need more data to be confident that our estimates are reasonable. There are a variety of straightforward work-arounds for this effect. In some cases, we have so much data there is no need to worry. In other cases, we assume that the covariance matrix has a particular form, and just estimate those parameters. There are two strategies that are usual. In one, we assume that the covariance matrix is diagonal, and estimate only the diagonal entries. In the other, we assume that the covariance matrix is a scaled version of the identity, and just estimate this scale. You should see these strategies as acts of desperation, to be used only when computing the full covariance matrix seems to produce more problems than using these approaches.

It is difficult to build histogram representations for high dimensional data. The strategy of dividing the domain into boxes, then counting data into them, fails miserably because there are too many boxes. In the case of our cube, imagine we wish to divide each dimension in half (i.e. between $[-1, 0]$ and between $[0, 1]$). Then we must have 2^d boxes. This presents two problems. First, we will have difficulty representing this number of boxes. Second, unless we are exceptionally lucky, most boxes must be empty because we will not have 2^d data items.

However, one representation is extremely effective. We can represent data as a collection of **clusters** — coherent blobs of similar datapoints that could, under appropriate circumstances, be regarded as the same. We could then represent the dataset by, for example, the center of each cluster and the number of data items in each cluster. Since each cluster is a blob, we could also report the covariance of each cluster, if we can compute it.

Remember this: *High dimensional data does not behave in a way that is consistent with most people's intuition. Points are always close to the boundary and further apart than you think. This property makes a nuisance of itself in a variety of ways. The most important is that only the simplest models work well in high dimensions.*

6.2 THE MULTIVARIATE NORMAL DISTRIBUTION

All the nasty facts about high dimensional data, above, suggest that we need to use quite simple probability models. By far the most important model is the multivariate normal distribution, which is quite often known as the multivariate gaussian distribution. There are two sets of parameters in this model, the mean μ and the covariance Σ. For a d-dimensional model, the mean is a d-dimensional column vector and the covariance is a $d \times d$ dimensional matrix. The covariance is a symmetric matrix. For our definitions to be meaningful, the covariance matrix must be positive definite.

The form of the distribution $p(\mathbf{x}|\mu, \Sigma)$ is

$$p(\mathbf{x}|\mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^d \det(\Sigma)}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right).$$

The following facts explain the names of the parameters:

Useful Facts: 6.1 *Parameters of a Multivariate Normal Distribution*

Assuming a multivariate normal distribution, we have

- $\mathbb{E}[\mathbf{x}] = \mu$, meaning that the mean of the distribution is μ.
- $\mathbb{E}\left[(\mathbf{x}-\mu)(\mathbf{x}-\mu)^T\right] = \Sigma$, meaning that the entries in Σ represent covariances.

Assume I know have a dataset of items $\mathbf{x}_i$, where i runs from 1 to N, and we wish to model this data with a multivariate normal distribution. The maximum likelihood estimate of the mean, $\hat{\mu}$, is

$$\hat{\mu} = \frac{\sum_i \mathbf{x}_i}{N}$$

(which is quite easy to show). The maximum likelihood estimate of the covariance, $\hat{\Sigma}$, is

$$\hat{\Sigma} = \frac{\sum_i (\mathbf{x}_i - \hat{\mu})(\mathbf{x}_i - \hat{\mu})^T}{N}$$

(which is rather a nuisance to show, because you need to know how to differentiate a determinant). These facts mean that we already know most of what is interesting about multivariate normal distributions (or gaussians).

6.2.1 Affine Transformations and Gaussians

Gaussians behave very well under affine transformations. In fact, we've already worked out all the math. Assume I have a dataset $\mathbf{x}_i$. The mean of the maximum likelihood gaussian model is $\mathsf{mean}(\{\mathbf{x}_i\})$, and the covariance is $\mathsf{Covmat}(\{\mathbf{x}_i\})$. I can now transform the data with an affine transformation, to get $\mathbf{y}_i = \mathcal{A}\mathbf{x}_i + \mathbf{b}$.

The mean of the maximum likelihood gaussian model for the transformed dataset is $\mathsf{mean}\left(\{\mathbf{y}_i\}\right)$, and we've dealt with this; similarly, the covariance is $\mathsf{Covmat}\left(\{\mathbf{y}_i\}\right)$, and we've dealt with this, too.

A very important point follows in an obvious way. I can apply an affine transformation to any multivariate gaussian to obtain one with (a) zero mean and (b) independent components. In turn, this means that, *in the right coordinate system*, any gaussian is a product of zero mean one-dimensional normal distributions. This fact is quite useful. For example, it means that simulating multivariate normal distributions is quite straightforward — you could simulate a standard normal distribution for each component, then apply an affine transformation.

6.2.2 Plotting a 2D Gaussian: Covariance Ellipses

There are some useful tricks for plotting a 2D Gaussian, which are worth knowing both because they're useful, and they help to understand Gaussians. Assume we are working in 2D; we have a Gaussian with mean μ (which is a 2D vector), and covariance Σ (which is a 2x2 matrix). We could plot the collection of points $\mathbf{x}$ that has some fixed value of $p(\mathbf{x}|\mu, \Sigma)$. This set of points is given by:

$$\frac{1}{2}\left((\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right) = c^2$$

where c is some constant. I will choose $c^2 = \frac{1}{2}$, because the choice doesn't matter, and this choice simplifies some algebra. You might recall that a set of points $\mathbf{x}$ that satisfies a quadratic like this is a conic section. Because Σ (and so Σ^{-1}) is positive definite, the curve is an ellipse. There is a useful relationship between the geometry of this ellipse and the Gaussian.

This ellipse — like all ellipses — has a major axis and a minor axis. These are at right angles, and meet at the center of the ellipse. We can determine the properties of the ellipse in terms of the Gaussian quite easily. The geometry of the ellipse isn't affected by rotation or translation, so we will translate the ellipse so that $\mu = \mathbf{0}$ (i.e. the mean is at the origin) and rotate it so that Σ^{-1} is diagonal. Writing $\mathbf{x} = [x, y]$ we get that the set of points on the ellipse satisfies

$$\frac{1}{2}(\frac{1}{k_1^2}x^2 + \frac{1}{k_2^2}y^2) = \frac{1}{2}$$

where $\frac{1}{k_1^2}$ and $\frac{1}{k_2^2}$ are the diagonal elements of Σ^{-1}. We will assume that the ellipse has been rotated so that $k_1 < k_2$. The points $(k_1, 0)$ and $(-k_1, 0)$ lie on the ellipse, as do the points $(0, k_2)$ and $(0, -k_2)$. The major axis of the ellipse, in this coordinate system, is the x-axis, and the minor axis is the y-axis. In this coordinate system, x and y are independent. If you do a little algebra, you will see that the standard deviation of x is $\mathsf{abs}\left(k_1\right)$ and the standard deviation of y is $\mathsf{abs}\left(k_2\right)$. So the ellipse is longer in the direction of largest standard deviation and shorter in the direction of smallest standard deviation.

Now rotating the ellipse is means we will pre- and post-multiply the covariance matrix with some rotation matrix. Translating it will move the origin to the mean. As a result, the ellipse has its center at the mean, its major axis is in the direction of the eigenvector of the covariance with largest eigenvalue, and its minor axis is

in the direction of the eigenvector with smallest eigenvalue. A plot of this ellipse, which can be coaxed out of most programming environments with relatively little effort, gives us a great deal of information about the underlying Gaussian. These ellipses are known as **covariance ellipses**.

Remember this: *The multivariate normal distribution has the form*

$$p(\mathbf{x}|\mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^d det(\Sigma)}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right).$$

Assume you wish to model a dataset $\{\mathbf{x}\}$ *with a multivariate normal distribution. The maximum likelihood estimate of the mean is* $\mathsf{mean}(\{\mathbf{x}\})$. *The maximum likelihood estimate of the covariance* Σ *is* $\mathsf{Covmat}(\{\mathbf{x}\})$.

6.3 AGGLOMERATIVE AND DIVISIVE CLUSTERING

There are two natural recipes you can use to produce clustering algorithms. In **agglomerative clustering**, you start with each data item being a cluster, and then merge clusters recursively to yield a good clustering (procedure 6.2). The difficulty here is that we need to know a good way to measure the distance between clusters, which can be somewhat harder than the distance between points. In **divisive clustering**, you start with the entire data set being a cluster, and then split clusters recursively to yield a good clustering (procedure **??**). The difficulty here is we need to know some criterion for splitting clusters.

Procedure: 6.1 *Agglomerative Clustering*

Choose an inter-cluster distance. Make each point a separate cluster. Now, until the clustering is satisfactory,

- Merge the two clusters with the smallest inter-cluster distance.

Procedure: 6.2 *Divisive Clustering*

Choose a splitting criterion. Regard the entire dataset as a single cluster. Now, until the clustering is satisfactory,

- choose a cluster to split;
- then split this cluster into two parts.

To turn these recipes into algorithms requires some more detail. For agglomerative clustering, we need to choose a good inter-cluster distance to fuse nearby clusters. Even if a natural distance between data points is available, there is no canonical inter-cluster distance. Generally, one chooses a distance that seems appropriate for the data set. For example, one might choose the distance between the closest elements as the inter-cluster distance, which tends to yield extended clusters (statisticians call this method **single-link clustering**). Another natural choice is the maximum distance between an element of the first cluster and one of the second, which tends to yield rounded clusters (statisticians call this method **complete-link clustering**). Finally, one could use an average of distances between elements in the cluster, which also tends to yield rounded clusters (statisticians call this method **group average clustering**).

For divisive clustering, we need a splitting method. This tends to be something that follows from the logic of the application, because the ideal is an efficient method to find a natural split in a large dataset. We won't pursue this question further.

Finally, we need to know when to stop. This is an intrinsically difficult task if there is no model for the process that generated the clusters. The recipes I have described generate a hierarchy of clusters. Usually, this hierarchy is displayed to a user in the form of a **dendrogram**—a representation of the structure of the hierarchy of clusters that displays inter-cluster distances—and an appropriate choice of clusters is made from the dendrogram (see the example in Figure 6.1).

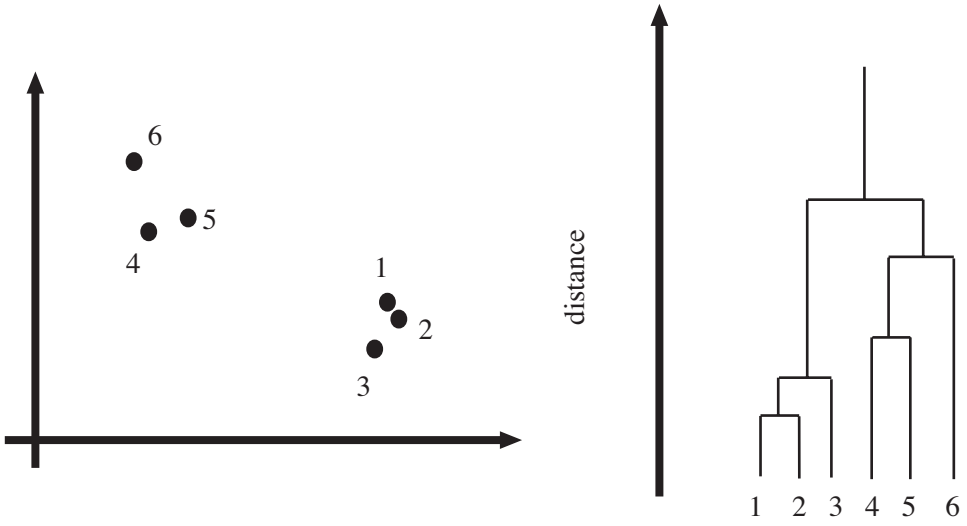


FIGURE 6.1: **Left**, *a data set;* **right**, *a dendrogram obtained by agglomerative clustering using single-link clustering. If one selects a particular value of distance, then a horizontal line at that distance splits the dendrogram into clusters. This representation makes it possible to guess how many clusters there are and to get some insight into how good the clusters are.*

Another important thing to notice about clustering from the example of figure 6.1 is that there is no right answer. There are a variety of different clusterings of the same data. For example, depending on what scales in that figure mean, it might be right to zoom out and regard all of the data as a single cluster, or to zoom in and regard each data point as a cluster. Each of these representations may be useful.

6.3.1 Clustering and Distance

In the algorithms above, and in what follows, we assume that the features are scaled so that distances (measured in the usual way) between data points are a good representation of their similarity. This is quite an important point. For example, imagine we are clustering data representing brick walls. The features might contain several distances: the spacing between the bricks, the length of the wall, the height of the wall, and so on. If these distances are given in the same set of units, we could have real trouble. For example, assume that the units are centimeters. Then the spacing between bricks is of the order of one or two centimeters, but the heights of the walls will be in the hundreds of centimeters. In turn, this means that the distance between two datapoints is likely to be completely dominated by the height and length data. This could be what we want, but it might also not be a good thing.

There are some ways to manage this issue. One is to know what the features measure, and know how they should be scaled. Usually, this happens because you have a deep understanding of your data. If you don't (which happens!), then it is often a good idea to try and normalize the scale of the data set. There are two good strategies. The simplest is to translate the data so that it has zero mean (this is just for neatness - translation doesn't change distances), then scale each direction so that it has unit variance. More sophisticated is to translate the data so that it has zero mean, then transform it so that each direction is independent and has unit variance. Doing so is sometimes referred to as **decorrelation** or **whitening** (because you make the data more like white noise); I described how to do this in section **??**.

Worked example 6.1 *Agglomerative clustering*

Cluster the seed dataset from the UC Irvine Machine Learning Dataset Repository (you can find it at http://archive.ics.uci.edu/ml/datasets/seeds).

Solution: Each item consists of seven measurements of a wheat kernel; there are three types of wheat represented in this dataset. For this example, I used Matlab, but many programming environments will provide tools that are useful for agglomerative clustering. I show a dendrogram in figure **??**). I deliberately forced Matlab to plot the whole dendrogram, which accounts for the crowded look of the figure (you can allow it to merge small leaves, etc.). As you can see from the dendrogram and from Figure 6.3, this data clusters rather well.

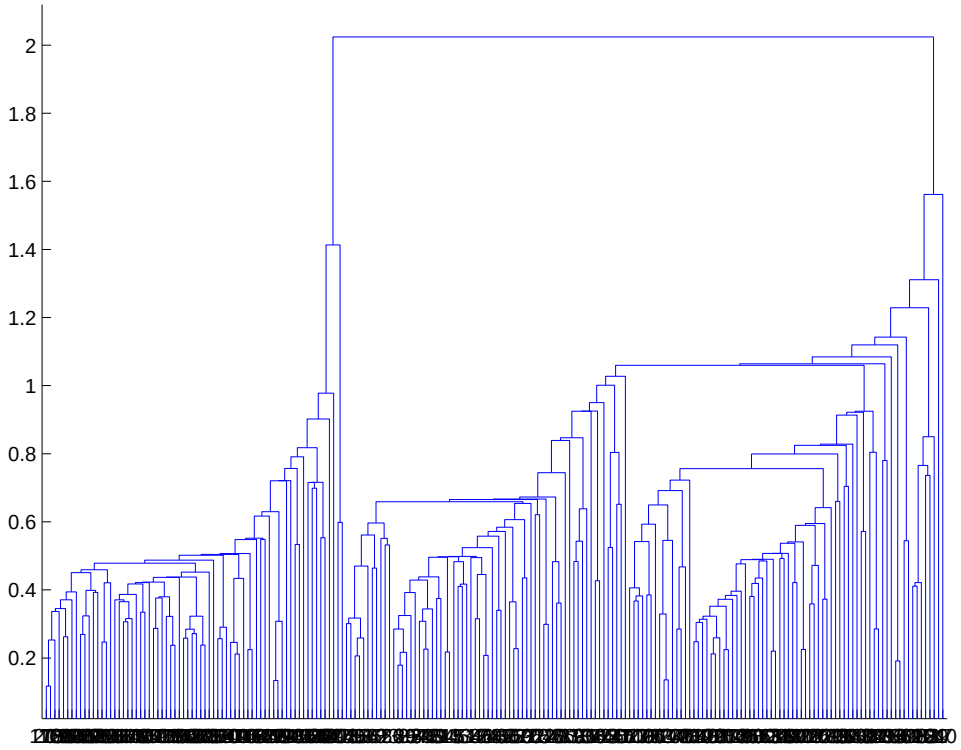


FIGURE 6.2: *A dendrogram obtained from the seed dataset, using single link clustering. Recall that the data points are on the horizontal axis, and that the vertical axis is distance; there is a horizontal line linking two clusters that get merged, established at the height at which they're merged. I have plotted the entire dendrogram, despite the fact it's a bit crowded at the bottom, because you can now see how clearly the data set clusters into a small set of clusters — there are a small number of vertical "runs".*

Remember this: *Agglomerative clustering starts with each data point a cluster, then recursively merges. There are three main ways to compute the distance between clusters. Divisive clustering starts with all in one cluster, then recursively splits. Choosing a split can be tricky.*

6.4 THE K-MEANS ALGORITHM AND VARIANTS

Assume we have a dataset that, we believe, forms many clusters that look like blobs. If we knew where the center of each of the clusters was, it would be easy to

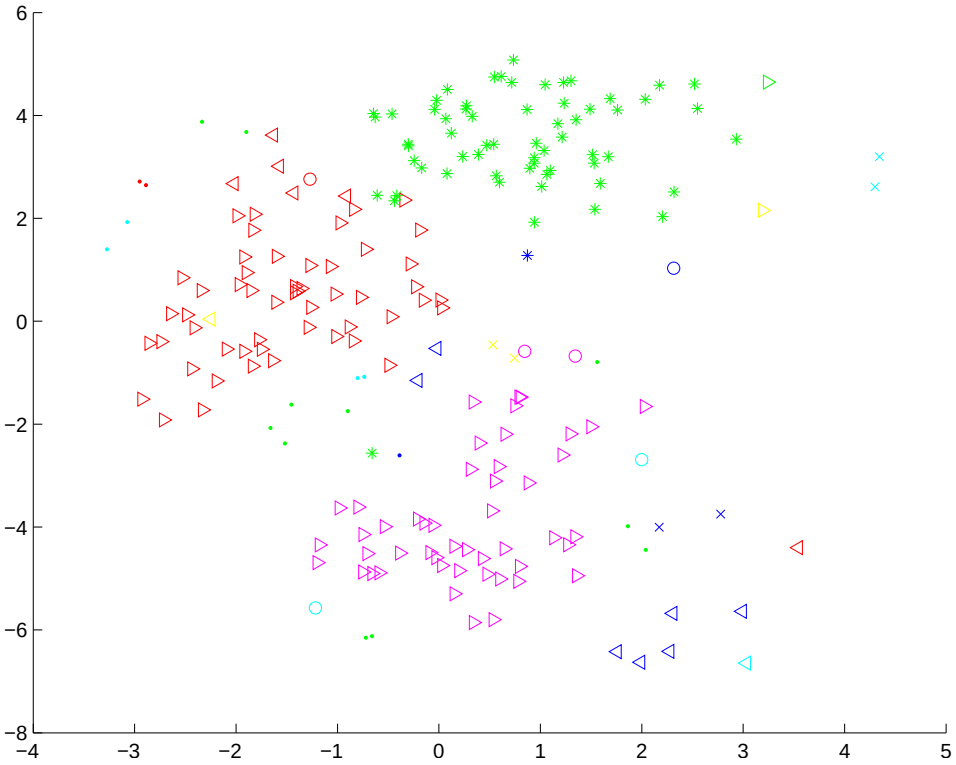


FIGURE 6.3: *A clustering of the seed dataset, using agglomerative clustering, single link distance, and requiring a maximum of 30 clusters. I have plotted each cluster with a distinct marker (though some markers differ only by color; you might need to look at the PDF version to see this figure at its best). Notice that there are a set of fairly natural isolated clusters. The original data is 8 dimensional, which presents plotting problems; I show a scatter plot on the first two principal components (though I computed distances for clustering in the original 8 dimensional space).*

tell which cluster each data item belonged to — it would belong to the cluster with the closest center. Similarly, if we knew which cluster each data item belonged to, it would be easy to tell where the cluster centers were — they'd be the mean of the data items in the cluster. This is the point closest to every point in the cluster.

We can turn these observations into an algorithm. Assume that we know how many clusters there are in the data, and write k for this number. The ith data item to be clustered is described by a feature vector $\mathbf{x}_i$. We write $\mathbf{c}_j$ for the center of the jth cluster. We assume that most data items are close to the center of their cluster. This suggests that we cluster the data by minimizing the the cost function

$$\Phi(\text{clusters}, \text{data}) = \sum_{j \in \text{clusters}} \left\{ \sum_{i \in i\text{th cluster}} (\mathbf{x}_i - \mathbf{c}_j)^T (\mathbf{x}_i - \mathbf{c}_j) \right\}.$$

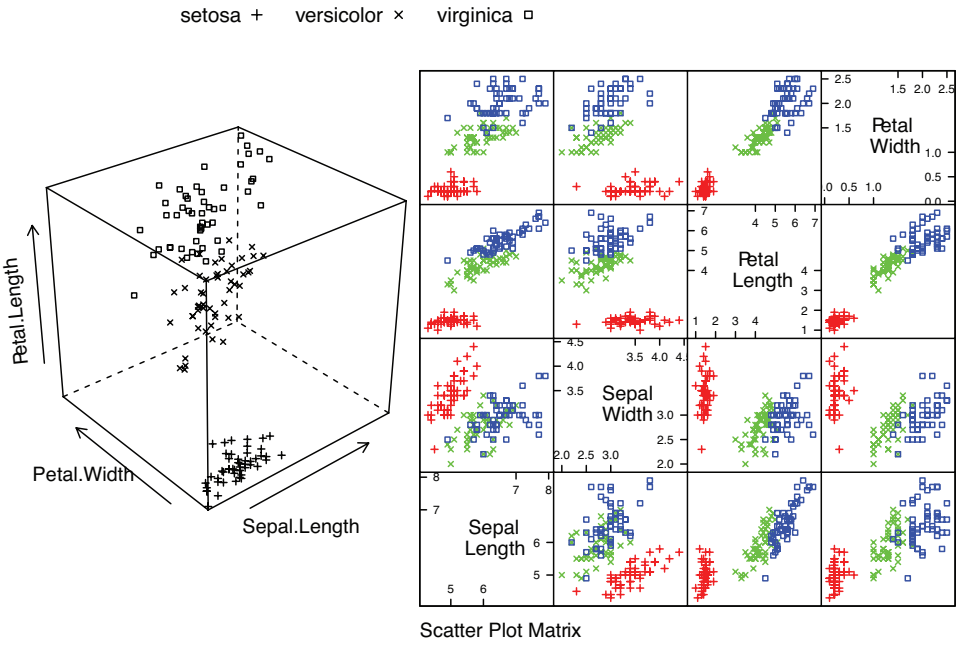


FIGURE 6.4: **Left:** *a 3D scatterplot for the famous Iris data, collected by Edgar Anderson in 1936, and made popular amongst statisticians by Ronald Fisher in that year. I have chosen three variables from the four, and have plotted each species with a different marker. You can see from the plot that the species cluster quite tightly, and are different from one another.* **Right:** *a scatterplot matrix for the Iris data. There are four variables, measured for each of three species of iris. I have plotted each species with a different marker. You can see from the plot that the species cluster quite tightly, and are different from one another.*

Notice that if we know the center for each cluster, it is easy to determine which cluster is the best choice for each point. Similarly, if the allocation of points to clusters is known, it is easy to compute the best center for each cluster. However, there are far too many possible allocations of points to clusters to search this space for a minimum. Instead, we define an algorithm that iterates through two activities:

- Assume the cluster centers are known and, allocate each point to the closest cluster center.
- Assume the allocation is known, and choose a new set of cluster centers. Each center is the mean of the points allocated to that cluster.

We then choose a start point by randomly choosing cluster centers, and then iterate these stages alternately. This process eventually converges to a local minimum of the objective function (the value either goes down or is fixed at each step, and it is bounded below). It is not guaranteed to converge to the global minimum of the objective function, however. It is also not guaranteed to produce k clusters,

unless we modify the allocation phase to ensure that each cluster has some nonzero number of points. This algorithm is usually referred to as **k-means** (summarized in Algorithm 6.3).

Procedure: 6.3 *K-Means Clustering*

Choose k. Now choose k data points $\mathbf{c}_j$ to act as cluster centers. Until the cluster centers change very little

- Allocate each data point to cluster whose center is nearest.
- Now ensure that every cluster has at least one data point; one way to do this is by supplying empty clusters with a point chosen at random from points far from their cluster center.
- Replace the cluster centers with the mean of the elements in their clusters.

Usually, we are clustering high dimensional data, so that visualizing clusters can present a challenge. If the dimension isn't too high, then we can use panel plots. An alternative is to project the data onto two principal components, and plot the clusters there; the process for plotting 2D covariance ellipses from section 6.2.2 comes in useful here. A natural dataset to use to explore k-means is the iris data, where we know that the data should form three clusters (because there are three species). Recall this dataset from section **??**. I reproduce figure 4.3 from that section as figure 6.8, for comparison. Figures 6.5, **??** and **??** show different k-means clusterings of that data.

Worked example 6.2 *K-means clustering in R*

Cluster the iris dataset into two clusters using k-means, then plot the results on the first two principal components

Solution: I used the code fragment in listing 6.1, which produced figure **??**

6.4.1 How to choose K

The iris data is just a simple example. We know that the data forms clean clusters, and we know there should be three of them. Usually, we don't know how many clusters there should be, and we need to choose this by experiment. One strategy is to cluster for a variety of different values of k, then look at the value of the cost function for each. If there are more centers, each data point can find a center that is close to it, so we expect the value to go down as k goes up. This means that

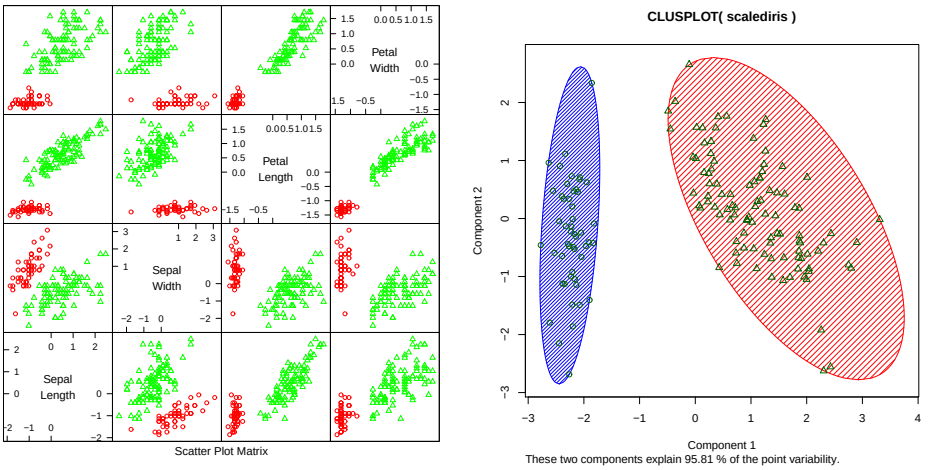


FIGURE 6.5: *On the* **left**, *a panel plot of the iris data clustered using k-means with* $k = 2$. *By comparison with figure 6.8, notice how the* versicolor *and* verginica *clusters appear to have been merged. On the* **right**, *this data set projected onto the first two principal components, with one blob drawn over each cluster.*

looking for the k that gives the smallest value of the cost function is not helpful, because that k is always the same as the number of data points (and the value is then zero). However, it can be very helpful to plot the value as a function of k, then look at the "knee" of the curve. Figure 6.8 shows this plot for the iris data. Notice that $k = 3$ — the "true" answer — doesn't look particularly special, but $k = 2$, $k = 3$, or $k = 4$ all seem like reasonable choices. It is possible to come up with a procedure that makes a more precise recommendation by penalizing clusterings that use a large k, because they may represent inefficient encodings of the data. However, this is often not worth the bother.

Listing 6.1: R code for iris example.

```
setwd('/users/daf/Current/courses/Probcourse/Clustering/RCode')
#library('lattice')
# work with iris dataset this is famous, and included in R
# there are three species
head(iris)
#
library('cluster')
numiris=iris[,c(1, 2, 3, 4)]  #the numerical values
#scalediris<-scale(numiris) # scale to unit variance
nclus<-2
sfit<-kmeans(numiris, nclus)
colr<-c('red', 'green', 'blue', 'yellow', 'orange')
clusplot(numiris, sfit$cluster, color=TRUE, shade=TRUE,
         labels=0, lines=0)
```

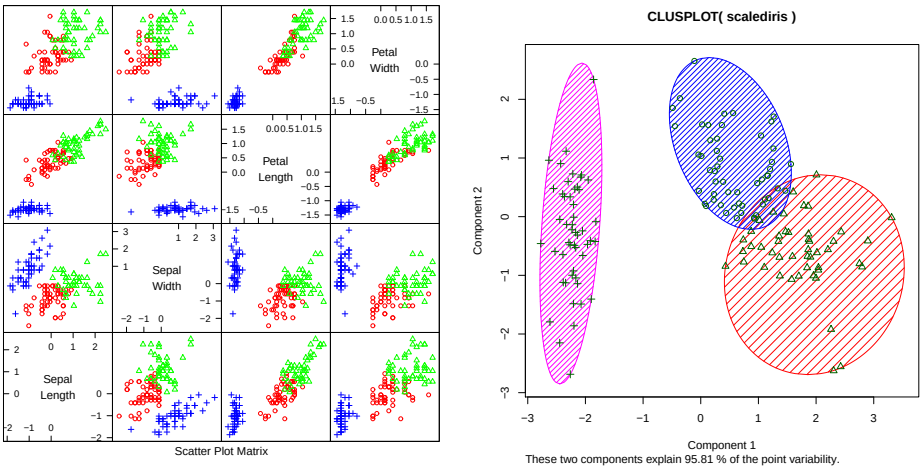


FIGURE 6.6: *On the* **left**, *a panel plot of the iris data clustered using k-means with* $k = 3$. *By comparison with figure 6.8, notice how the clusters appear to follow the species labels. On the* **right**, *this data set projected onto the first two principal components, with one blob drawn over each cluster.*

In some special cases (like the iris example), we might know the right answer to check our clustering against. In such cases, one can evaluate the clustering by looking at the number of different labels in a cluster (sometimes called the purity), and the number of clusters. A good solution will have few clusters, all of which have high purity. Mostly, we don't have a right answer to check against. An alternative strategy, which might seem crude to you, for choosing k is extremely important in practice. Usually, one clusters data to use the clusters in an application (one of the most important, vector quantization, is described in section 6.5). There are usually natural ways to evaluate this application. For example, vector quantization is often used as an early step in texture recognition or in image matching; here one can evaluate the error rate of the recognizer, or the accuracy of the image matcher. One then chooses the k that gets the best evaluation score on validation data. In this view, the issue is not how good the clustering is; it's how well the system that uses the clustering works.

6.4.2 Soft Assignment

One difficulty with k-means is that each point must belong to exactly one cluster. But, given we don't know how many clusters there are, this seems wrong. If a point is close to more than one cluster, why should it be forced to choose? This reasoning suggests we assign points to cluster centers with weights.

We allow each point to carry a total weight of 1. In the conventional k-means algorithm, it must choose a single cluster, and assign its weight to that cluster alone. In soft-assignment k-means, the point can allocate some weight to each cluster center, as long as (a) the weights are all non-negative and (b) the weights sum to one. Write $w_{i,j}$ for the weight connecting point i to cluster center j. We

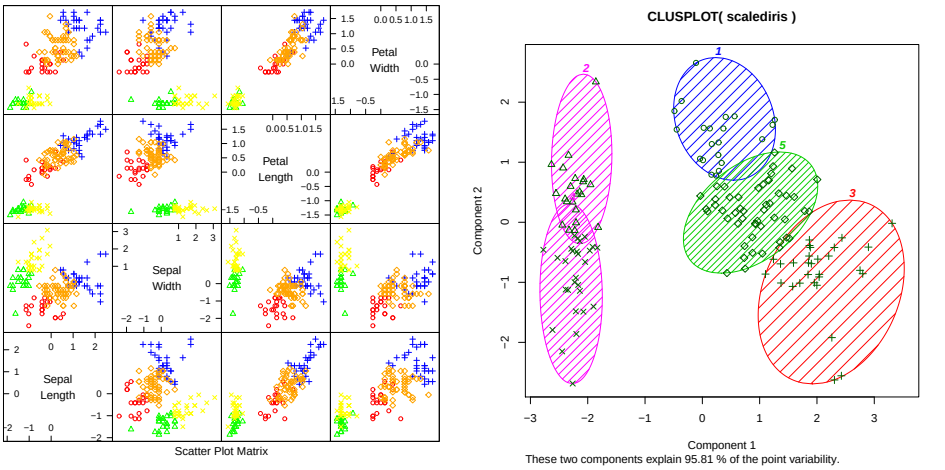


FIGURE 6.7: *On the* **left**, *a panel plot of the iris data clustered using k-means with* $k = 5$. *By comparison with figure 6.8, notice how* setosa *seems to have been broken in two groups, and* versicolor *and* verginica *into a total of three . On the* **right**, *this data set projected onto the first two principal components, with one blob drawn over each cluster.*

interpret these weights as the degree to which the point participates in a particular cluster. We require $w_{i,j} \geq 0$ and $\sum_j w_{i,j} = 1$.

We would like $w_{i,j}$ to be large when $\mathbf{x}_i$ is close to $\mathbf{c}_j$, and small otherwise. Write $d_{i,j}$ for the distance $\|\mathbf{x}_i - \mathbf{c}_j\|$ between these two. Write

$$s_{i,j} = e^{\frac{-d_{i,j}^2}{2\sigma^2}}$$

where $\sigma > 0$ is a choice of scaling parameter. This is often called the affinity between the point i and the center j. Now a natural choice of weights is

$$w_{i,j} = \frac{s_{i,j}}{\sum_{l=1}^{k} s_{i,l}}.$$

All these weights are non-negative, they sum to one, and the weight is large if the point is much closer to one center than to any other. The scaling parameter σ sets the meaning of "much closer" — we measure distance in units of σ.

Once we have weights, re-estimating the cluster centers is easy. We use a weights to compute a weighted average of the points. In particular, we re-estimate the j'th cluster center by

$$\frac{\sum_i w_{i,j}\mathbf{x}_i}{\sum_i w_{i,j}}.$$

Notice that k-means is a special case of this algorithm where σ limits to zero. In this case, each point has a weight of one for some cluster, and zero for all others, and the weighted mean becomes an ordinary mean. I have collected the description into a box (procedure 6.4) for convenience.

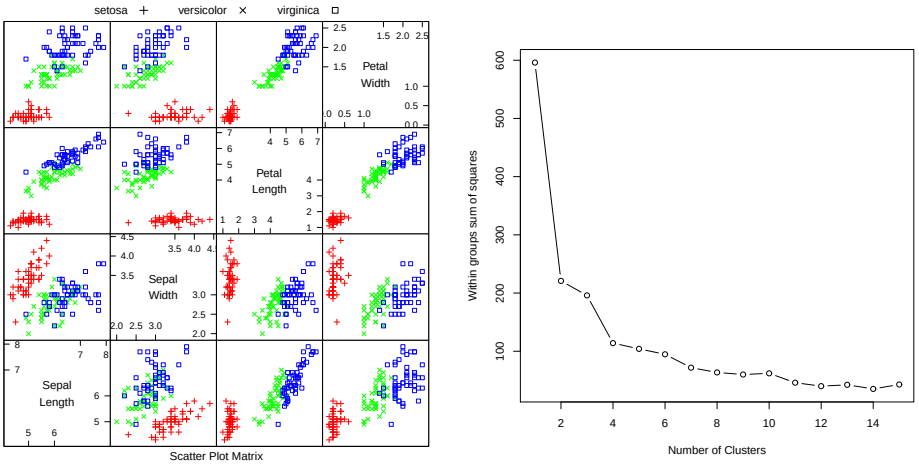


FIGURE 6.8: *On the* **left**, *the scatterplot matrix for the Iris data, for reference. On the* **right**, *a plot of the value of the cost function for each of several different values of k. Notice how there is a sharp drop in cost going from $k = 1$ to $k = 2$, and again at $k = 4$; after that, the cost falls off slowly. This suggests using $k = 2$, $k = 3$, or $k = 4$, depending on the precise application.*

Procedure: 6.4 *K-Means with Soft Weights*

Choose k. Choose k data points $\mathbf{c}_j$ to act as initial cluster centers. Until the cluster centers change very little:

- First, we estimate the weights. For each pair of a data point $\mathbf{x}_i$ and a cluster $\mathbf{c}_j$, compute the affinity

$$s_{i,j} = e^{\frac{-\|\mathbf{x}_i - \mathbf{c}_j\|}{2\sigma^2}}.$$

- Now for each pair of a data point $\mathbf{x}_i$ and a cluster $\mathbf{c}_j$ compute the soft weight linking the data point to the center

$$w_{i,j} = s_{i,j} / \sum_{l=1} k s_{i,l}.$$

- For each cluster, compute a new center

$$\mathbf{c}_j = \frac{\sum_i w_{i,j}\mathbf{x}_i}{\sum_i w_{i,j}}$$

6.4.3 General Comments on K-Means

If you experiment with k-means, you will notice one irritating habit of the algorithm. It almost always produces either some rather spread out clusters, or some single element clusters. Most clusters are usually rather tight and blobby clusters, but there is usually one or more bad cluster. This is fairly easily explained. Because every data point must belong to some cluster, data points that are far from all others (a) belong to some cluster and (b) very likely "drag" the cluster center into a poor location. This applies even if you use soft assignment, because every point must have total weight one. If the point is far from all others, then it will be assigned to the closest with a weight very close to one, and so may drag it into a poor location, or it will be in a cluster on its own.

There are ways to deal with this. If k is very big, the problem is often not significant, because then you simply have many single element clusters that you can ignore. It isn't always a good idea to have too large a k, because then some larger clusters might break up. An alternative is to have a junk cluster. Any point that is too far from the closest true cluster center is assigned to the junk cluster, and the center of the junk cluster is not estimated. Notice that points should not be assigned to the junk cluster permanently; they should be able to move in and out of the junk cluster as the cluster centers move.

6.4.4 K-Mediods

In some cases, we want to cluster objects that can't be averaged. For example, you can compute distances between two trees but you can't meaningfully average two trees. One case where this happens is when you have a table of distances between objects, but do not know vectors representing the objects. For example, you could collect data giving the distances between cities, without knowing where the cities are (as in Section 4.4.3, particularly Figure 4.16), then try and cluster using this data. As another example, you could collect data giving similarities between breakfast items as in Section 4.4.3, then turn the similarities into distances by taking the negative logarithm. This gives a useable table of distances. You still can't average kippers with oatmeal, so you couldn't use k-means to cluster this data. A variant of k-means, known as k-medoids, applies to this case.

In k-medoids, the cluster centers are data items rather than averages, and so are called "mediods". The rest of the algorithm has a familiar form. We assume k, the number of cluster centers, is known. We initialize the cluster centers by choosing examples at random. We then iterate two procedures. In the first, we allocate each data point to the closest mediod. In the second, we choose the best medoid for each cluster by finding the data point that minimizes the sum of distances of points in the cluster to that medoid. This point can be found by simply searching all points.

Remember this: *K-means clustering is the "go-to" clustering algorithm. You should see it as a basic recipe from which many algorithms can be concocted. The recipe is: iterate: allocate each data point to the closest cluster center; re-estimate cluster centers from their data points. There are many variations, improvements, etc. that are possible on this recipe. We have seen soft weights and k-mediods. K-means is not usually best implemented with the method I described (which isn't particularly efficient, but gets to the heart of what is going on). Implementations of k-means differ in important ways from my rather high-level description of the algorithm; for any but tiny problems, you should use a package, and you should look for a package that uses the Lloyd-Hartigan method.*

6.5 DESCRIBING REPETITION WITH VECTOR QUANTIZATION

Repetition is an important feature of many interesting signals. For example, images contain *textures*, which are orderly patterns that look like large numbers of small structures that are repeated. Examples include the spots of animals such as leopards or cheetahs; the stripes of animals such as tigers or zebras; the patterns on bark, wood, and skin. Similarly, speech signals contain *phonemes* — characteristic, stylised sounds that people assemble together to produce speech (for example, the "ka" sound followed by the "tuh" sound leading to "cat"). Another example comes from accelerometers. If a subject wears an accelerometer while moving around, the signals record the accelerations during their movements. So, for example, brushing one's teeth involves a lot of repeated twisting movements at the wrist, and walking involves swinging the hand back and forth.

Repetition occurs in subtle forms. The essence is that a small number of local patterns can be used to represent a large number of examples. You see this effect in pictures of scenes. If you collect many pictures of, say, a beach scene, you will expect most to contain some waves, some sky, and some sand. The individual patches of wave, sky or sand can be surprisingly similar, and different images are made by selecting some patches from a vocabulary of patches, then placing them down to form an image. Similarly, pictures of living rooms contain chair patches, TV patches, and carpet patches. Many different living rooms can be made from small vocabularies of patches; but you won't often see wave patches in living rooms, or carpet patches in beach scenes.

An important part of representing signals that repeat is building a vocabulary of patterns that repeat, then describing the signal in terms of those patterns. For many problems, problems, knowing what vocabulary elements appear and how often is much more important than knowing where they appear. For example, if you want to tell the difference between zebras and leopards, you need to know whether stripes or spots are more common, but you don't particularly need to know where they appear. As another example, if you want to tell the difference between brushing teeth and walking using accelerometer signals, knowing that there are lots of (or

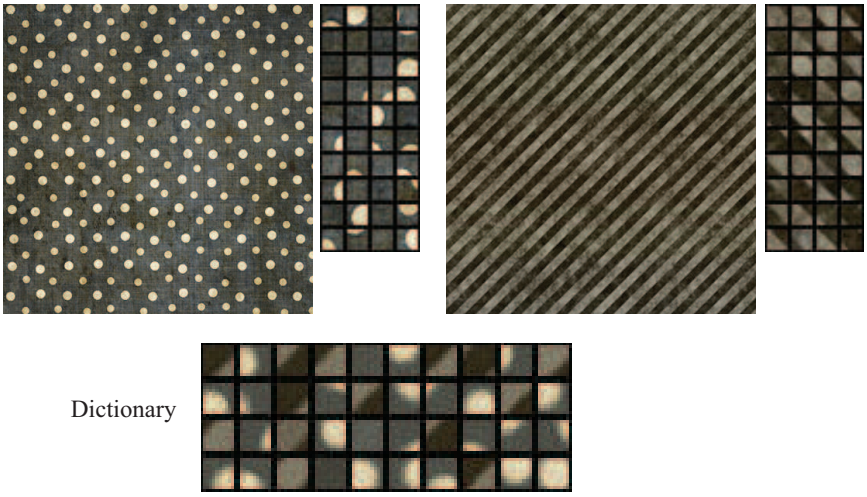


FIGURE 6.9: *It is hard to hide repetition in a signal, meaning that you don't need to be careful about whether patches overlap, etc. when vector quantizing a signal.* **Top:** *two images with rather exaggerated repetion, published on flickr.com with a creative commons license by* `webtreats`. *Next to these images, I have placed zoomed sampled 10x10 patches from those images; although the spots (resp. stripes) aren't necessarily centered in the patches, it's pretty clear which image each patch comes from.* **Bottom:** *a 40 patch dictionary computed using k-means from 4000 samples from each image. If you look closely, you'll see that some dictionary entries are clearly stripe entries, others clearly spot entries. Stripe images will have patches represented by stripe entries, spot images by spot entries; so the histogram construction in the text should make differences apparent.*

few) twisting movements is important, but knowing how the movements are linked together in time may not be.

6.5.1 Vector Quantization

It is natural to try and find patterns by looking for small pieces of signal of fixed size that appear often. In an image, a piece of signal might be a 10x10 patch; in a sound file, which is likely represented as a vector, it might be a subvector of fixed size. A 3-axis accelerometer signal is represented as a $3 \times r$ dimensional array (where r is the number of samples); in this case, a piece might be a 3×10 subarray. But finding patterns that appear often is hard to do, because the signal is continuous — each pattern will be slightly different, so we cannot simply count how many times a particular pattern occurs.

Here is a strategy. First, we take a training set of signals, and cut each signal into pieces of fixed size. We could use d dimensional vectors for a sound file; $d \times d$ dimensional patches for an image; or $3 \times d$ dimensional subarrays for an accelerometer signal. In each case, it is easy to compute the distance between two pieces: you use the sum of squared distances. It doesn't seem to matter too much

if these pieces overlap or not. We then build a set of clusters out of these pieces. This set of clusters is often thought of as a dictionary. We can now now describe any new piece with the cluster center closest to that piece. This means that a piece of signal is described with a number in the range $[1, \ldots, k]$ (where you get to choose k), and two pieces that are close should be described by the same number. This strategy is known as **vector quantization**.

Procedure: 6.5 *Vector Quantization - Building a Dictionary*

Take a training set of signals, and cut each signal into pieces of fixed size. The size of the piece will affect how well your method works, and is usually chosen by experiment. It does not seem to matter much if the pieces overlap. Cluster all the example pieces, and record the k cluster centers. It is usual, but not required, to use k-means clustering.

We can now build features that represent important repeated structure in signals. We take a signal, and cut it up into vectors of length d. These might overlap, or be disjoint; we follow whatever strategy we used in building the dictionary. We then take each vector, and compute the number that describes it (i.e. the number of the closest cluster center, as above). We then compute a histogram of the numbers we obtained for all the vectors in the signal. This histogram describes the signal.

Procedure: 6.6 *Vector Quantization - Representing a Signal*

Take your signal, and cut it into pieces of fixed size. The size of the piece will affect how well your method works, and is usually chosen by experiment. It does not seem to matter much if the pieces overlap. For each piece, record the closest cluster center in the dictionary. Represent the signal with a histogram of these numbers, which will be a k dimensional vector.

Notice several nice features to this construction. First, it can be applied to anything that can be thought of in terms of fixed size pieces, so it will work for speech signals, sound signals, accelerometer signals, images, and so on. Another nice feature is the construction can accept signals of different length, and produce a description of fixed length. One accelerometer signal might cover 100 time intervals; another might cover 200; but the description is always a histogram with k buckets, so it's always a vector of length k.

Yet another nice feature is that we don't need to be all that careful how we cut the signal into fixed length vectors. This is because it is hard to hide repetition. This point is easier to make with a figure than in text, so look at figure **??**.

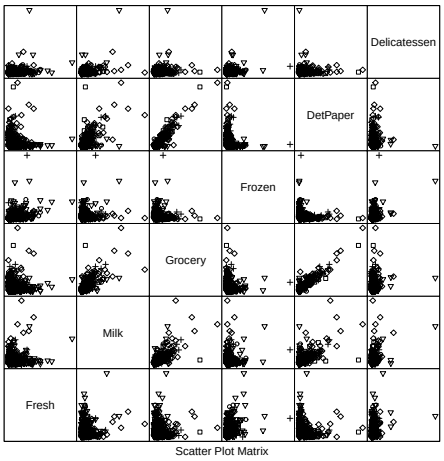


FIGURE 6.10: *A panel plot of the wholesale customer data of http://archive.ics.uci.edu/ml/datasets/Wholesale+customers, which records sums of money spent annually on different commodities by customers in Portugal. This data is recorded for six different groups (two channels each within three regions). I have plotted each group with a different marker, but you can't really see much structure here, for reasons explained in the text.*

6.5.2 Example: Groceries in Portugal

At http://archive.ics.uci.edu/ml/datasets/Wholesale+customers, you will find a dataset giving sums of money spent annually on different commodities by customers in Portugal. The commodities are divided into a set of categories (fresh; milk; grocery; frozen; detergents and paper; and delicatessen) relevant for the study. These customers are divided by channel (two channels) and by region (three regions). You can think of the data as being divided into six groups (one for each pair of channel and region). There are 440 customer records, and there are many customers in each group. Figure 6.10 shows a panel plot of the customer data; the data has been clustered, and I gave each of 20 clusters its own marker.

People tend to like to live near people who are "like" them, so you could expect people in a region to be somewhat similar; you could reasonably expect differences between regions (regional preferences; differences in wealth; and so on). Retailers have different channels to appeal to different people, so you could expect people using different channels to be different. But you don't see this in the plot of clusters. In fact, the plot doesn't really show much structure.

Here is a way to think about structure in the data. There are likely to be different "types" of customer. For example, customers who prepare food at home might spend more money on fresh or on grocery, and those who mainly buy prepared food might spend more money on delicatessan; similarly, coffee drinkers with cats or with children might spend more on milk than the lactose-intolerant, and so on. So we can expect customers to cluster in types. An effect like this is hard to see on a panel plot (Figure 6.10). The plot for this dataset is hard to read, because the

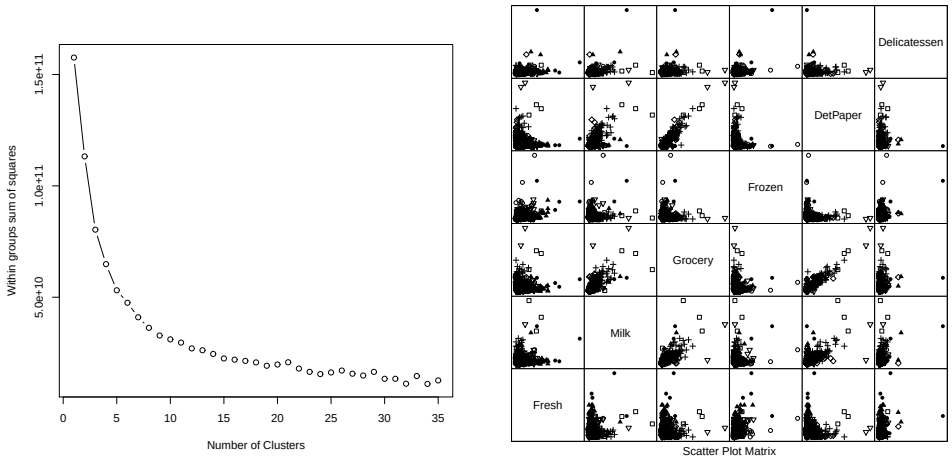


FIGURE 6.11: *On the* **left**, *the cost function (of section 6.4) for clusterings of the customer data with k-means for k running from 2 to 35. This suggests using a k somewhere in the range 10-30; I chose 20. On the* **right**, *I have clustered this data to 20 cluster centers with k-means. The clusters do seem to be squashed together, but the plot on the left suggests that clusters do capture some important information. Using too few clusters will clearly lead to problems. Notice that I did not scale the data, because each of the measurements is in a comparable unit. For example, it wouldn't make sense to scale expenditures on fresh and expenditures on grocery with a different scale.*

dimension is fairly high for a panel plot and the data is squashed together in the bottom left corner. However, you can see the effect when you cluster the data and look at the cost function in representing the data with different values of k — quite a small set of clusters gives quite a good representation (Figure 6.12). The panel plot of cluster membership (also in that figure) isn't particularly informative. The dimension is quite high, and clusters get squashed together.

There is another effect, which isn't apparent in these plots. Some of what cause customers to cluster in types are driven by things like wealth and the tendency of people to have neighbors who are similar to them. This means that different groups should have different fractions of each type of customer. There might be more deli-spenders in wealthier regions; more milk-spenders and detergent-spenders in regions where it is customary to have many children; and so on. This sort of structure will not be apparent in a panel plot. A group of a few milk-spenders and many detergent-spenders will have a few data points with high milk expenditure values (and low other values) and also many data points with high detergent expenditure values (and low other values). In a panel plot, this will look like two blobs; but if there is a second group with many milk-spenders and few detergent-spenders will also look like two blobs, lying roughly on top of the first set of blobs. It will be hard to spot the difference between the groups.

An easy way to see this difference is to look at histograms of the types of

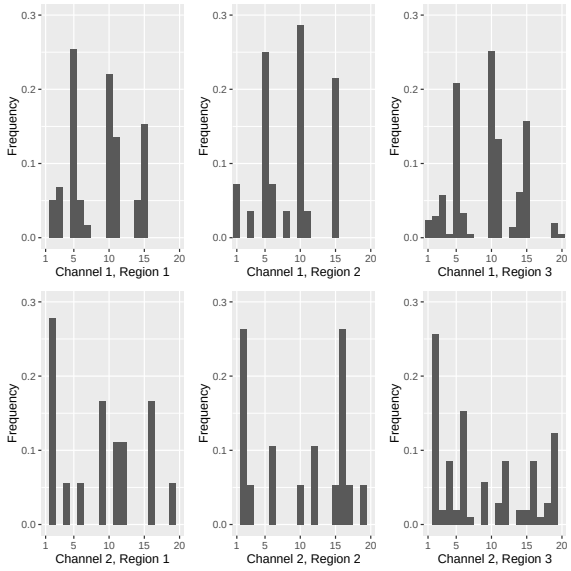


FIGURE 6.12: *The histogram of different types of customer, by group, for the customer data. Notice how the distinction between the groups is now apparent — the groups do appear to contain quite different distributions of customer type. It looks as though the channels (rows in this figure) are more different than the regions (columns in this figure).*

customer within each group. I described the each group of data by the histogram of customer types that appeared in that group (Figure **??**). Notice how the distinction between the groups is now apparent — the groups do appear to contain quite different distributions of customer type. It looks as though the channels (rows in this figure) are more different than the regions (columns in this figure). To be more confident in this analysis, we would need to be sure that different types of customer really are different. We could do this by repeating the analysis for fewer clusters, or by looking at the similarity of customer types.

6.5.3 Efficient Clustering and Hierarchical K Means

One important difficulty occurs in applications. We might need to have an enormous dataset (millions of image patches are a real possibility), and so a very large k. In this case, k means clustering becomes difficult because identifying which cluster center is closest to a particular data point scales linearly with k (and we have to do this for every data point at every iteration). There are two useful strategies for dealing with this problem.

The first is to notice that, if we can be reasonably confident that each cluster contains many data points, some of the data is redundant. We could randomly subsample the data, cluster that, then keep the cluster centers. This works, but doesn't scale particularly well.

A more effective strategy is to build a hierarchy of k-means clusters. We

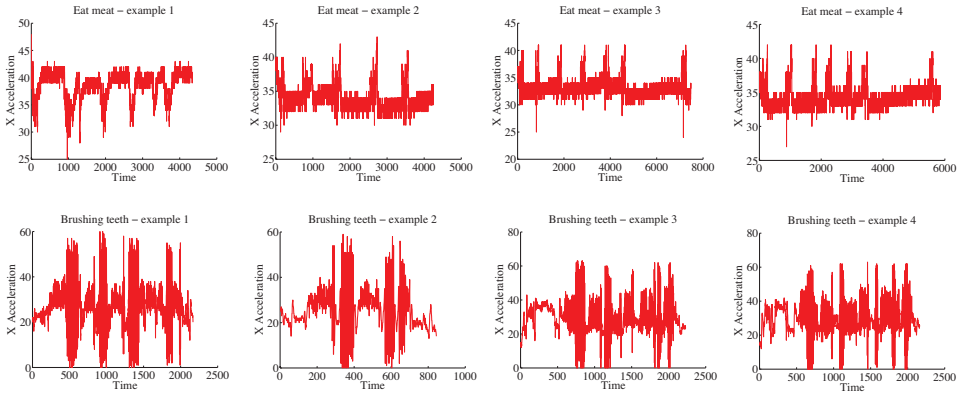


FIGURE 6.13: *Some examples from the accelerometer dataset at https://archive.ics.uci.edu/ml/datasets/Dataset+for+ADL+Recognition+with+Wrist-worn+Accelerometer. I have labelled each signal by the activity. These show acceleration in the X direction (Y and Z are in the dataset, too). There are four examples for* **brushing teeth** *and four for* **eat meat**. *You should notice that the examples don't have the same length in time (some are slower and some faster eaters, etc.), but that there seem to be characteristic features that are shared within a category (brushing teeth seems to involve faster movements than eating meet).*

randomly subsample the data (typically, quite aggressively), then cluster this with a small value of k. Each data item is then allocated to the closest cluster center, and the data in each cluster is clustered again with k-means. We now have something that looks like a two-level tree of clusters. Of course, this process can be repeated to produce a multi-level tree of clusters. It is easy to use this tree to vector quantize a query data item. We vector quantize at the first level. Doing so chooses a branch of the tree, and we pass the data item to this branch. It is either a leaf, in which case we report the number of the leaf, or it is a set of clusters, in which case we vector quantize, and pass the data item down. This procedure is efficient both when one clusters and at run time.

6.5.4 Example: Activity from Accelerometer Data

A complex example dataset appears at https://archive.ics.uci.edu/ml/datasets/Dataset+for+ADL+Recognition+w
This dataset consists of examples of the signal from a wrist mounted accelerometer, produced as different subjects engaged in different activities of daily life. Activities include: brushing teeth, climbing stairs, combing hair, descending stairs, and so on. Each is performed by sixteen volunteers. The accelerometer samples the data at 32Hz (i.e. this data samples and reports the acceleration 32 times per second). The accelerations are in the x, y and z-directions. Figure 6.13 shows the x-component of various examples of toothbrushing.

There is an important problem with using data like this. Different subjects take quite different amounts of time to perform these activities. For example, some

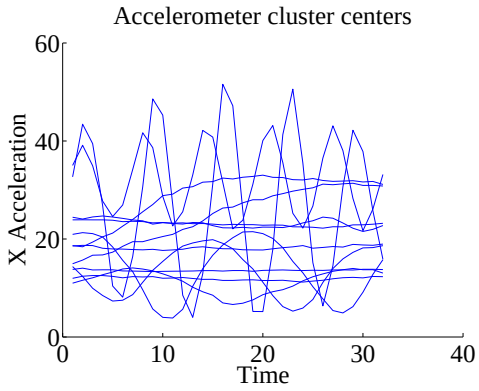


FIGURE 6.14: *Some cluster centers from the accelerometer dataset. Each cluster center represents a one-second burst of activity. There are a total of 480 in my model, which I built using hierarchical k-means. Notice there are a couple of centers that appear to represent movement at about 5Hz; another few that represent movement at about 2Hz; some that look like 0.5Hz movement; and some that seem to represent much lower frequency movement. These cluster centers are samples (rather than chosen to have this property).*

subjects might be more thorough tooth-brushers than other subjects. As another example, people with longer legs walk at somewhat different frequencies than people with shorter legs. This means that the same activity performed by different subjects will produce data vectors *that are of different lengths*. It's not a good idea to deal with this by warping time and resampling the signal. For example, doing so will make a thorough toothbrusher look as though they are moving their hands very fast (or a careless toothbrusher look ludicrously slow: think speeding up or slowing down a movie). So we need a representation that can cope with signals that are a bit longer or shorter than other signals.

Another important property of these signals is that all examples of a particular activity should contain repeated patterns. For example, brushing teeth should show fast accelerations up and down; walking should show a strong signal at somewhere around 2 Hz; and so on. These two points should suggest vector quantization to you. Representing the signal in terms of stylized, repeated structures is probably a good idea because the signals probably contain these structures. And if we represent the signal in terms of the relative frequency with which these structures occur, the representation will have a fixed length, even if the signal doesn't. To do so, we need to consider (a) over what time scale we will see these repeated structures and (b) how to ensure we segment the signal into pieces so that we see these structures.

Generally, repetition in activity signals is so obvious that we don't need to be smart about segment boundaries. I broke these signals into 32 sample segments, one following the other. Each segment represents one second of activity. This is long enough for the body to do something interesting, but not so long that our representation will suffer if we put the segment boundaries in the wrong place. This resulted in about 40, 000 segments. I then used hierarchical k-means to cluster these

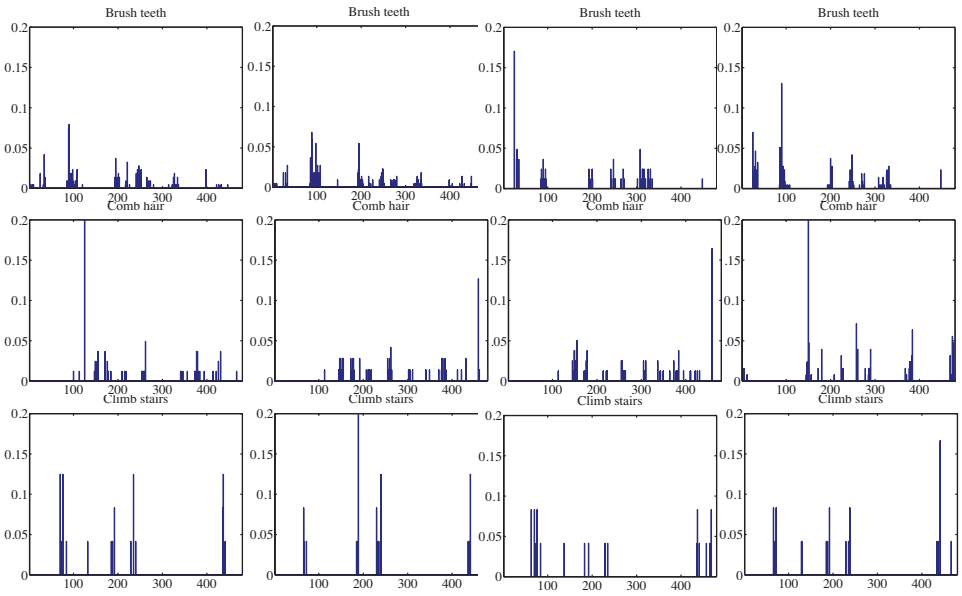


FIGURE 6.15: *Histograms of cluster centers for the accelerometer dataset, for different activities. You should notice that (a) these histograms look somewhat similar for different actors performing the same activity and (b) these histograms look somewhat different for different activities.*

segments. I used two levels, with 40 cluster centers at the first level, and 12 at the second. Figure 6.14 shows some cluster centers at the second level.

I then computed histogram representations for different example signals (Figure 6.15). You should notice that when the activity label is different, the histogram looks different, too.

Another useful way to check this representation is to compare the average within class chi-squared distance with the average between class chi-squared distance. I computed the histogram for each example. Then, for each pair of examples, I computed the chi-squared distance between the pair. Finally, for each pair of *activity labels*, I computed the average distance between pairs of examples where one example has one of the activity labels and the other example has the other activity label. In the ideal case, all the examples with the same label would be very close to one another, and all examples with different labels would be rather different. Table 6.1 shows what happens with the real data. You should notice that for some pairs of activity label, the mean distance between examples is smaller than one would hope for (perhaps some pairs of examples are quite close?). But generally, examples of activities with different labels tend to be further apart than examples of activities with the same label.

Yet another way to check the representation is to try classification with nearest neighbors, using the chi-squared distance to compute distances. I split the dataset into 80 test pairs and 360 training pairs; using 1-nearest neighbors, I was able to

0.9	2.0	1.9	2.0	2.0	2.0	1.9	2.0	1.9	1.9	2.0	2.0	2.0	2.0
	1.6	2.0	1.8	2.0	2.0	2.0	1.9	1.9	2.0	1.9	1.9	2.0	1.7
		1.5	2.0	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.9	2.0
			1.4	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	1.8
				1.5	1.8	1.7	1.9	1.9	1.8	1.9	1.9	1.8	2.0
					0.9	1.7	1.9	1.9	1.8	1.9	1.9	1.9	2.0
						0.3	1.9	1.9	1.5	1.9	1.9	1.9	2.0
							1.8	1.8	1.9	1.9	1.9	1.9	1.9
								1.7	1.9	1.9	1.9	1.9	1.9
									1.6	1.9	1.9	1.9	2.0
										1.8	1.9	1.9	1.9
											1.8	2.0	1.9
												1.5	2.0
													1.5

TABLE 6.1: *Each column of the table represents an activity for the activity dataset* https://archive.ics.uci.edu/ml/datasets/Dataset+for+ADL+Recognition+with+Wrist-worn+Accelerometer, *as does each row. In each of the upper diagonal cells, I have placed the average chi-squared distance between histograms of examples from that pair of classes (I dropped the lower diagonal for clarity). Notice that in general the diagonal terms (average within class distance) are rather smaller than the off diagonal terms. This quite strongly suggests we can use these histograms to classify examples successfully.*

get a held-out error rate of 0.79. This suggests that the representation is fairly good at exposing what is important.

6.6 YOU SHOULD

6.6.1 remember these definitions:

6.6.2 remember these terms:

6.6.3 remember these facts:

6.6.4 remember these procedures:

PROGRAMMING EXERCISES

6.1. You can find a dataset dealing with European employment in 1979 at http://dasl.datadesk.com/data/view/47. This dataset gives the percentage of people employed in each of a set of areas in 1979 for each of a set of European countries.

- **(a)** Use an agglomerative clusterer to cluster this data. Produce a dendrogram of this data for each of single link, complete link, and group average clustering. You should label the countries on the axis. What structure in the data does each method expose? it's fine to look for code, rather than writing your own. **Hint:** I made plots I liked a lot using R's `hclust` clustering function, and then turning the result into a phylogenetic tree and using a fan plot, a trick I found on the web; try `plot(as.phylo(hclustresult), type=''fan'')`. You should see dendrograms that "make sense" (at least if you remember some European history), and have interesting differences.
- **(b)** Using k-means, cluster this dataset. What is a good choice of k for this data and why?

6.2. Obtain the activities of daily life dataset from the UC Irvine machine learning website (https://archive.ics.uci.edu/ml/datasets/Dataset+for+ADL+Recognition+with+Wrist-worn+Acceleron data provided by Barbara Bruno, Fulvio Mastrogiovanni and Antonio Sgorbissa).

- **(a)** Build a classifier that classifies sequences into one of the 14 activities provided. To make features, you should vector quantize, then use a histogram of cluster centers (as described in the subsection; this gives a pretty explicit set of steps to follow). You will find it helpful to use hierarchical k-means to vector quantize. You may use whatever multi-class classifier you wish, though I'd start with R's decision forest, because it's easy to use and effective. You should report (a) the total error rate and (b) the class confusion matrix of your classifier.
- **(b)** Now see if you can improve your classifier by (a) modifying the number of cluster centers in your hierarchical k-means and (b) modifying the size of the fixed length samples that you use.

6.3. This is a fairly ambitious exercise. It will demonstrate how to use vector quantization to handle extremely sparse data. The 20 newsgroups dataset is a famous text dataset. It consists of posts collected from 20 different newsgroups. There are a variety of tricky data issues that this presents (for example, what aspects of the header should one ignore? should one reduce words to their stems, so "winning" goes to "win", "hugely" to "huge", and so on?). We will ignore these issues, and deal with a cleaned up version of the dataset. This consists of three items each for train and test: a document-word matrix, a set of labels, and a map. You can find this cleaned up version of the dataset at http://qwone.com/~jason/20Newsgroups/. You should look for the cleaned up version, identified as `20news-bydate-matlab.tgz` on that page. The usual task is to label a test article with which newsgroup it came from. Instead, we will assume you have a set of test articles, all from the same newsgroup, and you need to identify the newsgroup. The document-word matrix is a table of counts of how many times a particular word appears in a particular document. The collection of words is very large (53975 distinct words), and most words do not appear in most documents, so most entries of this matrix are zero. The file `train.data` contains this matrix for a collection of training data; each row represents a distinct document (there are 11269), and each column represents

a distinct word.

(a) Cluster the rows of this matrix to get a set of cluster centers using k-means. You should have about one center for every 10 documents. Use k-means, and you should find an efficient package rather than using your own implementation. In particular, implementations of k-means differ in important ways from my rather high-level description of the algorithm; you should look for a package that uses the Lloyd-Hartigan method. **Hint:** Clustering all these points is a bit of a performance; check your code on small subsets of the data first, because the size of this dataset means that clustering the whole thing will be slow.

(b) You can now think of each cluster center as a document "type". For each newsgroup, plot a histogram of the "types" of document that appear in the training data for that newsgroup. You'll need to use the file `train.label`, which will tell you what newsgroup a particular item comes from.

(c) Now train a classifier that accepts a small set of documents (10-100) from a single newsgroup, and predicts which of 20 newsgroups it comes from. You should use the histogram of types from the previous sub-exercise as a feature vector. Compute the performance of this classifier on the test data (`test.data` and `test.label`).

6.4. This is another fairly ambitious exercise. We will use the document clustering method of section **??** to identify clusters of documents, which we will associate with topics. The 20 newsgroups dataset is a famous text dataset. It consists of posts collected from 20 different newsgroups. There are a variety of tricky data issues that this presents (for example, what aspects of the header should one ignore? should one reduce words to their stems, so "winning" goes to "win", "hugely" to "huge", and so on?). We will ignore these issues, and deal with a cleaned up version of the dataset. This consists of three items each for train and test: a document-word matrix, a set of labels, and a map. You can find this cleaned up version of the dataset at http://qwone.com/~jason/20Newsgroups/. You should look for the cleaned up version, identified as `20news-bydate-matlab.tgz` on that page. The usual task is to label a test article with which newsgroup it came from. The document-word matrix is a table of counts of how many times a particular word appears in a particular document. The collection of words is very large (53975 distinct words), and most words do not appear in most documents, so most entries of this matrix are zero. The file `train.data` contains this matrix for a collection of training data; each row represents a distinct document (there are 11269), and each column represents a distinct word.

(a) Cluster the rows of this matrix, using the method of section **??**, to get a set of cluster centers which we will identify as topics. **Hint:** Clustering all these points is a bit of a performance; check your code on small subsets of the data first, because the size of this dataset means that clustering the whole thing will be slow.

(b) You can now think of each cluster center as a document "type". Assume you have k clusters (topics). Represent each document by a k-dimensional vector. Each entry of the vector should be the negative log-probability of the document under that cluster model. Now use this information to build a classifier that identifies the newsgroup using the vector. You'll need to use the file `train.label`, which will tell you what newsgroup a particular item comes from. I advise you use a randomized decision forest, but other choices are plausible. Evaluate your classifier usingthe test data

(`test.data` and `test.label`).

CHAPTER 7

Regression

Classification tries to predict a class from a data item. **Regression** tries to predict a value. For example, we know the zip code of a house, the square footage of its lot, the number of rooms and the square footage of the house, and we wish to predict its likely sale price. As another example, we know the cost and condition of a trading card for sale, and we wish to predict a likely profit in buying it and then reselling it. As yet another example, we have a picture with some missing pixels – perhaps there was text covering them, and we want to replace it – and we want to fill in the missing values. As a final example, you can think of classification as a special case of regression, where we want to predict either $+1$ or -1; this isn't usually the best way to proceed, however. Predicting values is very useful, and so there are many examples like this.

7.1 OVERVIEW

Some formalities are helpful here. In the simplest case, we have a dataset consisting of a set of N pairs $(\mathbf{x}_i, y_i)$. We think of y_i as the value of some function evaluated at $\mathbf{x}_i$, but with some random component. This means there might be two data items where the $\mathbf{x}_i$ are the same, and the y_i are different. We refer to the $\mathbf{x}_i$ as **explanatory variables** and the y_i is a **dependent variable**. We regularly say that we are regressing the dependent variable against the explanatory variables. We want to use the examples we have — the **training examples** — to build a model of the dependence between y and $\mathbf{x}$. This model will be used to predict values of y for new values of $\mathbf{x}$, which are usually called **test examples**. By far the most important model has the form $y = \mathbf{x}^T\beta + \xi$, where β are some set of parameters we need to choose and ξ are random effects. Now imagine that we have one independent variable. An appropriate choice of $\mathbf{x}$ (details below) will mean that the predictions made by this model will lie on a straight line. Figure 7.1 shows two regressions. The data are plotted with a scatter plot, and the line gives the prediction of the model for each value on the x axis.

We do not guarantee that different values of $\mathbf{x}$ produce different values of y. Data just isn't like this (see the crickets example Figure 7.1). Traditionally, regression produces some representation of a probability distribution for y conditioned on $\mathbf{x}$, so that we would get (say) some representation of a distribution on the houses likely sale value. The best prediction would then be the expected value of that distribution.

It should be clear that none of this will work if there is not some relationship between the training examples and the test examples. If I collect training data on the height and weight of children, I'm unlikely to get good predictions of the weight of adults from their height. We can be more precise with a probabilistic framework. We think of $\mathbf{x}_i$ as IID samples from some (usually unknown) probability distribution $P(X)$. Then the test examples should also be IID samples from $P(X)$,

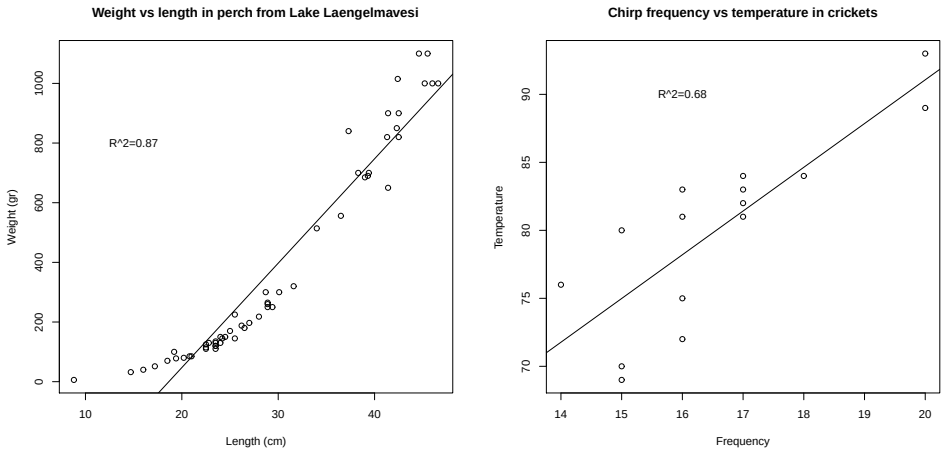


FIGURE 7.1: *On the* **left**, *a regression of weight against length for perch from a Finnish lake (you can find this dataset, and the back story at* http://www.amstat.org/publications/jse/jse_data_archive.htm*; look for "fishcatch" on that page). Notice that the linear regression fits the data fairly well, meaning that you should be able to predict the weight of a perch from its length fairly well. On the* **right**, *a regression of air temperature against chirp frequency for crickets. The data is fairly close to the line, meaning that you should be able to tell the temperature from the pitch of cricket's chirp fairly well. This data is from* http://mste.illinois.edu/patel/amar430/keyprob1.html*. The* R^2 *you see on each figure is a measure of the goodness of fit of the regression (section 7.2.4).*

or, at least, rather like them – you usually can't check this point with any certainty. A probabilistic formalism can help be precise about the y_i, too. Assume another random variable Y has joint distribution with X given by $P(Y, X)$. We think of each y_i as a sample from $P(Y|\{X = \mathbf{x}_i\})$. Then our modelling problem would be: given the training data, build a model that takes a test example $\mathbf{x}$ and yields a model of $P(Y|\{X = \mathbf{x}_i\})$.

Thinking about the problem this way should make it clear that we're not relying on any exact, physical, or causal relationship between Y and X. It's enough that their joint probability makes useful predictions possible, something we will test by experiment. This means that you can build regressions that work in somewhat surprising circumstances. For example, regressing childrens' reading ability against their foot size can be quite successful. This isn't because having big feet somehow helps you read; it's because on the whole, older children read better, and also have bigger feet.

To do anything useful with this formalism requires some aggressive simplifying assumptions. There are very few circumstances that require a comprehensive representation of $P(Y|\{X = \mathbf{x}_i\})$. Usually, we are interested in $\mathbb{E}[Y|\{X = \mathbf{x}_i\}]$ (the mean of $P(Y|\{X = \mathbf{x}_i\})$) and in $\mathsf{var}(\{P(Y|\{X = \mathbf{x}_i\})\})$. To recover this representation, we assume that, for any pair of examples $(\mathbf{x}, y)$, the value of y is obtained

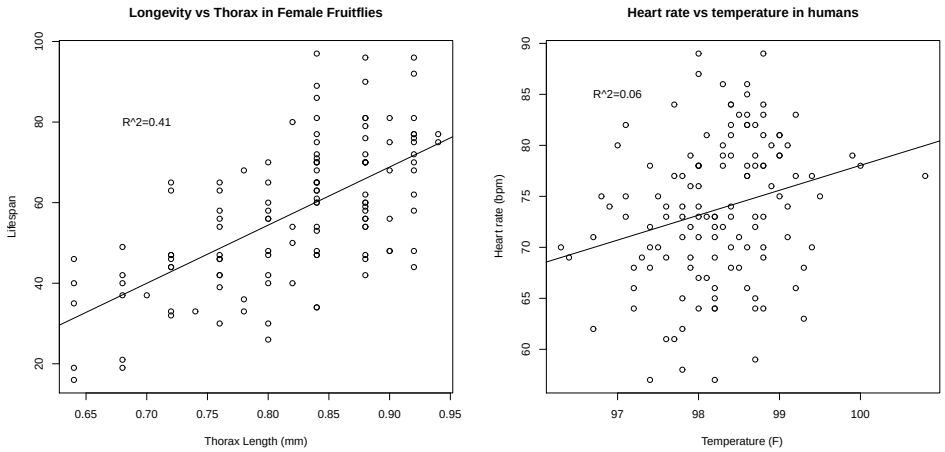


FIGURE 7.2: *Regressions do not necessarily yield good predictions or good model fits. On the* **left**, *a regression of the lifespan of female fruitflies against the length of their torso as adults (apparently, this doesn't change as a fruitfly ages; you can find this dataset, and the back story at http://www.amstat.org/publications/jse/jse_data_archive.htm; look for "fruitfly" on that page). The figure suggests you can make some prediction of how long your fruitfly will last by measuring its torso, but not a particularly accurate one. On the* **right**, *a regression of heart rate against body temperature for adults. You can find the data at http://www.amstat.org/publications/jse/jse_data_archive.htm as well; look for "temperature" on that page. Notice that predicting heart rate from body temperature isn't going to work that well, either.*

by applying some (unknown) function f to $\mathbf{x}$, then adding some random variable ξ with zero mean. We can write $y(\mathbf{x}) = f(\mathbf{x}) + \xi$, though it's worth remembering that there can be many different values of y associated with a single $\mathbf{x}$. Now we must make some estimate of f — which yields $\mathbb{E}[Y|\{X = \mathbf{x}_i\}]$ — and estimate the variance of ξ. The variance of ξ might be constant, or might vary with $\mathbf{x}$.

7.1.1 Regression to Spot Trends

Regression isn't only used to predict values. Another reason to build a regression model is to compare trends in data. Doing so can make it clear what is really happening. Here is an example from Efron ("Computer-Intensive methods in statistical regression", B. Efron, SIAM Review, 1988). The table in the appendix shows some data from medical devices, which sit in the body and release a hormone. The data shows the amount of hormone currently in a device after it has spent some time in service, and the time the device spent in service. The data describes devices from three production lots (A, B, and C). Each device, from each lot, is supposed to have the same behavior. The important question is: Are the lots the same? The amount of hormone changes over time, so we can't just compare the amounts currently in each device. Instead, we need to determine the relationship between time in service

and hormone, and see if this relationship is different between batches. We can do so by regressing hormone against time.

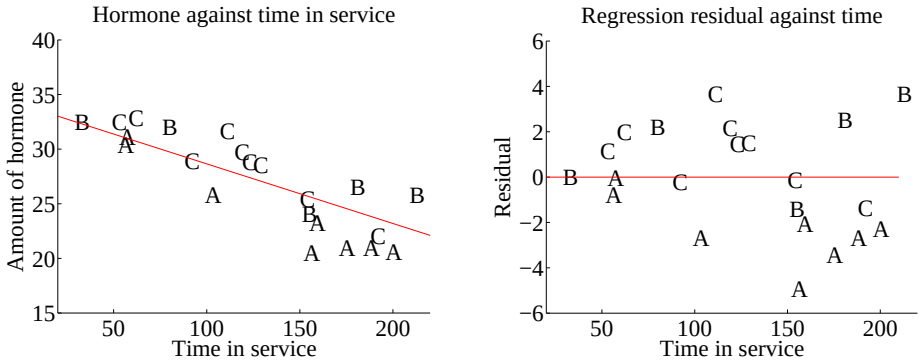


FIGURE 7.3: *On the* **left**, *a scatter plot of hormone against time for devices from tables 7.1 and 7.1. Notice that there is a pretty clear relationship between time and amount of hormone (the longer the device has been in service the less hormone there is). The issue now is to understand that relationship so that we can tell whether lots A, B and C are the same or different. The best fit line to all the data is shown as well, fitted using the methods of section 7.2. On the* **right**, *a scatter plot of residual — the distance between each data point and the best fit line — against time for the devices from tables 7.1 and 7.1. Now you should notice a clear difference; some devices from lots B and C have positive and some negative residuals, but all lot A devices have negative residuals. This means that, when we account for loss of hormone over time, lot A devices still have less hormone in them. This is pretty good evidence that there is a problem with this lot.*

Figure 7.3 shows how a regression can help. In this case, we have modelled the amount of hormone in the device as

$$a \times (\text{time in service}) + b$$

for a, b chosen to get the best fit (much more on this point later!). This means we can plot each data point on a scatter plot, together with the best fitting line. This plot allows us to ask whether any particular batch behaves differently from the overall model in any interesting way.

However, it is hard to evaluate the distances between data points and the best fitting line by eye. A sensible alternative is to subtract the amount of hormone predicted by the model from the amount that was measured. Doing so yields a **residual** — the difference between a measurement and a prediction. We can then plot those residuals (Figure 7.3). In this case, the plot suggests that lot A is special — all devices from this lot contain less hormone than our model predicts.

Definition: 7.1 *Regression*

Regression accepts a feature vector and produces a prediction, which is usually a number, but can sometimes have other forms. You can use these predictions as predictions, or to study trends in data. It is possible, but not usually particularly helpful, to see classification as a form of regression.

7.2 LINEAR REGRESSION AND LEAST SQUARES

Assume we have a dataset consisting of a set of N pairs $(\mathbf{x}_i, y_i)$. We think of y_i as the value of some function evaluated at $\mathbf{x}_i$, with some random component added. This means there might be two data items where the $\mathbf{x}_i$ are the same, and the y_i are different. We refer to the $\mathbf{x}_i$ as **explanatory variables** and the y_i is a **dependent variable**. We want to use the examples we have — the **training examples** — to build a model of the dependence between y and $\mathbf{x}$. This model will be used to predict values of y for new values of $\mathbf{x}$, which are usually called **test examples**. It can also be used to understand the relationships between the $\mathbf{x}$. The model needs to have some probabilistic component; we do not expect that y is a function of $\mathbf{x}$, and there is likely some error in evaluating y anyhow.

7.2.1 Linear Regression

We cannot expect that our model makes perfect predictions. Furthermore, y may not be a function of $\mathbf{x}$ — it is quite possible that the same value of $\mathbf{x}$ could lead to different y's. One way that this could occur is that y is a measurement (and so subject to some measurement noise). Another is that there is some randomness in y. For example, we expect that two houses with the same set of features (the $\mathbf{x}$) might still sell for different prices (the y's).

A good, simple model is to assume that the dependent variable (i.e. y) is obtained by evaluating a linear function of the explanatory variables (i.e. $\mathbf{x}$), then adding a zero-mean normal random variable. We can write this model as

$$y = \mathbf{x}^T\beta + \xi$$

where ξ represents random (or at least, unmodelled) effects. We will always assume that ξ has zero mean. In this expression, β is a vector of weights, which we must estimate. When we use this model to predict a value of y for a particular set of explanatory variables $\mathbf{x}^*$, we cannot predict the value that ξ will take. Our best available prediction is the mean value (which is zero). Notice that if $\mathbf{x} = 0$, the model predicts $y = 0$. This may seem like a problem to you — you might be concerned that we can fit only lines through the origin — but remember that $\mathbf{x}$ contains explanatory variables, and we can choose what appears in $\mathbf{x}$. The two examples show how a sensible choice of $\mathbf{x}$ allows us to fit a line with an arbitrary y-intercept.

Definition: 7.2 *Linear regression*

A linear regression takes the feature vector $\mathbf{x}$ and predicts $\mathbf{x}^T\beta$, for some vector of coefficients β. The coefficients are adjusted, using data, to produce the best predictions.

Example: 7.1 *A linear model fitted to a single explanatory variable*

Assume we fit a linear model to a single explanatory variable. Then the model has the form $y = x\beta + \xi$, where ξ is a zero mean random variable. For any value x^* of the explanatory variable, our best estimate of y is βx^*. In particular, if $x^* = 0$, the model predicts $y = 0$, which is unfortunate. We can draw the model by drawing a line through the origin with slope β in the x, y plane. The y-intercept of this line must be zero.

Example: 7.2 *A linear model with a non-zero y-intercept*

Assume we have a single explanatory variable, which we write u. We can then create a *vector* $\mathbf{x} = [u, 1]^T$ from the explanatory variable. We now fit a linear model to this vector. Then the model has the form $y = \mathbf{x}^T\beta + \xi$, where ξ is a zero mean random variable. For any value $\mathbf{x}^* = [u^*, 1]^T$ of the explanatory variable, our best estimate of y is $(\mathbf{x}^*)^T\beta$, which can be written as $y = \beta_1 u^* + \beta_2$. If $x^* = 0$, the model predicts $y = \beta_2$. We can draw the model by drawing a line through the origin with slope β_1 and y-intercept β_2 in the x, y plane.

7.2.2 Choosing β

We must determine β. We can proceed in two ways. I show both because different people find different lines of reasoning more compelling. Each will get us to the same solution. One is probabilistic, the other isn't. Generally, I'll proceed as if they're interchangeable, although at least in principle they're different.

Probabilistic approach: we could assume that ξ is a zero mean normal random variable with unknown variance. Then $P(y|x, \beta)$ is normal, with mean $\mathbf{x}^T\beta$, and so we can write out the log-likelihood of the data. Write σ^2 for the variance of ξ, which we don't know, but will not worry about right now. We have

that

$$\begin{aligned}\log \mathcal{L}(\beta) &= -\sum_i \log P(y_i|\mathbf{x}_i, \beta) \\ &= \frac{1}{2\sigma^2}\sum_i (y_i - \mathbf{x}_i^T\beta)^2 + \text{term not depending on } \beta\end{aligned}$$

Maximizing the log-likelihood of the data is equivalent to minimizing the negative log-likelihood of the data. Furthermore, the term $\frac{1}{2\sigma^2}$ does not affect the location of the minimum, so we must have that β minimizes $\sum_i (y_i - \mathbf{x}_i^T\beta)^2$, or anything proportional to it. It is helpful to minimize an expression that is an average of squared errors, because (hopefully) this doesn't grow much when we add data. We therefore minimize

$$\left(\frac{1}{N}\right)\left(\sum_i (y_i - \mathbf{x}_i^T\beta)^2\right).$$

Direct approach: notice that, if we have an estimate of β, we have an estimate of the values of the unmodelled effects ξ_i for each example. We just take $\xi_i = y_i - \mathbf{x}_i^T\beta$. It is quite natural to make the unmodelled effects "small". A good measure of size is the mean of the squared values, which means we want to minimize

$$\left(\frac{1}{N}\right)\left(\sum_i (y_i - \mathbf{x}_i^T\beta)^2\right).$$

We can write all this more conveniently using vectors and matrices. Write $\mathbf{y}$ for the vector

$$\begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix}$$

and $\mathcal{X}$ for the matrix

$$\begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \dots \mathbf{x}_n^T \end{pmatrix}.$$

Then we want to minimize

$$\left(\frac{1}{N}\right)(\mathbf{y} - \mathcal{X}\beta)^T(\mathbf{y} - \mathcal{X}\beta)$$

which means that we must have

$$\mathcal{X}^T\mathcal{X}\beta - \mathcal{X}^T\mathbf{y} = 0.$$

For reasonable choices of features, we could expect that $\mathcal{X}^T\mathcal{X}$ — which should strike you as being a lot like a covariance matrix — has full rank. If it does, which is the usual case, this equation is easy to solve. If it does not, there is more to do, which we will do in section 7.4.2.

Listing 7.1: R code used for the linear regression example of worked example 7.1

```
efd<-read.table('efrontable.txt',header=TRUE)
# the table has the form
#N1   Ah   Bh   Ch N2  At   Bt   Ct
# now we need to construct a new dataset
hor<-stack(efd, select=2:4)
tim<-stack(efd, select=6:8)
foo<-data.frame(time=tim[, c("values")],
                hormone=hor[, c("values")])
foo.lm<-lm(hormone~time,data=foo)
plot(foo)
abline(foo.lm)
```

Remember this: *The vector of coefficients β for a linear regression is usually estimated using a least-squares procedure.*

Worked example 7.1 *Simple Linear Regression with R*

Regress the hormone data against time for all the devices in the Efron example.

Solution: This example is mainly used to demonstrate how to regress in R. There is sample code in listing 7.1. The summary in the listing produces a great deal of information (try it). Most of it won't mean anything to you yet. You can get a figure by doing `plot(foo.lm)`, but these figures will not mean anything yet, either. In the code, I've shown how to plot the data and a line on top of it.

7.2.3 Residuals

Assume we have produced a regression by solving

$$\mathcal{X}^T\mathcal{X}\hat{\beta} - \mathcal{X}^T\mathbf{y} = 0$$

for the value of $\hat{\beta}$. I write $\hat{\beta}$ because this is an *estimate*; we likely don't have the true value of the β that generated the data (the model might be wrong; etc.). We cannot expect that $\mathcal{X}\hat{\beta}$ is the same as $\mathbf{y}$. Instead, there is likely to be some error. The **residual** is the vector

$$\mathbf{e} = \mathbf{y} - \mathcal{X}\hat{\beta}$$

which gives the difference between the true value and the model's prediction at each point. Each component of the residual is an estimate of the unmodelled effects for

that data point. The **mean square error** is

$$m = \frac{\mathbf{e}^T\mathbf{e}}{N}$$

and this gives the average of the squared error of prediction on the training examples.

Notice that the mean squared error is not a great measure of how good the regression is. This is because the value depends on the units in which the dependent variable is measured. So, for example, if you measure y in meters you will get a different mean squared error than if you measure y in kilometers.

7.2.4 R-squared

There is an important quantitative measure of how good a regression is which doesn't depend on units. Unless the dependent variable is a constant (which would make prediction easy), it has some variance. If our model is of any use, it should explain some aspects of the value of the dependent variable. This means that the variance of the residual should be smaller than the variance of the dependent variable. If the model made perfect predictions, then the variance of the residual should be zero.

We can formalize all this in a relatively straightforward way. We will ensure that $\mathcal{X}$ always has a column of ones in it, so that the regression can have a non-zero y-intercept. We now fit a model

$$\mathbf{y} = \mathcal{X}\beta + \mathbf{e}$$

(where $\mathbf{e}$ is the vector of residual values) by choosing β such that $\mathbf{e}^T\mathbf{e}$ is minimized. Then we get some useful technical results.

Useful Facts: 7.1 *Regression*

We write $\mathbf{y} = \mathcal{X}\hat{\beta} + \mathbf{e}$, where $\mathbf{e}$ is the residual. Assume $\mathcal{X}$ has a column of ones, and $\hat{\beta}$ is chosen to minimize $\mathbf{e}^T\mathbf{e}$. Then we have

1. $\mathbf{e}^T\mathcal{X} = \mathbf{0}$, i.e. that $\mathbf{e}$ is orthogonal to any column of $\mathcal{X}$. This is because, if $\mathbf{e}$ is not orthogonal to some column of $\mathbf{e}$, we can increase or decrease the $\hat{\beta}$ term corresponding to that column to make the error smaller. Another way to see this is to notice that $\hat{\beta}$ is chosen to minimize $\frac{1}{N}\mathbf{e}^T\mathbf{e}$, which is $\frac{1}{N}(\mathbf{y} - \mathcal{X}\hat{\beta})^T(\mathbf{y} - \mathcal{X}\hat{\beta})$. Now because this is a minimum, the gradient with respect to $\hat{\beta}$ is zero, so $(\mathbf{y} - \mathcal{X}\hat{\beta})^T(-\mathcal{X}) = -\mathbf{e}^T\mathcal{X} = 0$.
2. $\mathbf{e}^T\mathbf{1} = 0$ (recall that $\mathcal{X}$ has a column of all ones, and apply the previous result).
3. $\mathbf{1}^T(\mathbf{y} - \mathcal{X}\hat{\beta}) = 0$ (same as previous result).
4. $\mathbf{e}^T\mathcal{X}\hat{\beta} = 0$ (first result means that this is true).

Now $\mathbf{y}$ is a one dimensional dataset arranged into a vector, so we can compute $\mathsf{mean}\,(\{y\})$ and $\mathsf{var}[y]$. Similarly, $\mathcal{X}\hat{\beta}$ is a one dimensional dataset arranged into a vector (its elements are $\mathbf{x}_i^T\hat{\beta}$), as is $\mathbf{e}$, so we know the meaning of mean and variance for each. We have a particularly important result:

$$\mathsf{var}[y] = \mathsf{var}\left[\mathcal{X}\hat{\beta}\right] + \mathsf{var}[e].$$

This is quite easy to show, with a little more notation. Write $\overline{\mathbf{y}} = (1/N)(\mathbf{1}^T\mathbf{y})\mathbf{1}$ for the vector whose entries are all $\mathsf{mean}\,(\{y\})$; similarly for $\overline{\mathbf{e}}$ and for $\overline{\mathcal{X}\hat{\beta}}$. We have

$$\mathsf{var}[y] = (1/N)(\mathbf{y} - \overline{\mathbf{y}})^T(\mathbf{y} - \overline{\mathbf{y}})$$

and so on for $\mathsf{var}[e_i]$, etc. Notice from the facts that $\overline{\mathbf{y}} = \overline{\mathcal{X}\hat{\beta}}$. Now

$$\begin{aligned}
\mathsf{var}[y] &= (1/N)\left(\left[\mathcal{X}\hat{\beta} - \overline{\mathcal{X}\hat{\beta}}\right] + [\mathbf{e} - \overline{\mathbf{e}}]\right)^T\left(\left[\mathcal{X}\hat{\beta} - \overline{\mathcal{X}\hat{\beta}}\right] + [\mathbf{e} - \overline{\mathbf{e}}]\right) \\
&= (1/N)\left(\left[\mathcal{X}\hat{\beta} - \overline{\mathcal{X}\hat{\beta}}\right]^T\left[\mathcal{X}\hat{\beta} - \overline{\mathcal{X}\hat{\beta}}\right] + 2\left[\mathbf{e} - \overline{\mathbf{e}}\right]^T\left[\mathcal{X}\hat{\beta} - \overline{\mathcal{X}\hat{\beta}}\right] + \left[\mathbf{e} - \overline{\mathbf{e}}\right]^T\left[\mathbf{e} - \overline{\mathbf{e}}\right]\right) \\
&= (1/N)\left(\left[\mathcal{X}\hat{\beta} - \overline{\mathcal{X}\hat{\beta}}\right]^T\left[\mathcal{X}\hat{\beta} - \overline{\mathcal{X}\hat{\beta}}\right] + \left[\mathbf{e} - \overline{\mathbf{e}}\right]^T\left[\mathbf{e} - \overline{\mathbf{e}}\right]\right) \\
&\quad \text{because } \overline{\mathbf{e}} = 0 \text{ and } \mathbf{e}^T\mathcal{X}\hat{\beta} = 0 \text{ and } \mathbf{e}^T\mathbf{1} = 0 \\
&= \mathsf{var}\left[\mathcal{X}\hat{\beta}\right] + \mathsf{var}[e].
\end{aligned}$$

This is extremely important, because us allows us to think about a regression as explaining variance in $\mathbf{y}$. As we are better at explaining $\mathbf{y}$, $\mathsf{var}[e]$ goes down. In turn, a natural measure of the goodness of a regression is what percentage of the variance of $\mathbf{y}$ it explains. This is known as R^2 (the r-squared measure). We have

$$R^2 = \frac{\mathsf{var}\left[\mathbf{x}_i^T\hat{\beta}\right]}{\mathsf{var}[y_i]}$$

which gives some sense of how well the regression explains the training data. Notice that the value of R^2 is not affected by the units of $\mathbf{y}$ (exercises)

Good predictions result in high values of R^2, and a perfect model will have $R^2 = 1$ (which doesn't usually happen). For example, the regression of figure 7.3 has an R^2 value of 0.87. Figures 7.1 and 7.2 show the R^2 values for the regressions plotted there; notice how better models yield larger values of R^2. Notice that if you look at the summary that R provides for a linear regression, it will offer you *two* estimates of the value for R^2. These estimates are obtained in ways that try to account for (a) the amount of data in the regression, and (b) the number of variables in the regression. For our purposes, the differences between these numbers and the R^2 I defined are not significant. For the figures, I computed R^2 as I described in the text above, but if you substitute one of R's numbers nothing terrible will happen.

Remember this: *The quality of predictions made by a regression can be evaluated by looking at the fraction of the variance in the dependent variable that is explained by the regression. This number is called R^2, and lies between zero and one; regressions with larger values make better predictions.*

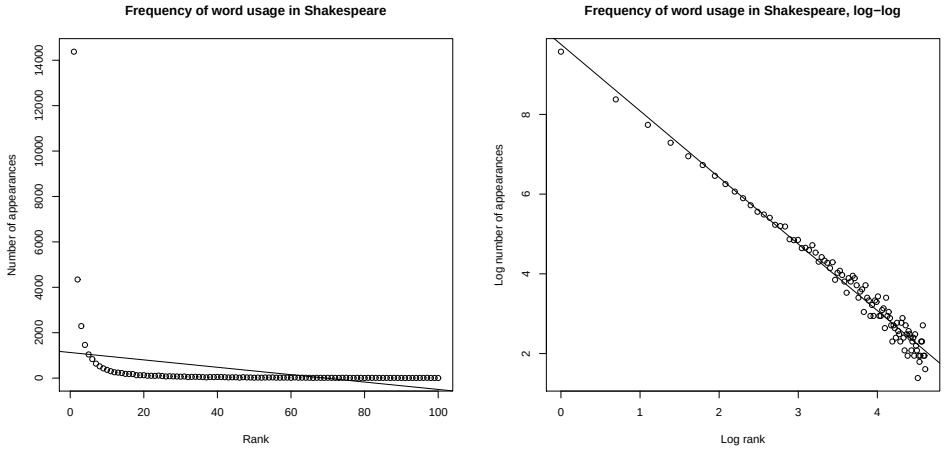


FIGURE 7.4: *On the* **left**, *word count plotted against rank for the 100 most common words in Shakespeare, using a dataset that comes with R (called "bard", and quite likely originating in an unpublished report by J. Gani and I. Saunders). I show a regression line too. This is a poor fit by eye, and the R^2 is poor, too ($R^2 = 0.1$). On the* **right**, *log word count plotted against log rank for the 100 most common words in Shakespeare, using a dataset that comes with R (called "bard", and quite likely originating in an unpublished report by J. Gani and I. Saunders). The regression line is very close to the data.*

7.2.5 Transforming Variables

Sometimes the data isn't in a form that leads to a good linear regression. In this case, transforming explanatory variables, the dependent variable, or both can lead to big improvements. Figure 7.4 shows one example, based on the idea of word frequencies. Some words are used very often in text; most are used seldom. The dataset for this figure consists of counts of the number of time a word occurred for the 100 most common words in Shakespeare's printed works. It was originally collected from a concordance, and has been used to attack a variety of interesting questions, including an attempt to assess how many words Shakespeare knew. This is hard, because he likely knew many words that he didn't use in his works, so one can't just count. If you look at the plot of Figure 7.4, you can see that a linear regression of count (the number of times a word is used) against rank (how

common a word is, 1-100) is not really useful. The most common words are used very often, and the number of times a word is used falls off very sharply as one looks at less common words. You can see this effect in the scatter plot of residual against dependent variable in Figure 7.4 — the residual depends rather strongly on the dependent variable. This is an extreme example that illustrates how poor linear regressions can be.

However, if we regress log-count against log-rank, we get a very good fit indeed. This suggests that Shakespeare's word usage (at least for the 100 most common words) is consistent with **Zipf's law**. This gives the relation between frequency f and rank r for a word as

$$f \propto \frac{1}{r}^s$$

where s is a constant characterizing the distribution. Our linear regression suggests that s is approximately 1.67 for this data.

In some cases, the natural logic of the problem will suggest variable transformations that improve regression performance. For example, one could argue that humans have approximately the same density, and so that weight should scale as the cube of height; in turn, this suggests that one regress weight against the cube root of height. Generally, shorter people tend not to be scaled versions of taller people, so the cube root might be too aggressive, and so one thinks of the square root.

Remember this: *The performance of a regression can be improved by transforming variables. Transformations can follow from looking at plots, or thinking about the logic of the problem*

The **Box-Cox transformation** is a method that can search for a transformation of the dependent variable that improves the regression. The method uses a one-parameter family of transformations, with parameter λ, then searches for the best value of this parameter using maximum likelihood. A clever choice of transformation means that this search is relatively straightforward. We define the Box-Cox transformation of the dependent variable to be

$$y_i^{(bc)} = \begin{cases} \frac{y_i^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0 \\ \log y_i & \text{if } \lambda = 0 \end{cases} .$$

It turns out to be straightforward to estimate a good value of λ using maximum likelihood. One searches for a value of λ that makes residuals look most like a normal distribution. Statistical software will do it for you; the exercises sketch out the method. This transformation can produce significant improvements in a regression. For example, the transformation suggests a value of $\lambda = 0.303$ for the fish example of Figure 7.1. It isn't natural to plot weight$^{0.303}$ against height,

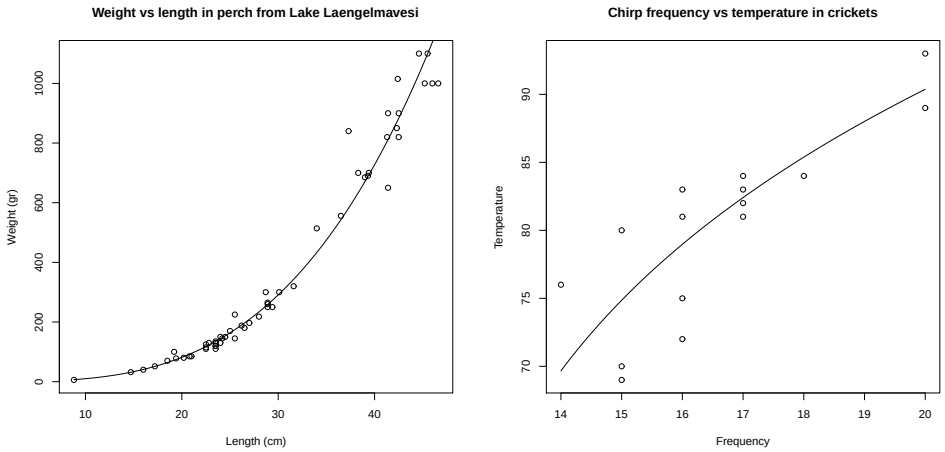


FIGURE 7.5: *The Box-Cox transformation suggests a value of $\lambda = 0.303$ for the regression of weight against height for the perch data of Figure 7.1. You can find this dataset, and the back story at http://www.amstat.org/publications/jse/jse_data_archive.htm; look for "fishcatch" on that page). On the* **left**, *a plot of the resulting curve overlaid on the data. For the cricket temperature data of that figure (from http://mste.illinois.edu/patel/amar430/keyprob1.html), the transformation suggests a value of $\lambda = 4.75$. On the* **right**, *a plot of the resulting curve overlaid on the data.*

because we don't really want to predict weight$^{0.303}$. Instead, we plot the predictions of weight that come from this model, which will lie on a curve with the form $(ax + b)^{\frac{1}{0.303}}$, rather than on a straight line. Similarly, the transformation suggests a value of $\lambda = 0.475$ for the cricket data. Figure 7.5 shows the result of these transforms.

7.2.6 Can you Trust Your Regression?

Linear regression is useful, but it isn't magic. Some regressions make poor predictions (recall the regressions of figure 7.2). As another example, regressing the first digit of your telephone number against the length of your foot won't work.

We have some straightforward tests to tell whether a regression is working. You can **look at a plot** for a dataset with one explanatory variable and one dependent variable. You plot the data on a scatter plot, then plot the model as a line on that scatterplot. Just looking at the picture can be informative (compare Figure 7.1 and Figure 7.2).

You can check if the regression **predicts a constant**. This is usually a bad sign. You can check this by looking at the predictions for each of the training data items. If the variance of these predictions is small compared to the variance of the independent variable, the regression isn't working well. If you have only one explanatory variable, then you can plot the regression line. If the line is horizontal, or close, then the value of the explanatory variable makes very little contribution

to the prediction. This suggests that there is no particular relationship between the explanatory variable and the independent variable.

You can also check, by eye, if **the residual isn't random**. If $y - \mathbf{x}^T\beta$ is a zero mean normal random variable, then the value of the residual vector should not depend on the corresponding y-value. Similarly, if $y - \mathbf{x}^T\beta$ is just a zero mean collection of unmodelled effects, we want the value of the residual vector to not depend on the corresponding y-value either. If it does, that means there is some phenomenon we are not modelling. Looking at a scatter plot of $\mathbf{e}$ against $\mathbf{y}$ will often reveal trouble in a regression (Figure 7.7). In the case of Figure 7.7, the trouble is caused by a few data points that are very different from the others severely affecting the regression. We will discuss how to identify and deal with such points in Section **??**. Once they have been removed, the regression improves markedly (Figure 7.8).

Remember this: *Linear regressions can make bad predictions. You can check for trouble by: evaluating R^2; looking at a plot; looking to see if the regression makes a constant prediction; or checking whether the residual is random. Other strategies exist, but are beyond the scope of this book.*

Procedure: 7.1 *Linear Regression using Least Squares*

We have a dataset containing N pairs $(\mathbf{x}_i, y_i)$. Each x_i is a d-dimensional explanatory vector, and each y_i is a single dependent variable. We assume that each data point conforms to the model

$$y_i = \mathbf{x}_i^T \beta + \xi_i$$

where ξ_i represents unmodelled effects. We assume that ξ_i are samples of a random variable with 0 mean and unknown variance. Sometimes, we assume the random variable is normal. Write

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix}$$

and

$$\mathcal{X} = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \dots \mathbf{x}_n^T \end{pmatrix}.$$

We estimate $\hat{\beta}$ (the value of β) by solving the linear system

$$\mathcal{X}^T \mathcal{X} \hat{\beta} - \mathcal{X}^T \mathbf{y} = 0.$$

For a data point $\mathbf{x}$, our model predicts $\mathbf{x}^T \hat{\beta}$. The residuals are

$$\mathbf{e} = \mathbf{y} - \mathcal{X}\hat{\beta}.$$

We have that $\mathbf{e}^T \mathbf{1} = 0$. The mean square error is given by

$$m = \frac{\mathbf{e}^T \mathbf{e}}{N}.$$

The R^2 is given by

$$\frac{\mathsf{var}\left(\left\{\mathbf{x}_i^T \hat{\beta}\right\}\right)}{\mathsf{var}\left(\{\mathbf{y}\}\right)}.$$

Values of R^2 range from 0 to 1; a larger value means the regression is better at explaining the data.

7.3 PROBLEM DATA POINTS

I have described regressions on a single explanatory variable, because it is easy to plot the line in this case. You can find most problems by looking at the line and

the data points. But a single explanatory variable isn't the most common or useful case. If we have many explanatory variables, it can be hard to plot the regression in a way that exposes problems. This section mainly describes methods to identify and solve difficulties that don't involve looking at the line.

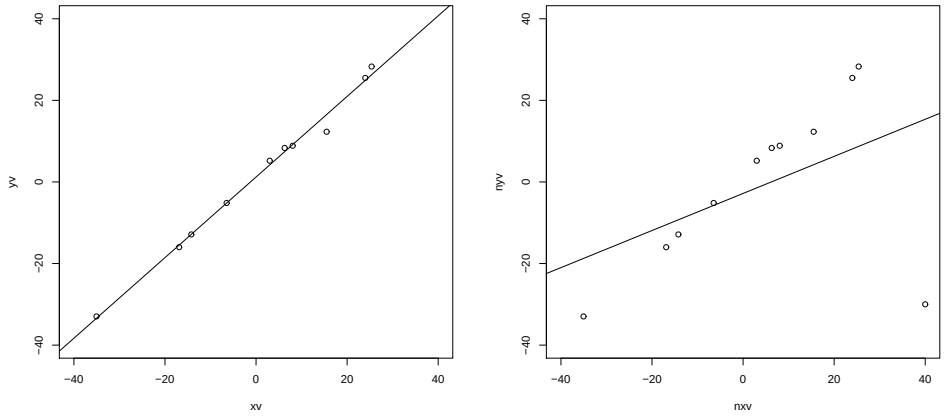


FIGURE 7.6: *On the* **left**, *a synthetic dataset with one independent and one explanatory variable, with the regression line plotted. Notice the line is close to the data points, and its predictions seem likely to be reliable. On the* **right**, *the result of adding a single outlying datapoint to that dataset. The regression line has changed significantly, because the regression line tries to minimize the sum of squared vertical distances between the data points and the line. Because the outlying datapoint is far from the line, the squared vertical distance to this point is enormous. The line has moved to reduce this distance, at the cost of making the other points further from the line.*

7.3.1 Problem Data Points have Significant Impact

Outlying data points can significantly weaken the usefulness of a regression. For some regression problems, we can identify data points that might be a problem, and then resolve how to deal with them. One possibility is that they are true outliers — someone recorded a data item wrong, or they represent an effect that just doesn't occur all that often. Another is that they are important data, and our linear model may not be good enough. If the data points really are outliers, we can drop them from the data set. If they aren't, we may be able to improve the regression by transforming features or by finding a new explanatory variable.

When we construct a regression, we are solving for the β that minimizes $\sum_i (y_i - \mathbf{x}_i^T \beta)^2$, equivalently for the β that produces the smallest value of $\sum_i e_i^2$. This means that residuals with large value can have a very strong influence on the outcome — we are squaring that large value, resulting in an enormous value. Generally, many residuals of medium size will have a smaller cost than one large residual and the rest tiny. As figure 7.6 illustrates, this means that a data point

that lies far from the others can swing the regression line significantly.

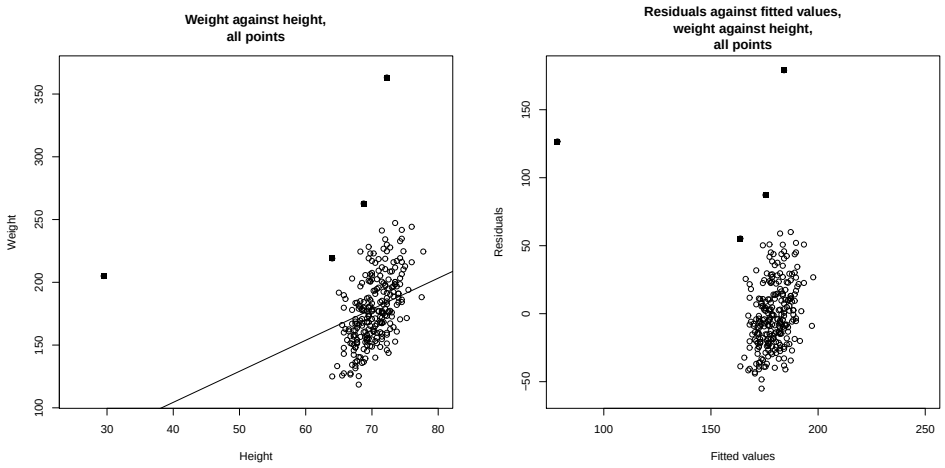


FIGURE 7.7: *On the* **left**, *weight regressed against height for the bodyfat dataset. The line doesn't describe the data particularly well, because it has been strongly affected by a few data points (filled-in markers). On the* **right**, *a scatter plot of the residual against the value predicted by the regression. This doesn't look like noise, which is a sign of trouble.*

This creates a problem, because data points that are very different from most others (sometimes called **outliers**) can also have the highest influence on the outcome of the regression. Figure 7.8 shows this effect for a simple case. When we have only one explanatory variable, there's an easy method to spot problem data points. We produce a scatter plot and a regression line, and the difficulty is usually obvious. In particularly tricky cases, printing the plot and using a see-through ruler to draw a line by eye can help (if you use an opaque ruler, you may not see some errors).

These data points can come from many sources. They may simply be errors. Failures of equipment, transcription errors, someone guessing a value to replace lost data, and so on are some methods that might produce outliers. Another possibility is your understanding of the problem is wrong. If there are some rare effects that are very different than the most common case, you might see outliers. Major scientific discoveries have resulted from investigators taking outliers seriously, and trying to find out what caused them (though you shouldn't see a Nobel prize lurking behind every outlier).

What to do about outliers is even more fraught. The simplest strategy is to find them, then remove them from the data. I will describe some methods that can identify outliers, but you should be aware that this strategy can get dangerous fairly quickly. First, you might find that each time you remove a few problematic data points, some more data points look strange to you. This process is unlikely to end well. Second, you should be aware that throwing out outliers can *increase* your future prediction error, particularly if they're caused by real effects. An alternative

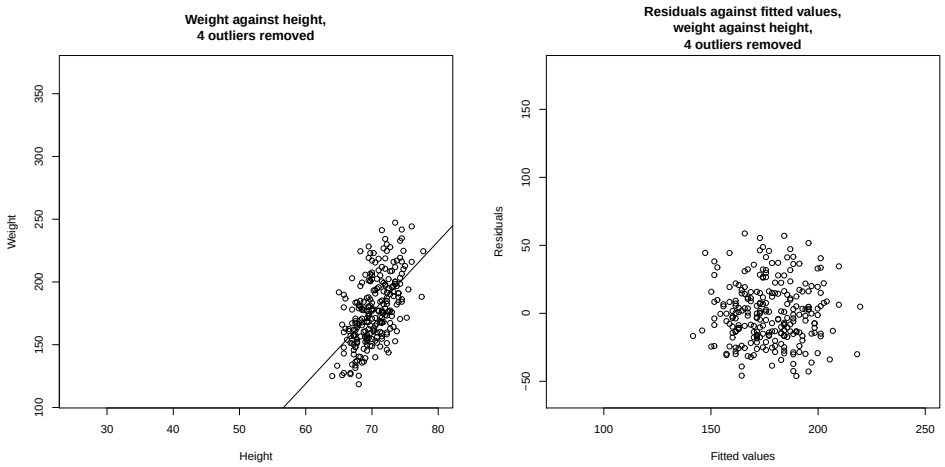


FIGURE 7.8: *On the* **left**, *weight regressed against height for the bodyfat dataset. I have now removed the four suspicious looking data points, identified in Figure 7.7 with filled-in markers; these seemed the most likely to be outliers. On the* **right**, *a scatter plot of the residual against the value predicted by the regression. Notice that the residual looks like noise. The residual seems to be uncorrelated to the predicted value; the mean of the residual seems to be zero; and the variance of the residual doesn't depend on the predicted value. All these are good signs, consistent with our model, and suggest the regression will yield good predictions.*

strategy is to build methods that can either discount the effects of outliers, or model them; I describe some such methods, which can be technically complex, in the following chapter.

Remember this: *Outliers can affect linear regressions significantly. Usually, if you can plot the regression, you can look for outliers by eyeballing the plot. Other methods exist, but are beyond the scope of this text.*

7.3.2 The Hat Matrix and Leverage

Write $\hat{\beta}$ for the estimated value of β, and $\mathbf{y}^{(p)} = \mathcal{X}\hat{\beta}$ for the predicted y values. Then we have

$$\hat{\beta} = \left(\mathcal{X}^T\mathcal{X}\right)^{-1}\left(\mathcal{X}^T\mathbf{y}\right)$$

so that

$$\mathbf{y}^{(p)} = (\mathcal{X}\left(\mathcal{X}^T\mathcal{X}\right)^{-1}\mathcal{X}^T)\mathbf{y}.$$

What this means is that the values the model predicts at training points are a linear function of the true values at the training points. The matrix $(\mathcal{X}\left(\mathcal{X}^T\mathcal{X}\right)^{-1}\mathcal{X}^T)$ is

sometimes called the **hat matrix**. The hat matrix is written $\mathcal{H}$, and I shall write the i, j'th component of the hat matrix h_{ij}.

Remember this: *The predictions of a linear regression at training points are a linear function of the y-values at the training points. The linear function is given by the hat matrix.*

The hat matrix has a variety of important properties. I won't prove any here, but the proofs are in the exercises. It is a symmetric matrix. The eigenvalues can be only 1 or 0. And the row sums have the important property that

$$\sum_j h_{ij}^2 \leq 1.$$

This is important, because it can be used to find data points that have values that are hard to predict. The **leverage** of the i'th training point is the i'th diagonal element, h_{ii}, of the hat matrix $\mathcal{H}$. Now we can write the prediction at the i'th training point $y_{p,i} = h_{ii}y_i + \sum_{j \neq i} h_{ij}y_j$. But if h_{ii} has large absolute value, then all the other entries in that row of the hat matrix must have small absolute value. This means that, if a data point has high leverage, the model's value at that point is predicted almost entirely by the observed value at that point. Alternatively, it's hard to use the other training data to predict a value at that point.

Here is another way to see this importance of h_{ii}. Imagine we change the value of y_i by adding Δ; then $y_i^{(p)}$ becomes $y_i^{(p)} + h_{ii}\Delta$. In turn, a large value of h_{ii} means that the predictions at the i'th point are very sensitive to the value of y_i.

Remember this: *Ideally, the value predicted for a particular data point depends on many other data points. Leverage measures the importance of a data point in producing a prediction at that data point. If the leverage of a point is high, other points are not contributing much to the prediction for that point, and it may well be an outlier.*

7.3.3 Cook's Distance

Another way to find points that may be creating problems is to look at the effect of omitting the point from the regression. We could compute $\mathbf{y}^{(p)}$ using the whole data set. We then omit the i'th point from the dataset, compute the regression coefficients from the remaining data (which I will write $\hat{\beta}_{\hat{i}}$), then compare $\mathbf{y}^{(p)}$ to $\mathcal{X}\hat{beta}_{\hat{i}}$. If there is a large difference, the point is suspect, because omitting it

strongly changes the predictions. The score for the comparison is called **Cook's distance**. If a point has a large value of Cook's distance, then it has a strong influence on the regression and might well be an outlier. Typically, one computes Cook's distance for each point, and takes a closer look at any point with a large value. This procedure is described in more detail in procedure 25

Notice the rough similarity to cross-validation (omit some data and recompute). But in this case, we are using the procedure to identify points we might not trust, rather than to get an unbiased estimate of the error.

Procedure: 7.2 *Computing Cook's distance*

We have a dataset containing N pairs $(\mathbf{x}_i, y_i)$. Each x_i is a d-dimensional explanatory vector, and each y_i is a single dependent variable. Write $\hat{\beta}$ for the coefficients of a linear regression (see procedure 7.1), and $\hat{\beta}_{\hat{i}}$ for the coefficients of the linear regression computed by omitting the i'th data point, $\mathbf{y}^{(p)}$ for $\mathcal{X}\hat{\beta}$, and m for the mean square error. The Cook's distance of the i'th data point is

$$\frac{(\mathbf{y}^{(p)} - \mathcal{X}\hat{\beta}_{\hat{i}})^T(\mathbf{y}^{(p)} - \mathcal{X}\hat{\beta}_{\hat{i}})}{dm}.$$

Large values of this distance suggest a point may present problems. Statistical software will compute and plot this distance for you.

Remember this: *The Cook's distance of a training data point measures the effect on predictions of leaving that point out of the regression. A large value of Cook's distance suggests other points are poor at predicting the value at a given point, so a point with a large value of Cook's distance may be an outlier.*

7.3.4 Standardized Residuals

The hat matrix has another use. It can be used to tell how "large" a residual is. The residuals that we measure depend on the units in which y was expressed, meaning we have no idea what a "large" residual is. For example, if we were to express y in kilograms, then we might want to think of 0.1 as a small residual. Using exactly the same dataset, but now with y expressed in grams, that residual value becomes 100 — is it really "large" because we changed units?

Now recall that we assumed, in section 7.2.1, that $y - \mathbf{x}^T\beta$ was a zero mean normal random variable, but we didn't know its variance. It can be shown that,

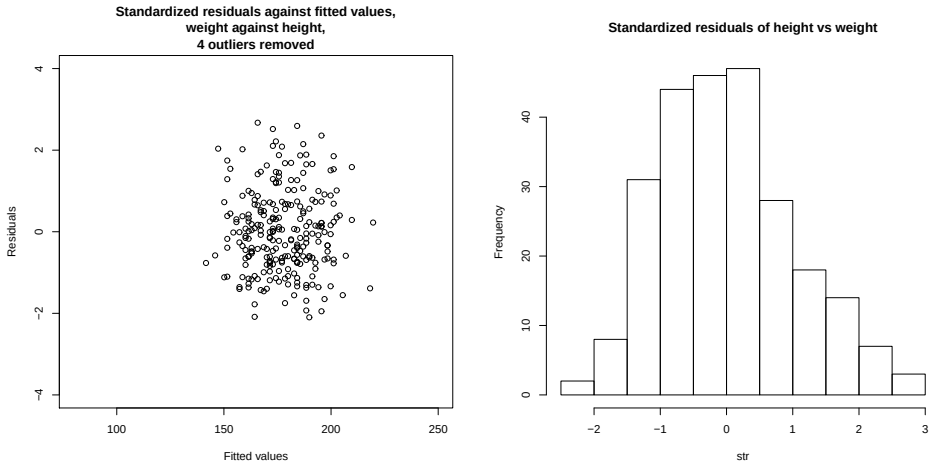


FIGURE 7.9: *On the* **left**, *standardized residuals plotted against predicted value for weight regressed against height for the bodyfat dataset. I removed the four suspicious looking data points, identified in Figure 7.7 with filled-in markers ; these seemed the most likely to be outliers. You should compare this plot with the residuals in figure 7.8, which are not standardized. On the* **right**, *a histogram of the residual values. Notice this looks rather like a histogram of a standard normal random variable, though there are slightly more large positive residuals than one would like. This suggests the regression is working tolerably.*

under our assumption, the i'th residual value, e_i, is a sample of a normal random variable whose variance is

$$\left(\frac{(\mathbf{e}^T\mathbf{e})}{N}\right)(1-h_{ii}).$$

This means we can tell whether a residual is large by **standardizing** it – that is, dividing by its standard deviation. Write s_i for the standard residual at the i'th training point. Then we have that

$$s_i = \frac{e_i}{\sqrt{\left(\frac{(\mathbf{e}^T\mathbf{e})}{N}\right)(1-h_{ii})}}.$$

When the regression is behaving, this standard residual should look like a sample of a standard normal random variable. In turn, this means that if all is going well, about 66% of the residuals should have values in the range $[-1, 1]$, and so on. Large values of the standard residuals are a sign of trouble.

R produces a nice diagnostic plot that can be used to look for problem data points (code and details in the appendix). The plot is a scatter plot of the standardized residuals against leverage, with level curves of Cook's distance superimposed. Figure 7.10 shows an example. Some bad points that are likely to present problems are identified with a number (you can control how many, and the number, with arguments to `plot`; appendix). Problem points will have high leverage and/or high

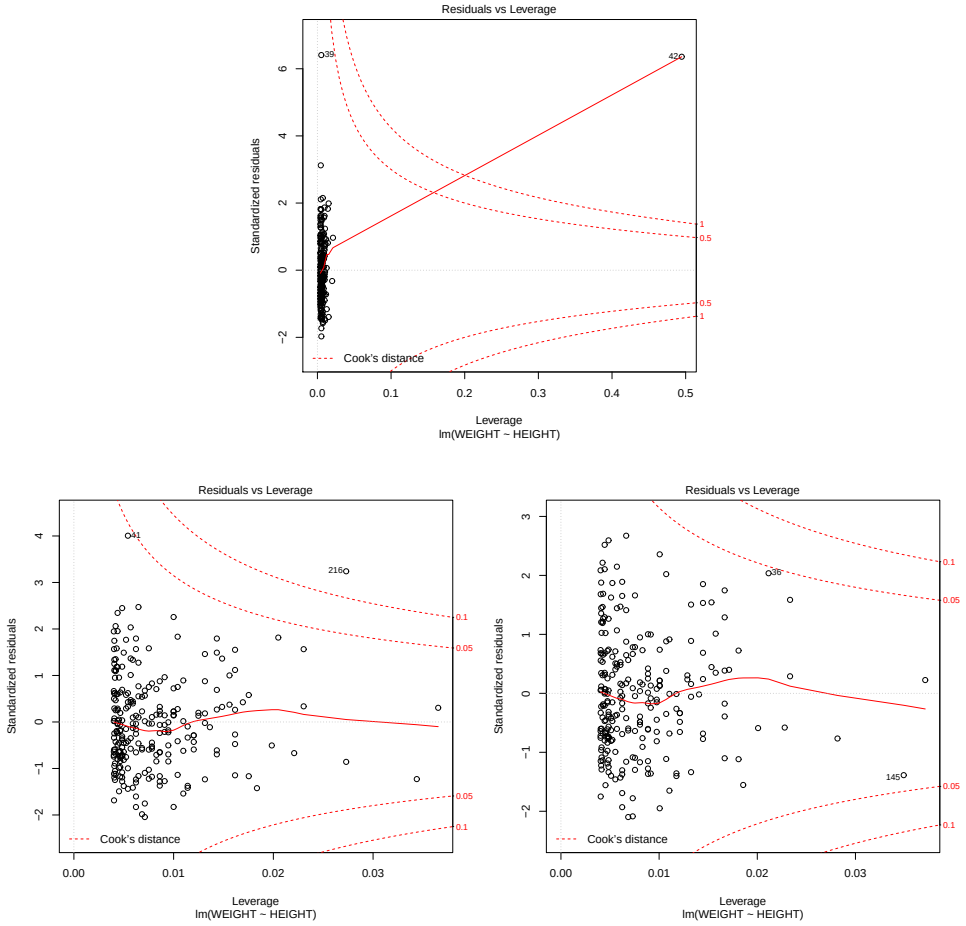


FIGURE 7.10: *A diagnostic plot, produced by R, of a linear regression of weight against height for the bodyfat dataset.* **Top:** *the whole dataset;* **bottom left:** *with the two most extreme points in the top figure removed;* **bottom right:** *with two further points (highest residual) removed. Details in text.*

Cook's distance and/or high residual. The figure shows this plot for three different versions of the dataset (original; two problem points removed; and two further problem points removed).

7.4 MANY EXPLANATORY VARIABLES

In earlier sections, I implied you could put anything into the explanatory variables. This is correct, and makes it easy to do the math for the general case. However, I have plotted only cases where there was one explanatory variable (together with a constant, which hardly counts). In some cases (section 7.4.1), we can add explana-

tory variables and still have an easy plot. Adding explanatory variables can cause the matrix $\mathcal{X}^T\mathcal{X}$ to have poor condition number; there's an easy strategy to deal with this (section 7.4.2).

Most cases are hard to plot successfully, and one needs better ways to visualize the regression than just plotting. The value of R^2 is still a useful guide to the goodness of the regression, but the way to get more insight is to use the tools of the previous section.

7.4.1 Functions of One Explanatory Variable

Imagine we have only one measurement to form explanatory variables. For example, in the perch data of Figure 7.1, we have only the length of the fish. If we evaluate functions of that measurement, and insert them into the vector of explanatory variables, the resulting regression is still easy to plot. It may also offer better predictions. The fitted line of Figure 7.1 looks quite good, but the data points look as though they might be willing to follow a curve. We can get a curve quite easily. Our current model gives the weight as a linear function of the length with a noise term (which we wrote $y_i = \beta_1 x_i + \beta_0 + \xi_i$). But we could expand this model to incorporate other functions of the length. In fact, it's quite suprising that the weight of a fish should be predicted by its length. If the fish doubled in each direction, say, its weight should go up by a factor of eight. The success of our regression suggests that fish do not just scale in each direction as they grow. But we might try the model $y_i = \beta_2 x_i^2 + \beta_1 x_i + \beta_0 + \xi_i$. This is easy to do. The i'th row of the matrix $\mathcal{X}$ currently looks like $[x_i, 1]$. We build a new matrix $\mathcal{X}^{(b)}$, where the i'th row is $[x_i^2, x_i, 1]$, and proceed as before. This gets us a new model. The nice thing about this model is that it is easy to plot – our predicted weight is still a function of the length, it's just not a linear function of the length. Several such models are plotted in Figure 7.11.

You should notice that it can be quite easy to add a lot of functions like this (in the case of the fish, I tried x_i^3 as well). However, it's hard to decide whether the regression has actually gotten better. The least-squares error *on the training data* will *never* go up when you add new explanatory variables, so the R^2 will *never* get worse. This is easy to see, because you could always use a coefficient of zero with the new variables and get back the previous regression. However, the models that you choose are likely to produce worse and worse predictions as you add explanatory variables. Knowing when to stop can be tough (Section 8.1), though it's sometimes obvious that the model is untrustworthy (Figure 7.11).

Remember this: *If you have only one measurement, you can construct a high dimensional* $\mathbf{x}$ *by using functions of that measurement. This produces a regression that has many explanatory variables, but is still easy to plot. Knowing when to stop is hard. An understanding of the problem is helpful.*

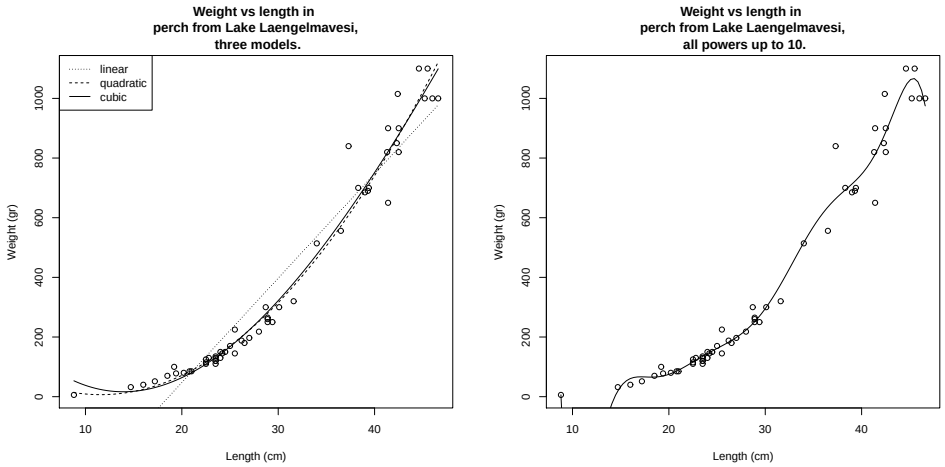


FIGURE 7.11: *On the* **left**, *several different models predicting fish weight from length. The line uses the explanatory variables* 1 *and* x_i*; and the curves use other monomials in* x_i *as well, as shown by the legend. This allows the models to predict curves that lie closer to the data. It is important to understand that, while you can make a curve go closer to the data by inserting monomials, that doesn't mean you necessarily have a better model. On the* **right**, *I have used monomials up to* x_i^{10}*. This curve lies very much closer to the data points than any on the other side, at the cost of some very odd looking wiggles inbetween data points (look at small lengths; the model goes quite strongly negative there, but I can't bring myself to change the axes and show predictions that are obvious nonsense). I can't think of any reason that these structures would come from true properties of fish, and it would be hard to trust predictions from this model.*

7.4.2 Regularizing Linear Regressions

When we have many explanatory variables, some might be significantly correlated. This means that we can predict, quite accurately, the value of one explanatory variable using the values of the other variables. This means there must be a vector $\mathbf{w}$ so that $\mathcal{X}\mathbf{w}$ is small (exercises). In turn, that $\mathbf{w}^T\mathcal{X}^T\mathcal{X}\mathbf{w}$ must be small, so that $\mathcal{X}^T\mathcal{X}$ has some small eigenvalues. These small eigenvalues lead to bad predictions, as follows. The vector $\mathbf{w}$ has the property that $\mathcal{X}^T\mathcal{X}\mathbf{w}$ is small. This means that $\mathcal{X}^T\mathcal{X}(\hat{\beta} + \mathbf{w})$ is not much different from $\mathcal{X}^T\mathcal{X}\hat{\beta}$ (equivalently, the matrix can turn large vectors into small ones). All this means that $(\mathcal{X}^T\mathcal{X})^{-1}$ will turn some small vectors into big ones. A small change in $\mathcal{X}^T\mathbf{Y}$ can lead to a large change in the estimate of $\hat{\beta}$.

This is a problem, because we can expect that different samples from the same data will have somewhat different values of $\mathcal{X}^T\mathbf{Y}$. For example, imagine the person recording fish measurements in Lake Laengelmavesi recorded a different set of fish; we expect changes in $\mathcal{X}$ and $\mathbf{Y}$. But, if $\mathcal{X}^T\mathcal{X}$ has small eigenvalues, these changes could produce large changes in our model.

The problem is relatively easy to control. When there are small eigenvalues

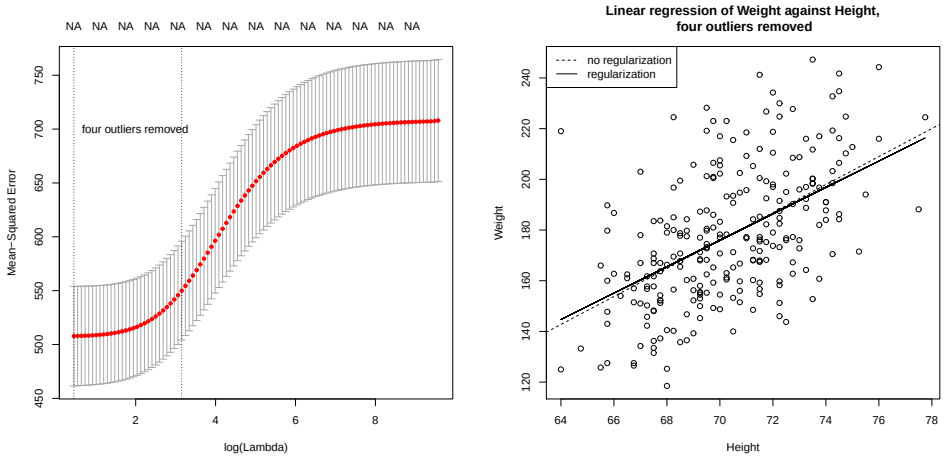


FIGURE 7.12: *On the* **left**, *cross-validated error estimated for different choices of regularization constant for a linear regression of weight against height for the bodyfat dataset, with four outliers removed. The horizontal axis is log regression constant; the vertical is cross-validated error. The mean of the error is shown as a spot, with vertical error bars. The vertical lines show a range of reasonable choices of regularization constant (***left*** yields the lowest observed error,* **right** *the error whose mean is within one standard error of the minimum). On the* **right**, *two regression lines on a scatter plot of this dataset; one is the line computed without regularization, the other is obtained using the regularization parameter that yields the lowest observed error. In this case, the regularizer doesn't change the line much, but may produce improved values on new data (notice how the cross-validated error is fairly flat with low values of the regularization constant).*

in $\mathcal{X}^T\mathcal{X}$, we expect that $\hat{\beta}$ will be large (because we can add components in the direction of $\mathbf{w}$ without changing all that much), and the largest components in $\hat{\beta}$ might be very inaccurately estimated. If we are trying to predict new y values, we expect that large components in $\hat{\beta}$ turn into large errors in prediction (exercises).

An important and useful way to suppress these errors is to try to find a $\hat{\beta}$ that isn't large, and also gives a low error. We can do this by regularizing, using the same trick we saw in the case of classification. Instead of choosing the value of β that minimizes

$$\left(\frac{1}{N}\right)(\mathbf{y}-\mathcal{X}\beta)^T(\mathbf{y}-\mathcal{X}\beta)$$

we minimize

$$\begin{array}{rcl}\left(\frac{1}{N}\right)(\mathbf{y}-\mathcal{X}\beta)^T(\mathbf{y}-\mathcal{X}\beta) & + & \lambda\beta^T\beta \\ \text{Error} & + & \text{Regularizer}\end{array}$$

Here $\lambda > 0$ is a constant that weights the two requirements (small error; small $\hat{\beta}$) relative to one another. Notice also that dividing the total error by the number of

data points means that our choice of λ shouldn't be affected by changes in the size of the data set.

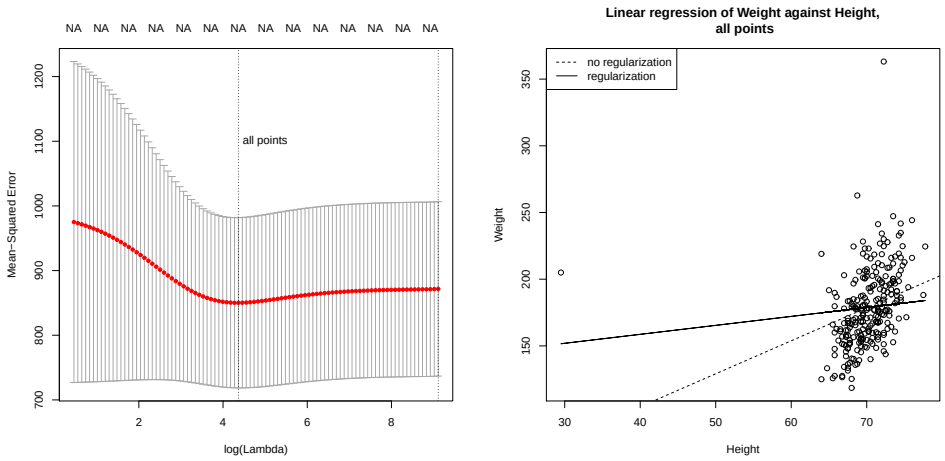


FIGURE 7.13: *Regularization doesn't make outliers go away. On the* **left**, *cross-validated error estimated for different choices of regularization constant for a linear regression of weight against height for the bodyfat dataset, with all points. The horizontal axis is log regression constant; the vertical is cross-validated error. The mean of the error is shown as a spot, with vertical error bars. The vertical lines show a range of reasonable choices of regularization constant (***left** *yields the lowest observed error,* **right** *the error whose mean is within one standard error of the minimum). On the* **right**, *two regression lines on a scatter plot of this dataset; one is the line computed without regularization, the other is obtained using the regularization parameter that yields the lowest observed error. In this case, the regularizer doesn't change the line much, but may produce improved values on new data (notice how the cross-validated error is fairly flat with low values of the regularization constant).*

Regularization helps deal with the small eigenvalue, because to solve for β we must solve the equation

$$\left[\left(\frac{1}{N}\right)\mathcal{X}^T\mathcal{X} + \lambda\mathcal{I}\right]\hat{\beta} = \left(\frac{1}{N}\right)\mathcal{X}^T\mathbf{y}$$

(obtained by differentiating with respect to β and setting to zero) and the smallest eigenvalue of the matrix $\left(\left(\frac{1}{N}\right)\left(\mathcal{X}^T\mathcal{X} + \lambda\mathcal{I}\right)\right.$ will be at least λ (exercises). Penalizing a regression with the size of β in this way is sometimes known as **ridge regression**.

We choose λ in the same way we used for classification; split the training set into a training piece and a validation piece, train for different values of λ, and test the resulting regressions on the validation piece. The error is a random variable, random because of the random split. It is a fair model of the error that would occur on a randomly chosen test example (assuming that the training set is "like" the test set, in a way that I do not wish to make precise yet). We could use multiple

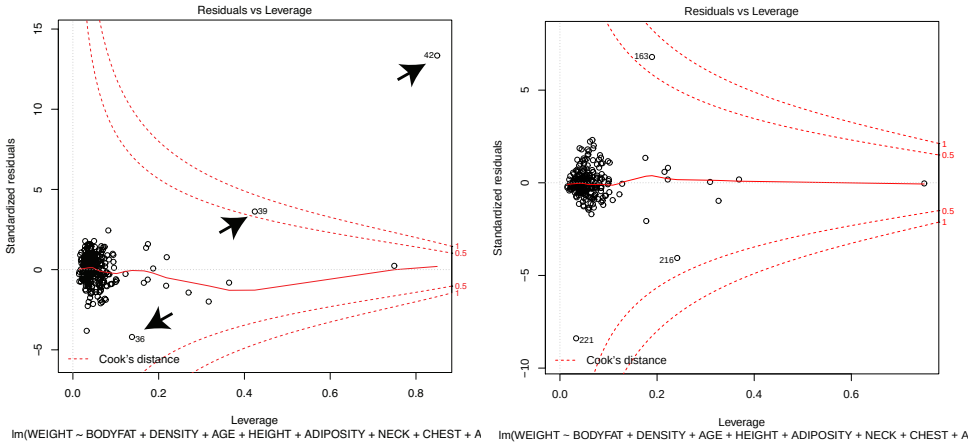


FIGURE 7.14: *On the* **left**, *residuals plotted against leverage for a regression of weight against all other measurements for the bodyfat dataset. I did* not *remove the outliers. The contours on the plot are contours of Cook's distance; I have overlaid arrows showing points with suspiciously large Cook's distance. Notice also that several points have high leverage, without having a large residual value. These points may or may not present problems. On the* **right**, *the same plot for this dataset with points 36, 39, 41 and 42 removed (these are the points I have been removing for each such plot). Notice that another point now has high Cook's distance, but mostly the residual is much smaller.*

splits, and average over the splits. Doing so yields both an average error for a value of λ and an estimate of the standard deviation of error.

Statistical software will do all the work for you. I used the `glmnet` package in R (see exercises for details). Figure 7.12 shows an example, for weight regressed against height. Notice the regularization doesn't change the model (plotted in the figure) all that much. For each value of λ (horizontal axis), the method has computed the mean error and standard deviation of error using cross-validation splits, and displays these with error bars. Notice that $\lambda = 0$ yields poorer predictions than a larger value; large $\hat{\beta}$ really are unreliable. Notice that now there is now no λ that yields the smallest validation error, because the value of error depends on the random splits used in cross-validation. A reasonable choice of λ lies between the one that yields the smallest error encountered (one vertical line in the plot) and the largest value whose mean error is within one standard deviation of the minimum (the other vertical line in the plot).

All this is quite similar to regularizing a classification problem. We started with a cost function that evaluated the errors caused by a choice of β, then added a term that penalized β for being "large". This term is the squared length of β, as a vector. It is sometimes known as the L_2 **norm** of the vector. In section 10.1, I describe the consequences of using other norms.

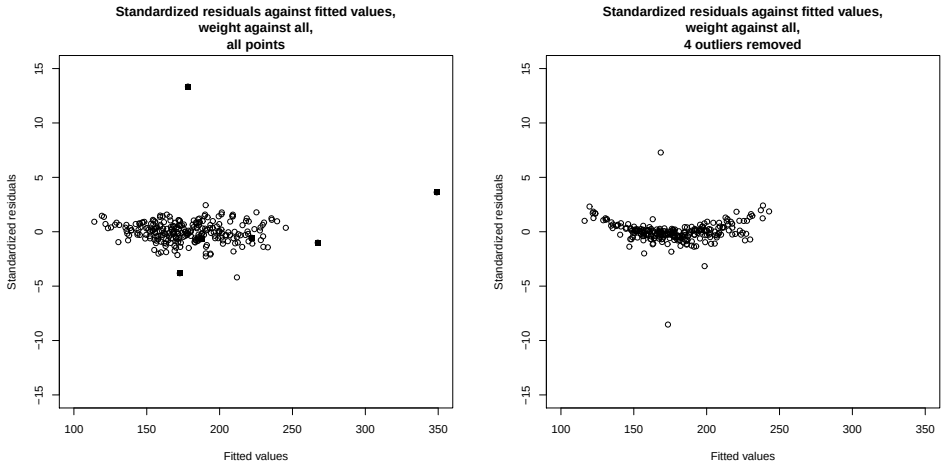


FIGURE 7.15: *On the* **left**, *standardized residuals plotted against predicted value for weight regressed against all variables for the bodyfat dataset. Four data points appear suspicious, and I have marked these with a filled in marker. On the* **right**, *standardized residuals plotted against predicted value for weight regressed against all variables for the bodyfat dataset, but with the four suspicious looking data points removed. Notice two other points stick out markedly.*

> **Remember this:** *The performance of a regression can be improved by regularizing, particularly if some explanatory variables are correlated. The procedure is similar to that used for classification.*

7.4.3 Example: Weight against Body Measurements

We can now look at regressing weight against all body measurements for the bodyfat dataset. We can't plot this regression (too many independent variables), but we can approach the problem in a series of steps.

Finding suspect points: Figure 7.14 shows the R diagnostic plots for a regression of weight against all body measurements for the bodyfat dataset. We've already seen there are outliers, so the odd structure of this plot should be no particular surprise. There are several really worrying points here. As the figure shows, removing the four points identified in the caption, based on their very high standardized residuals, high leverage, and high Cook's distance, yields improvements. We can get some insight by plotting standardized residuals against predicted value (Figure 7.9). There is clearly a problem here; the residual seems to depend quite strongly on the predicted value. Removing the four outliers we have already identified leads to a much improved plot, also shown in Figure 7.15. This is banana-shaped, which is suspicious. There are two points that seem to come from some

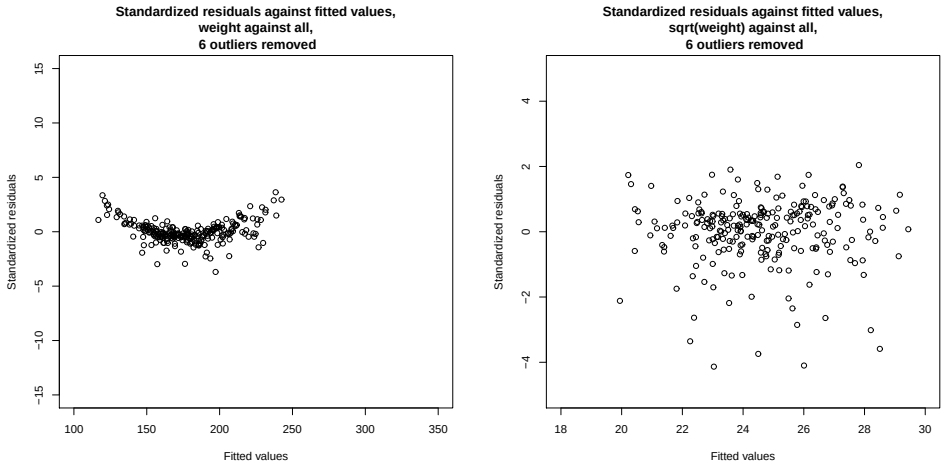


FIGURE 7.16: *On the* **left**, *standardized residuals plotted against predicted value for weight regressed against all variables for the bodyfat dataset. I removed the four suspicious data points of Figure 7.15, and the two others identified in that figure. Notice a suspicious "banana" shape – the residuals are distinctly larger for small and for large predicted values. This suggests a non-linear transformation of something might be helpful. I used a Box-Cox transformation, which suggested a value of 0.5 (i.e. regress* $2(\sqrt{weight} - 1)$*) against all variables. On the* **right**, *the standardized residuals for this regression. Notice that the "banana" has gone, though there is a suspicious tendency for the residuals to be smaller rather than larger. Notice also the plots are on different axes. It's fair to compare these plots by eye; but it's not fair to compare details, because the residual of a predicted square root means something different than the residual of a predicted value.*

other model (one above the center of the banana, one below). Removing these points gives the residual plot shown in Figure 7.16.

Transforming variables: The banana shape of the plot of standardized residuals against value is a suggestion that some non-linearity somewhere would improve the regression. One option is a non-linear transformation of the independent variables. Finding the right one might require some work, so it's natural to try a Box-Cox transformation first. This gives the best value of the parameter as 0.5 (i.e. the dependent variable should be $\sqrt{\text{weight}}$, which makes the residuals look much better (Figure 7.16).

Choosing a regularizing value: Figure 7.17 shows the `glmnet` plot of cross-validated error as a function of regularizer weight. A sensible choice of value here seems to be a bit smaller than -2 (between the value that yields the smallest error encountered – one vertical line in the plot – and the largest value whose mean error is within one standard deviation of the minimum – the other vertical line in the plot). I chose -2.2

How good are the resulting predictions likely to be: the standardized residuals don't seem to depend on the predicted values, but how good are the

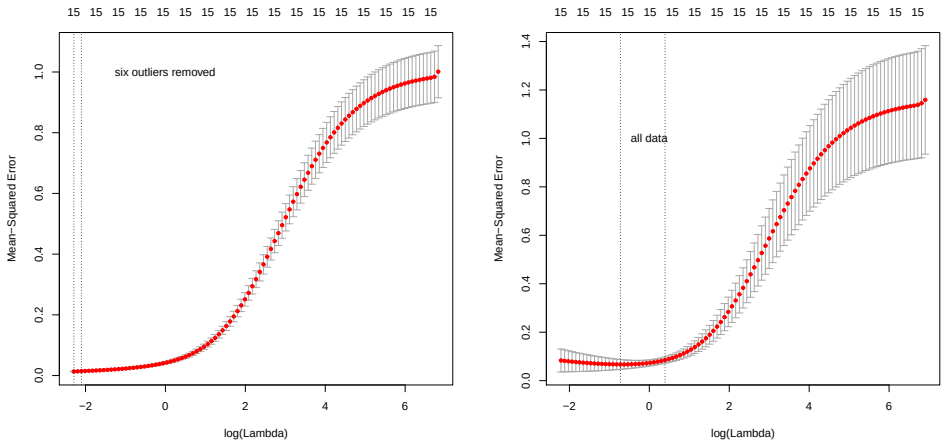


FIGURE 7.17: *Plots of mean-squared error as a function of log regularization parameter (i.e.* $\log \lambda$*) for a regression of weight*$^{1/2}$ *against all variables for the bodyfat dataset. These plots show mean-squared error averaged over cross-validation folds with a vertical one standard deviation bar. On the* **left***, the plot for the dataset with the six outliers identified in Figure 10.1 removed. On the* **right***, the plot for the whole dataset. Notice how the outliers increase the variability of the error, and the best error.*

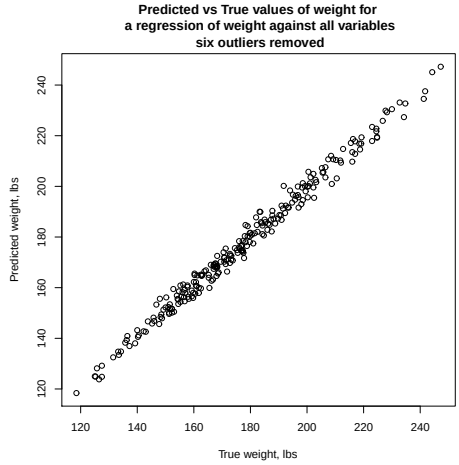


FIGURE 7.18: *A scatter plot of the predicted weight against the true weight for the bodyfat dataset. The prediction is made with all variables, but the six outliers identified above are omitted. I used a Box-Cox transformation with parameter* $1/2$*, and the regularization parameter that yielded the smallest mean square error in Figure 7.17.*

predictions? We already have some information on this point. Figure 7.17 shows cross-validation errors for regressions of weight$^{1/2}$ against height for different regularization weights, but some will find this slightly indirect. We want to predict weight, not weight$^{1/2}$. I chose the regularization weight that yielded the lowest mean-square-error for the model of Figure 7.17, omitting the six outliers previously mentioned. I then computed the predicted weight for each data point using that model (which predicts weight$^{1/2}$, remember; but squaring takes care of that). Figure 7.18 shows the predicted values plotted against the true values. You should not regard this plot as a safe way to estimate generalization (the points were used in training the model; Figure 7.17 is better for that), but it helps to visualize the errors. This regression looks as though it is quite good at predicting bodyweight from other measurements.

7.5 YOU SHOULD

7.5.1 remember:

APPENDIX: DATA

Batch A		Batch B		Batch C	
Amount of Hormone	Time in Service	Amount of Hormone	Time in Service	Amount of Hormone	Time in Service
25.8	99	16.3	376	28.8	119
20.5	152	11.6	385	22.0	188
14.3	293	11.8	402	29.7	115
23.2	155	32.5	29	28.9	88
20.6	196	32.0	76	32.8	58
31.1	53	18.0	296	32.5	49
20.9	184	24.1	151	25.4	150
20.9	171	26.5	177	31.7	107
30.4	52	25.8	209	28.5	125

TABLE 7.1: *A table showing the amount of hormone remaining and the time in service for devices from lot A, lot B and lot C. The numbering is arbitrary (i.e. there's no relationship between device 3 in lot A and device 3 in lot B). We expect that the amount of hormone goes down as the device spends more time in service, so cannot compare batches just by comparing numbers.*

PROBLEMS

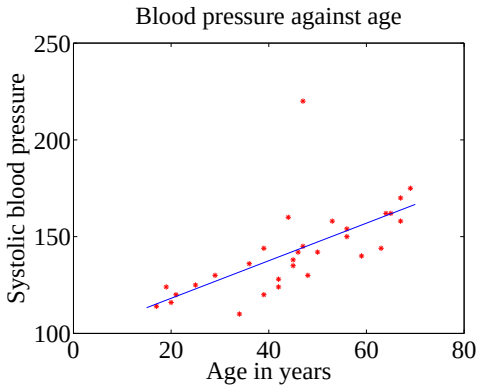


FIGURE 7.19: *A regression of blood pressure against age, for 30 data points.*

7.1. Figure 7.19 shows a linear regression of systolic blood pressure against age. There are 30 data points.

(a) Write $e_i = y_i - \mathbf{x}_i^T \beta$ for the residual. What is the $\mathsf{mean}\,(\{e\})$ for this regression?

(b) For this regression, $\mathsf{var}\,(\{y\}) = 509$ and the R^2 is 0.4324. What is $\mathsf{var}\,(\{e\})$ for this regression?

(c) How well does the regression explain the data?

(d) What could you do to produce better predictions of blood pressure (without actually measuring blood pressure)?

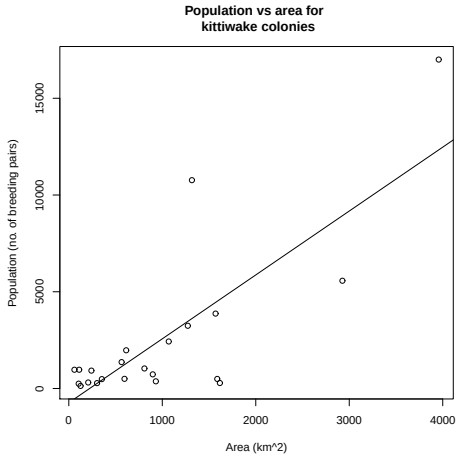


FIGURE 7.20: *A regression of the number of breeding pairs of kittiwakes against the area of an island, for 22 data points.*

7.2. At http://www.statsci.org/data/general/kittiwak.html, you can find a dataset collected by D.K. Cairns in 1988 measuring the area available for a seabird (black-legged kittiwake) colony and the number of breeding pairs for a variety of different colonies. Figure 7.20 shows a linear regression of the number of breeding pairs against the area. There are 22 data points.

(a) Write $e_i = y_i - \mathbf{x}_i^T \beta$ for the residual. What is the $\mathsf{mean}(\{e\})$ for this regression?

(b) For this regression, $\mathsf{var}(\{y\}) = 16491357$ and the R^2 is 0.62. What is $\mathsf{var}(\{e\})$ for this regression?

(c) How well does the regression explain the data? If you had a large island, to what extent would you trust the prediction for the number of kittiwakes produced by this regression? If you had a small island, would you trust the answer more?

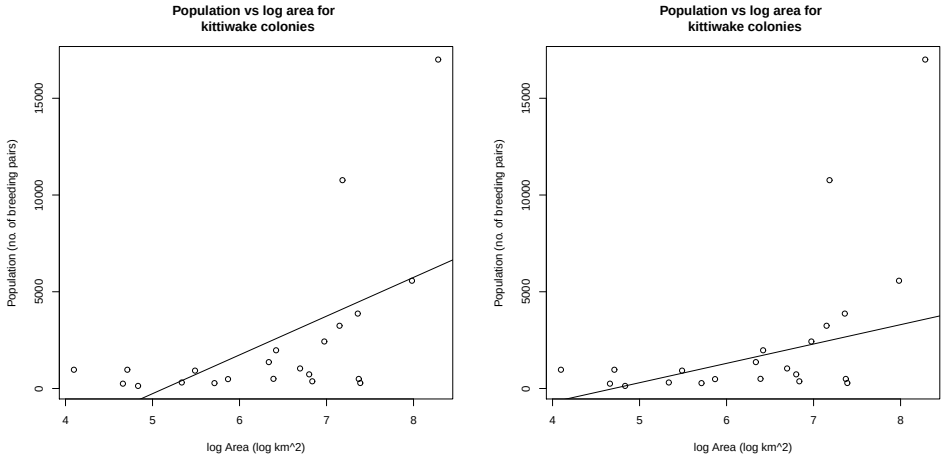


FIGURE 7.21: **Left:** *A regression of the number of breeding pairs of kittiwakes against the log of area of an island, for 22 data points.* **Right:** *A regression of the number of breeding pairs of kittiwakes against the log of area of an island, for 22 data points, using a method that ignores two likely outliers.*

7.3. At http://www.statsci.org/data/general/kittiwak.html, you can find a dataset collected by D.K. Cairns in 1988 measuring the area available for a seabird (black-legged kittiwake) colony and the number of breeding pairs for a variety of different colonies. Figure 7.21 shows a linear regression of the number of breeding pairs against the log of area. There are 22 data points.

(a) Write $e_i = y_i - \mathbf{x}_i^T \beta$ for the residual. What is the $\mathsf{mean}\,(\{e\})$ for this regression?

(b) For this regression, $\mathsf{var}\,(\{y\}) = 16491357$ and the R^2 is 0.31. What is $\mathsf{var}\,(\{e\})$ for this regression?

(c) How well does the regression explain the data? If you had a large island, to what extent would you trust the prediction for the number of kittiwakes produced by this regression? If you had a small island, would you trust the answer more? Why?

(d) Figure 7.21 shows the result of a linear regression that ignores two likely outliers. Would you trust the predictions of this regression more? Why?

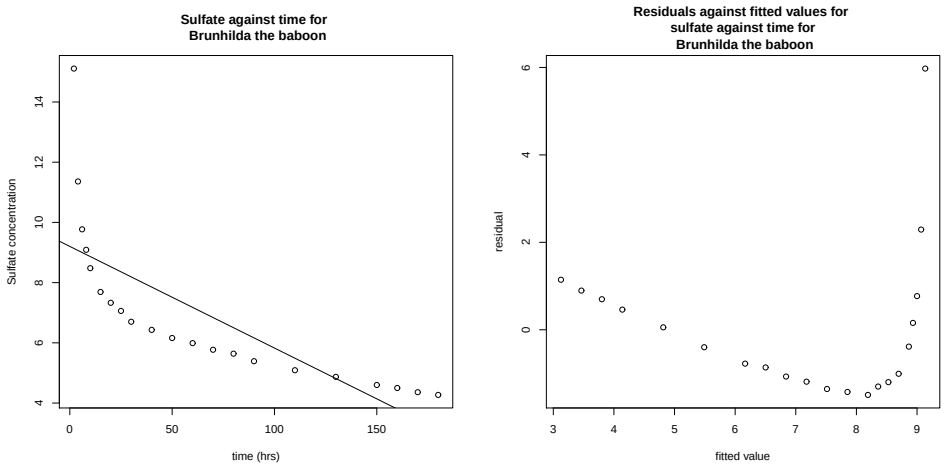


FIGURE 7.22: **Left:** *A regression of the concentration of sulfate in the blood of Brunhilda the baboon against time.* **Right:** *For this regression, a plot of residual against fitted value.*

7.4. At http://www.statsci.org/data/general/brunhild.html, you will find a dataset that measures the concentration of a sulfate in the blood of a baboon named Brunhilda as a function of time. Figure 7.22 plots this data, with a linear regression of the concentration against time. I have shown the data, and also a plot of the residual against the predicted value. The regression appears to be unsuccessful.

(a) What suggests the regression has problems?

(b) What is the cause of the problem, and why?

(c) What could you do to improve the problems?

7.5. Assume we have a dataset where $\mathbf{Y} = \mathcal{X}\beta + \xi$, for some unknown β and ξ. The term ξ is a normal random variable with zero mean, and covariance $\sigma^2\mathcal{I}$ (i.e. this data really does follow our model).

(a) Write $\hat{\beta}$ for the estimate of β recovered by least squares, and $\hat{\mathbf{Y}}$ for the values predicted by our model for the training data points. Show that

$$\hat{\mathbf{Y}} = \mathcal{X}\left(\mathcal{X}^T\mathcal{X}\right)^{-1}\mathcal{X}^T\mathbf{Y}$$

(b) Show that

$$\mathbb{E}[\hat{y}_i - y_i] = 0$$

for each training data point y_i, where the expectation is over the probability distribution of ξ.

(c) Show that

$$\mathbb{E}\left[(\hat{\beta} - \beta)\right] = 0$$

where the expectation is over the probability distribution of ξ.

7.6. In this exercise, I will show that the prediction process of chapter **??**(see page **??**) is a linear regression with two independent variables. Assume we have N data items which are 2-vectors $(x_1, y_1), \ldots, (x_N, y_N)$, where $N > 1$. These could be obtained, for example, by extracting components from larger vectors. As usual, we will write $\hat{x}_i$ for x_i in normalized coordinates, and so on. The correlation coefficient is r (this is an important, traditional notation).

(a) Assume that we have an x_o, for which we wish to predict a y value. Show that the value of the prediction obtained using the method of page **??** is

$$\begin{aligned} y_{\text{pred}} &= \frac{\mathsf{std}\,(y)}{\mathsf{std}\,(x)} r(x_0 - \mathsf{mean}\,(\{x\})) + \mathsf{mean}\,(\{y\}) \\ &= \left(\frac{\mathsf{std}\,(y)}{\mathsf{std}\,(x)} r\right) x_0 + \left(\mathsf{mean}\,(\{y\}) - \frac{\mathsf{std}\,(x)}{\mathsf{std}\,(y)}\mathsf{mean}\,(\{x\})\right). \end{aligned}$$

(b) Show that

$$\begin{aligned} r &= \frac{\mathsf{mean}\,(\{(x - \mathsf{mean}\,(\{x\}))(y - \mathsf{mean}\,(\{y\}))\})}{\mathsf{std}\,(x)\mathsf{std}\,(y)} \\ &= \frac{\mathsf{mean}\,(\{xy\}) - \mathsf{mean}\,(\{x\})\mathsf{mean}\,(\{y\})}{\mathsf{std}\,(x)\mathsf{std}\,(y)}. \end{aligned}$$

(c) Now write

$$\mathcal{X} = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \ldots & \ldots \\ x_n & 1 \end{pmatrix} \text{ and } \mathbf{Y} = \begin{pmatrix} y_1 \\ y_2 \\ \ldots \\ y_n \end{pmatrix}.$$

The coefficients of the linear regression will be $\hat{\beta}$, where $\mathcal{X}^T\mathcal{X}\hat{\beta} = \mathcal{X}^T\mathbf{Y}$. Show that

$$\begin{aligned} \mathcal{X}^T\mathcal{X} &= N\begin{pmatrix} \mathsf{mean}\left(\left\{x^2\right\}\right) & \mathsf{mean}\,(\{x\}) \\ \mathsf{mean}\,(\{x\}) & 1 \end{pmatrix} \\ &= N\begin{pmatrix} \mathsf{std}\,(x)^2 + \mathsf{mean}\,(\{x\})^2 & \mathsf{mean}\,(\{x\}) \\ \mathsf{mean}\,(\{x\}) & 1 \end{pmatrix} \end{aligned}$$

(d) Now show that

$$\begin{aligned}\mathcal{X}^T\mathbf{Y} &= N\begin{pmatrix} \mathsf{mean}\left(\{xy\}\right) \\ \mathsf{mean}\left(\{y\}\right)\end{pmatrix} \\ &= N\begin{pmatrix} \mathsf{std}\left(x\right)\mathsf{std}\left(y\right)r + \mathsf{mean}\left(\{x\}\right)\mathsf{mean}\left(\{y\}\right) \\ \mathsf{mean}\left(\{y\}\right)\end{pmatrix}.\end{aligned}$$

(e) Now show that

$$\left(\mathcal{X}^T\mathcal{X}\right)^{-1} = \frac{1}{N}\frac{1}{\mathsf{std}\left(x\right)^2}\begin{pmatrix} 1 & -\mathsf{mean}\left(\{x\}\right) \\ -\mathsf{mean}\left(\{x\}\right) & \mathsf{std}\left(x\right)^2 + \mathsf{mean}\left(\{x\}\right)^2\end{pmatrix}$$

(f) Now (finally!) show that if $\hat{\beta}$ is the solution to $\mathcal{X}^T\mathcal{X}\hat{\beta} - \mathcal{X}^T\mathbf{Y} = 0$, then

$$\hat{\beta} = \begin{pmatrix} r\frac{\mathsf{std}(y)}{\mathsf{std}(x)} \\ \mathsf{mean}\left(\{y\}\right) - \left(r\frac{\mathsf{std}(y)}{\mathsf{std}(x)}\right)\mathsf{mean}\left(\{x\}\right)\end{pmatrix}$$

and use this to argue that the process of page **??** is a linear regression with two independent variables.

7.7. This exercise investigates the effect of correlation on a regression. Assume we have N data items, $(\mathbf{x}_i, y_i)$. We will investigate what happens when the data have the property that the first component is relatively accurately predicted by the other components. Write x_{i1} for the first component of $\mathbf{x}_i$, and $\mathbf{x}_{i,\hat{1}}$ for the vector obtained by deleting the first component of $\mathbf{x}_i$. Choose $\mathbf{u}$ to predict the first component of the data from the rest *with minimum error*, so that $x_{i1} = \mathbf{x}_{i\hat{1}}^T\mathbf{u} + w_i$. The error of prediction is w_i. Write $\mathbf{w}$ for the vector of errors (i.e. the i'th component of $\mathbf{w}$ is w_i). Because $\mathbf{w}^T\mathbf{w}$ is minimized by choice of $\mathbf{u}$, we have $\mathbf{w}^T\mathbf{1} = 0$ (i.e. the average of the w_i's is zero). Assume that these predictions are very good, so that there is some small positive number ϵ so that $\mathbf{w}^T\mathbf{w} \leq \epsilon$.

(a) Write $\mathbf{a} = [-1, \mathbf{u}]^T$. Show that

$$\mathbf{a}^T\mathcal{X}^T\mathcal{X}\mathbf{a} \leq \epsilon.$$

(b) Now show that the smallest eigenvalue of $\mathcal{X}^T\mathcal{X}$ is less than or equal to ϵ.

(c) Write $s_k = \sum_u x_{uk}^2$, and $s\mathrm{max}$ for $\max(s_1, \ldots, s_d)$. Show that the largest eigenvalue of $\mathcal{X}^T\mathcal{X}$ is greater than or equal to $s\mathrm{max}$.

(d) The **condition number** of a matrix is the ratio of largest to smallest eigenvalue of a matrix. Use the information above to bound the **condition number** of $\mathcal{X}^T\mathcal{X}$.

(e) Assume that $\hat{\beta}$ is the solution to $\mathcal{X}^T\mathcal{X}\hat{\beta} = \mathcal{X}^T\mathbf{Y}$. Show that the

$$(\mathcal{X}^T\mathbf{Y} - \mathcal{X}^T\mathcal{X}(\hat{\beta} + \mathbf{a}))^T(\mathcal{X}^T\mathbf{Y} - \mathcal{X}^T\mathcal{X}(\hat{\beta} + \mathbf{a}))$$

(for $\mathbf{a}$ as above) is bounded above by

$$\epsilon^2(1 + \mathbf{u}^T\mathbf{u})$$

(f) Use the last sub exercises to explain why correlated data will lead to a poor estimate of $\hat{\beta}$.

7.8. This exercise explores the effect of regularization on a regression. Assume we have N data items, $(\mathbf{x}_i, y_i)$. We will investigate what happens when the data have the property that the first component is relatively accurately predicted by the other components. Write x_{i1} for the first component of $\mathbf{x}_i$, and $\mathbf{x}_{i,\hat{1}}$ for the vector obtained by deleting the first component of $\mathbf{x}_i$. Choose $\mathbf{u}$ to predict the first component of the data from the rest *with minimum error*, so that $x_{i1} = \mathbf{x}_{i\hat{1}}^T\mathbf{u} + w_i$. The error of prediction is w_i. Write $\mathbf{w}$ for the vector of errors (i.e. the i'th component of $\mathbf{w}$ is w_i). Because $\mathbf{w}^T\mathbf{w}$ is minimized by choice of $\mathbf{u}$, we have $\mathbf{w}^T\mathbf{1} = 0$ (i.e. the average of the w_i's is zero). Assume that these predictions are very good, so that there is some small positive number ϵ so that $\mathbf{w}^T\mathbf{w} \leq \epsilon$.

(a) Show that, *for any vector* $\mathbf{v}$,

$$\mathbf{v}^T\left(\mathcal{X}^T\mathcal{X} + \lambda\mathcal{I}\right)\mathbf{v} \geq \lambda\mathbf{v}^T\mathbf{v}$$

and use this to argue that the smallest eigenvalue of $\left(\mathcal{X}^T\mathcal{X} + \lambda\mathcal{I}\right)$ is greater than λ.

(b) Write $\mathbf{b}$ for an eigenvector of $\mathcal{X}^T\mathcal{X}$ with eigenvalue $\lambda_{\mathbf{b}}$. Show that $\mathbf{b}$ is an eigenvector of $\left(\mathcal{X}^T\mathcal{X} + \lambda\mathcal{I}\right)$ with eigenvalue $\lambda_{\mathbf{b}} + \lambda$.

(c) Recall $\mathcal{X}^T\mathcal{X}$ is a $d \times d$ matrix which is symmetric, and so has d orthonormal eigenvectors. Write $\mathbf{b}_i$ for the i'th such vector, and $\lambda_{\mathbf{b}_i}$ for the corresponding eigenvalue. Show that

$$\mathcal{X}^T\mathcal{X}\beta - \mathcal{X}^T\mathbf{Y} = 0$$

is solved by

$$\beta = \sum_{i=1}^{d} \frac{\mathbf{Y}^T\mathcal{X}\mathbf{b}_i}{\lambda_{\mathbf{b}_i}}.$$

(d) Using the notation of the previous sub exercise, show that

$$(\mathcal{X}^T\mathcal{X} + \lambda\mathcal{I})\beta - \mathcal{X}^T\mathbf{Y} = 0$$

is solved by

$$\beta = \sum_{i=1}^{d} \frac{\mathbf{Y}^T\mathcal{X}\mathbf{b}_i}{\lambda_{\mathbf{b}_i} + \lambda}.$$

Use this expression to explain why a regularized regression may produce better results on test data than an unregularized regression.

PROGRAMMING EXERCISES

7.9. At http://www.statsci.org/data/general/brunhild.html, you will find a dataset that measures the concentration of a sulfate in the blood of a baboon named Brunhilda as a function of time. Build a linear regression of the log of the concentration against the log of time.

(a) Prepare a plot showing (a) the data points and (b) the regression line in log-log coordinates.

(b) Prepare a plot showing (a) the data points and (b) the regression curve in the original coordinates.

(c) Plot the residual against the fitted values in log-log and in original coordinates.
(d) Use your plots to explain whether your regression is good or bad and why.

7.10. At http://www.statsci.org/data/oz/physical.html, you will find a dataset of measurements by M. Larner, made in 1996. These measurements include body mass, and various diameters. Build a linear regression of predicting the body mass from these diameters.
(a) Plot the residual against the fitted values for your regression.
(b) Now regress the cube root of mass against these diameters. Plot the residual against the fitted values in both these cube root coordinates and in the original coordinates.
(c) Use your plots to explain which regression is better.

7.11. At https://archive.ics.uci.edu/ml/datasets/Abalone, you will find a dataset of measurements by W. J. Nash, T. L. Sellers, S. R. Talbot, A. J. Cawthorn and W. B. Ford, made in 1992. These are a variety of measurements of blacklip abalone (*Haliotis rubra*; delicious by repute) of various ages and genders.
(a) Build a linear regression predicting the age from the measurements, ignoring gender. Plot the residual against the fitted values.
(b) Build a linear regression predicting the age from the measurements, including gender. There are three levels for gender; I'm not sure whether this has to do with abalone biology or difficulty in determining gender. You can represent gender numerically by choosing 1 for one level, 0 for another, and -1 for the third. Plot the residual against the fitted values.
(c) Now build a linear regression predicting the log of age from the measurements, ignoring gender. Plot the residual against the fitted values.
(d) Now build a linear regression predicting the log age from the measurements, including gender, represented as above. Plot the residual against the fitted values.
(e) It turns out that determining the age of an abalone is possible, but difficult (you section the shell, and count rings). Use your plots to explain which regression you would use to replace this procedure, and why.
(f) Can you improve these regressions by using a regularizer? Use glmnet to obtain plots of the cross-validated prediction error.

CHAPTER 8

Regression: Choosing and Managing Models

8.1 MODEL SELECTION: WHICH MODEL IS BEST?

It is usually quite easy to have many explanatory variables in a regression problem. Even if you have only one measurement, you could always compute a variety of non-linear functions of that measurement. As we have seen, inserting variables into a model will reduce the fitting cost, but that doesn't mean that better predictions will result (section 7.4.1). We need to choose which explanatory variables we will use. A linear model with few explanatory variables may make poor predictions because the model itself is incapable of representing the independent variable accurately (an effect known as bias). A linear model with many explanatory variables may make poor predictions because we can't estimate the coefficients well (an effect known as variance). Choosing which explanatory variables we will use (and so which model we will use) requires that we balance these effects, described in greater detail in section 8.1.1. In the following sections, we describe straightforward methods of doing so.

8.1.1 Bias and Variance

We now look at the process of finding a model in a fairly abstract way. Doing so makes plain three distinct and important effects that cause models to make predictions that are wrong. One is **irreducible error**. Even a perfect choice of model can make mistaken predictions, because more than one prediction could be correct for the same $\mathbf{x}$. Another way to think about this is that there could be many future data items, all of which have the same $\mathbf{x}$, but each of which has a different y. In this case some of our predictions must be wrong, and the effect is unavoidable.

A second effect is **bias**. We must use some collection of models. Even the best model in the collection may not be capable of predicting all the effects that occur in the data. Errors that are caused by the best model still not being able to predict the data accurately are attributed to bias.

The third effect is **variance**. We must choose our model from the collection of models. The model we choose is unlikely to be the best model. This might occur, for example, because our estimates of the parameters aren't exact because we have a limited amount of data. Errors that are caused by our choosing a model that is not the best in the family are attributed to variance.

All this can be written out in symbols. We have a vector of predictors $\mathbf{x}$, and a random variable Y. At any given point $\mathbf{x}$, we have

$$Y = f(\mathbf{x}) + \xi$$

where ξ is noise and f is an unknown function. We have $\mathbb{E}[\xi] = 0$, and $\mathbb{E}\left[\xi^2\right] = \mathsf{var}\left(\{\xi\}\right) = \sigma_\xi^2$; furthermore, ξ is independent of X. We have some procedure that takes a selection of training data, consisting of pairs $(\mathbf{x}_i, y_i)$, and selects a model $\hat{f}$. We will use this model to predict values for future $\mathbf{x}$. It is highly unlikely that $\hat{f}$ is the same as f; assuming that it is involves assuming that we can perfectly estimate the best model with a finite dataset, which doesn't happen.

We need to understand the error that will occur when we use $\hat{f}$ to predict for some data item that isn't in the training set. This is the error that we will encounter in practice. The error at any point $\mathbf{x}$ is

$$\mathbb{E}\left[(Y - \hat{f}(\mathbf{X}))^2\right]$$

where the expectation is taken over $P(Y, \text{training data}|\mathbf{x})$. But the new query point $\mathbf{x}$ does not depend on the training data, and so the distribution is $P(Y|\mathbf{x}) \times P(\text{training data})$. The expectation can be written in an extremely useful form. Recall $\mathsf{var}\left(\{U\}\right) = \mathbb{E}\left[U^2\right] - \mathbb{E}[U]^2$. This means we have

$$\begin{aligned}\mathbb{E}\left[(Y - \hat{f}(\mathbf{x}))^2\right] &= \mathbb{E}\left[Y^2\right] - 2\mathbb{E}\left[Y\hat{f}\right] + \mathbb{E}\left[\hat{f}^2\right] \\ &= \mathsf{var}\left(\{Y\}\right) + \mathbb{E}[Y]^2 - 2\mathbb{E}\left[Y\hat{f}\right] + \mathsf{var}\left(\left\{\hat{f}\right\}\right) + \mathbb{E}\left[\hat{f}\right]^2.\end{aligned}$$

Now $Y = f(X) + \xi$, $\mathbb{E}[\xi] = 0$, and ξ is independent of X so we have $\mathbb{E}[Y] = \mathbb{E}[f]$ and $\mathsf{var}\left(\{Y\}\right) = \mathsf{var}\left(\{\xi\}\right) = \sigma_\xi^2$. This yields

$$\begin{aligned}\mathbb{E}\left[(Y - \hat{f}(\mathbf{x}))^2\right] &= \mathsf{var}\left(\{Y\}\right) + \mathbb{E}[f]^2 - 2\mathbb{E}\left[f\hat{f}\right] + \mathsf{var}\left(\left\{\hat{f}\right\}\right) + \mathbb{E}\left[\hat{f}\right]^2 \\ &= \sigma_\xi^2 + \mathbb{E}\left[(f - \hat{f})^2\right] + \mathsf{var}\left(\left\{\hat{f}\right\}\right) \\ &= \sigma_\xi^2 + (f - \mathbb{E}\left[\hat{f}\right])^2 + \mathsf{var}\left(\left\{\hat{f}\right\}\right) \ (f \text{ isn't random}).\end{aligned}$$

The expected error on all future data is the sum of three terms. The irreducible error is σ_ξ^2; even the true model must produce this error, on average. The best model to choose would be $\mathbb{E}\left[\hat{f}\right]$ (remember, the expectation is over choices of training data; this model would be the one that best represented all possible attempts to train). But we don't have $\mathbb{E}\left[\hat{f}\right]$. Instead, we have $\hat{f}$. The variance is $\mathsf{var}\left(\left\{\hat{f}\right\}\right) = \mathbb{E}\left[(\hat{f} - \mathbb{E}\left[\hat{f}\right])^2\right]$. This term represents the fact that the model we chose ($\hat{f}$) is different from the mean model ($\mathbb{E}\left[\hat{f}\right]$). The difference arises because our training data is a subset of all data, and our model is chosen to be good on the training data, rather than on every possible training set. The bias is $(f - \mathbb{E}\left[\hat{f}\right])^2$. This term reflects the fact that even the best choice of model ($\mathbb{E}\left[\hat{f}\right]$) may not be the same as the true source of data ($\mathbb{E}[f]$ which is the same as f, because f is deterministic).

There is usually a tradeoff between bias and variance. Generally, when a model comes from a "small" or "simple" family, we expect that (a) we can estimate

the best model in the family reasonably accurately (so the variance will be low) but (b) the model may have real difficulty reproducing the data (meaning the bias is large). Similarly, if the model comes from a "large" or "complex" family, the variance is likely to be high (because it will be hard to estimate the best model in the family accurately) but the bias will be low (because the model can more accurately reproduce the data). All modelling involves managing this tradeoff between bias and variance. I am avoiding being precise about the complexity of a model because it can be tricky to do. One reasonable proxy is the number of parameters we have to estimate to determine the model.

You can see a crude version this tradeoff in the perch example of section 7.4.1 and Figure 7.11. Recall that, as I added monomials to the regression of weight against length, the fitting error went down; but the model that uses length^{10} as an explanatory variable makes very odd predictions away from the training data. When I use low degree monomials, the dominant source of error is bias; and when I use high degree monomials, the dominant source of error is variance. A common mistake is to feel that the major difficulty is bias, and so to use extremely complex models. Usually the result is poor estimates of model parameters, and huge variance. Experienced modellers fear variance far more than they fear bias.

The bias-variance discussion suggests it isn't a good idea simply to use all the explanatory variables that you can obtain (or think of). Doing so might lead to a model with serious variance problems. Instead, we must choose a model that uses a subset of the explanatory variables that is small enough to control variance, and large enough that the bias isn't a problem. We need some strategy to choose explanatory variables. The simplest (but by no means the best; we'll see better in this chapter) approach is to search sets of explanatory variables for a good set. The main difficulty is knowing when you have a good set.

8.1.2 Choosing a Model using Penalties: AIC and BIC

We would like to choose one of a set of models. We cannot do so using just the training error, because more complex models will tend to have lower training error, and so the model with the lowest training error will tend to be the most complex model. Training error is a poor guide to test error, because lower training error is evidence of lower bias on the models part; but with lower bias, we expect to see greater variance, and the training error doesn't take that into account.

One strategy is to penalize the model for complexity. We add some penalty, reflecting the complexity of the model, to the training error. We then expect to see the general behavior of figure 8.1. The training error goes down, and the penalty goes up as the model gets more complex, so we expect to see a point where the sum is at a minimum.

There are a variety of ways of constructing penalties. **AIC** (short for An Information Criterion) is a method due originally to Akaike, in ****. Rather than using the training error, AIC uses the maximum value of the log-likelihood of the model. Write $\mathcal{L}$ for this value. Write k for the number of parameters estimated to fit the model. Then the AIC is

$$2k - 2\mathcal{L}$$

and a better model has a smaller value of AIC (remember this by remembering

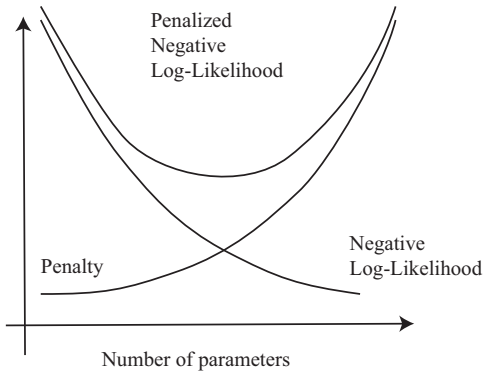


FIGURE 8.1: *When we add explanatory variables (and so parameters) to a model, the value of the negative log-likelihood of the best model can't go up, and usually goes down. This means that we cannot use the value as a guide to how many explanatory variables there should be. Instead, we add a penalty that increases as a function of the number of parameters, and search for the model that minimizes the sum of negative long-likelihood and penalty. AIC and BIC grow linearly with the number of parameters, but I am following the usual convention of plotting the penalty as a curve rather than a straight line.*

that a larger log-likelihood corresponds to a better model). Estimating AIC is straightforward for regression models if you assume that the noise is a zero mean normal random variable. You estimate the mean-squared error, which gives the variance of the noise, and so the log-likelihood of the model. You do have to keep track of two points. First, k is the total number of parameters estimated to fit the model. For example, in a linear regression model, where you model y as $\mathbf{x}^T\beta + \xi$, you need to estimate d parameters to estimate $\hat{\beta}$ *and* the variance of ξ (to get the log-likelihood). So in this case $k = d + 1$. Second, log-likelihood is usually only known up to a constant, so that different software implementations often use different constants. This is wildly confusing when you don't know about it (why would `AIC` and `extractAIC` produce different numbers on the same model?) but of no real significance – you're looking for the smallest value of the number, and the actual value doesn't mean anything. Just be careful to compare only numbers computed with the same routine.

An alternative is **BIC** (Bayes' Information Criterion), given by

$$2k \log N - 2\mathcal{L}$$

(where N is the size of the training data set). You will often see this written as $2\mathcal{L} - 2k \log N$; I have given the form above so that one always wants the smaller value as with AIC. There is a considerable literature comparing AIC and BIC. AIC has a mild reputation for overestimating the number of parameters required, but is often argued to have firmer theoretical foundations.

Worked example 8.1 *AIC and BIC*

Write M_d for the model that predicts weight from length for the perch dataset as $\sum_{j=0}^{j=d} \beta_j \text{length}^j$. Choose an appropriate value of $d \in [1, 10]$ using AIC and BIC.

Solution: I used the R functions `AIC` and `BIC`, and got the table below.

	1	2	3	4	5	6	7	8	9	10
AIC	677	617	617	613	615	617	617	612	613	614
BIC	683	625	627	625	629	633	635	633	635	638

The best model by AIC has (rather startlingly!) $d = 8$. One should not take small differences in AIC too seriously, so models with $d = 4$ and $d = 9$ are fairly plausible, too. BIC suggests $d = 2$.

8.1.3 Choosing a Model using Cross-Validation

AIC and BIC are estimates of error on future data. An alternative is to measure this error on held out data, using a cross-validation strategy (as in section **??**). One splits the training data into F **folds**, where each data item lies in exactly one fold. The case $F = N$ is sometimes called "leave-one-out" cross-validation. One then sets aside one fold in turn, fitting the model to the remaining data, and evaluating the model error on the left-out fold. The model error is then averaged. This process gives us an estimate of the performance of a model on held-out data. Numerous variants are available, particularly when lots of computation and lots of data are available. For example: one might not average over all folds; one might use fewer or more folds; and so on.

Worked example 8.2 *Cross-validation*

Write M_d for the model that predicts weight from length for the perch dataset as $\sum_{j=0}^{j=d} \beta_j \text{length}^j$. Choose an appropriate value of $d \in [1, 10]$ using leave-one-out cross validation.

Solution: I used the R functions `CVlm`, which takes a bit of getting used to. There is sample code in the exercises section. I found:

1	2	3	4	5	6	7	8	9	10
1.9e4	4.0e3	7.2e3	4.5e3	6.0e3	5.6e4	1.2e6	4.0e6	3.9e6	1.9e8

where the best model is $d = 2$.

8.1.4 A Search Process: Forward and Backward Stagewise Regression

Assume we have a set of explanatory variables and we wish to build a model, choosing some of those variables for our model. Our explanatory variables could be many distinct measurements, or they could be different non-linear functions of

the same measurement, or a combination of both. We can evaluate models relative to one another fairly easily (AIC, BIC or cross-validation, your choice). However, choosing which set of explanatory variables to use can be quite difficult, because there are so many sets. The problem is that you cannot predict easily what adding or removing an explanatory variable will do. Instead, when you add (or remove) an explanatory variable, the errors that the model makes change, and so the usefulness of all other variables changes too. This means that (at least in principle) you have to look at every subset of the explanatory variables. Imagine you start with a set of F possible explanatory variables (including the original measurement, and a constant). You don't know how many to use, so you might have to try every different group, of each size, and there are far too many groups to try. There are two useful alternatives.

In **forward stagewise regression**, you start with an empty working set of explanatory variables. You then iterate the following process. For each of the explanatory variables not in the working set, you construct a new model using the working set and that explanatory variable, and compute the model evaluation score. If the best of these models has a better score than the model based on the working set, you insert the appropriate variable into the working set and iterate. If no variable improves the working set, you decide you have the best model and stop. This is fairly obviously a greedy algorithm.

Backward stagewise regression is pretty similar, but you start with a working set containing all the variables, and remove variables one-by-one and greedily. As usual, greedy algorithms are very helpful but not capable of exact optimization. Each of these strategies can produce rather good models, but neither is guaranteed to produce the best model.

8.1.5 Significance: What Variables are Important?

Imagine you regress some measure of risk of death against blood pressure, whether someone smokes or not, and the length of their thumb. Because high blood pressure and smoking tend to increase risk of death, you would expect to see "large" coefficients for these explanatory variables. Since changes in the thumb length have no effect, you would expect to see "small" coefficients for these explanatory variables. This suggests a regression can be used to determine what effects are important in building a model.

One difficulty is the result of sampling variance. Imagine that we have an explanatory variable that has absolutely no relationship to the dependent variable. If we had an arbitrarily large amount of data, and could exactly identify the correct model, we'd find that, in the correct model, the coefficient of that variable was zero. But we don't have an arbitrarily large amount of data. Instead, we have a sample of data. Hopefully, our sample is random, so that the reasoning of section 10.1 can be applied. Using that reasoning, our estimate of the coefficient is the value of a random variable whose expected value is zero, but whose variance isn't. As a result, we are very unlikely to see a zero. This reasoning applies to each coefficient of the model. To be able to tell which ones are small, we would need to know the standard deviation of each, so we can tell whether the value we observe is a small number of standard deviations away from zero. This line of reasoning is very like hypothesis

testing. It turns out that the sampling variance of regression coefficients can be estimated in a straightforward way. In turn, we have an estimate of the extent to which their difference from zero could be a result of random sampling. R will produce this information routinely; use `summary` on the output of `lm`.

A second difficulty has to do with practical significance, and is rather harder. We could have explanatory variables that are genuinely linked to the independent variable, but might not matter very much. This is a common phenomenon, particularly in medical statistics. It requires considerable care to disentangle some of these issues. Here is an example. Bowel cancer is an unpleasant disease, which could kill you. Being screened for bowel cancer is at best embarrassing and unpleasant, and involves some startling risks. There is considerable doubt, from reasonable sources, about whether screening has value and if so, how much (as a start point, you could look at Ransohoff DF. How Much Does Colonoscopy Reduce Colon Cancer Mortality?. Ann Intern Med. 2009). There is some evidence linking eating red or processed meat to incidence of bowel cancer. A good practical question is: should one abstain from eating red or processed meat based on increased bowel cancer risk?

Coming to an answer is tough; the coefficient in any regression is clearly not zero, but it's pretty small as these numbers indicate. The UK population in 2012 was 63.7 million (this is a summary figure from Google, using World Bank data; there's no reason to believe that it's significantly wrong). I obtained the following figures from the UK cancer research institute website, at http://www.cancerresearchuk.org/health-professional/cancer-statistics/statistics-by-cancer-type/bowel-cancer. There were 41, 900 new cases of bowel cancer in the UK in 2012. Of these cases, 43% occurred in people aged 75 or over. 57% of people diagnosed with bowel cancer survive for ten years or more after diagnosis. Of diagnosed cases, an estimated 21% are linked to eating red or processed meat, and the best current estimate is that the risk of incidence is between 17% and 30% higher per 100g of red meat eaten per day (i.e. if you eat 100g of red meat per day, your risk increases by some number between 17% and 30%; 200g a day gets you twice that number; and – rather roughly – so on). These numbers are enough to confirm that there is a non-zero coefficient linking the amount of red or processed meat in your diet with your risk of bowel cancer (though you'd have a tough time estimating the exact value of that coefficient from the information here). If you eat more red meat, your risk of dying of bowel cancer really will go up. But the numbers I gave above suggest that (a) it won't go up much and (b) you might well die rather late in life, where the chances of dying of something are quite strong. The coefficient linking eating red meat and bowel cancer is clearly pretty small, because the incidence of the disease is about 1 in 1500 per year. Does it matter? you get to choose, and your choice has consequences.

8.2 ROBUST REGRESSION

We have seen that outlying data points can result in a poor model. This is caused by the squared error cost function: squaring a large error yields an enormous number. One way to resolve this problem is to identify and remove outliers before fitting a model. This can be difficult, because it can be hard to specify precisely when

a point is an outlier. Worse, in high dimensions most points will look somewhat like outliers, and we may end up removing all most all the data. The alternative solution I offer here is to come up with a cost function that is less susceptible to problems with outliers. The general term for a regression that can ignore some outliers is a **robust regression**.

8.2.1 M-Estimators and Iteratively Reweighted Least Squares

One way to reduce the effect of outliers on a least-squares solution would be to weight each point in the cost function. We need some method to estimate an appropriate set of weights. This would use a large weight for errors at points that are "trustworthy", and a low weight for errors at "suspicious" points.

We can obtain such weights using an **M-estimator**, which estimates parameters by replacing the negative log-likelihood with a term that is better behaved. In our examples, the negative log-likelihood has always been squared error. Write β for the parameters of the model being fitted, and $r_i(\mathbf{x}_i, \beta)$ for the residual error of the model on the ith data point. For us, r_i will always be $y_i - \mathbf{x}_i^T\beta$. So rather than minimizing

$$\sum_i (r_i(\mathbf{x}_i, \beta))^2$$

as a function of β, we will minimize an expression of the form

$$\sum_i \rho(r_i(\mathbf{x}_i, \beta); \sigma),$$

for some appropriately chosen function ρ. Clearly, our negative log-likelihood is one such estimator (use $\rho(u;\sigma) = u^2$). The trick to M-estimators is to make $\rho(u;\sigma)$ look like u^2 for smaller values of u, but ensure that it grows more slowly than u^2 for larger values of u.

The **Huber loss** is one important M-estimator. We use

$$\rho(u;\sigma) = \begin{cases} \frac{u^2}{2} & |u| < \sigma \\ \sigma|u| - \frac{\sigma^2}{2} & \end{cases}$$

which is the same as u^2 for $-\sigma \leq u \leq \sigma$, and then switches to $|u|$ for larger (or smaller) σ (Figure **??**). The Huber loss is convex (meaning that there will be a unique minimum for our models) and differentiable, but its derivative is not continuous. The choice of the parameter σ (which is known as **scale**) has an effect on the estimate. You should interpret this parameter as the distance that a point can lie from the fitted function while still being seen as an **inlier** (anything that isn't even partially an outlier).

Generally, M-estimators are discussed in terms of their **influence function**. This is

$$\frac{\partial \rho}{\partial u}.$$

Its importance becomes evidence when we consider algorithms to fit $\hat{\beta}$ using an

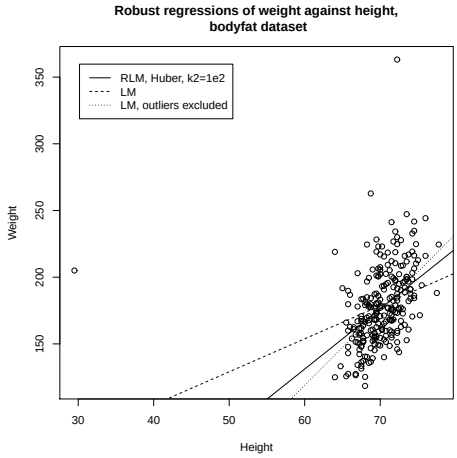


FIGURE 8.2: *Comparing three different linear regression strategies on the bodyfat data, regressing weight against height. Notice that using an M-estimator gives an answer very like that obtained by rejecting outliers by hand. The answer may well be "better" because it isn't certain that each of the four points rejected is an outlier, and the robust method may benefit from some of the information in these points. I tried a range of scales for the Huber loss (the 'k2' parameter), but found no difference in the line resulting over scales varying by a factor of 1e4, which is why I plot only one scale.*

M-estimator. Our minimization criterion is

$$\begin{aligned}\nabla_\beta \left(\sum_i \rho(y_i - \mathbf{x}_i^T\beta; \sigma) \right) &= \sum_i \left[\frac{\partial \rho}{\partial u}(y_i - \mathbf{x}_i^T\beta; \sigma) \right] (-\mathbf{x}_i) \\ &= 0.\end{aligned}$$

Now write $w_i(\beta)$ for

$$\frac{\frac{\partial \rho}{\partial u}(y_i - \mathbf{x}_i^T\beta; \sigma)}{y_i - \mathbf{x}_i^T\beta}.$$

We can write the minimization criterion as

$$\sum_i \left[w_i(\beta) \right] (y_i - \mathbf{x}_i^T\beta)(-\mathbf{x}_i) = 0.$$

Now write $\mathcal{W}(\beta)$ for the diagonal matrix whose i'th diagonal entry is $w_i(\beta)$ Then our fitting criterion is equivalent to

$$\mathcal{X}^T \left[\mathcal{W}(\beta) \right] \mathbf{Y} = \mathcal{X}^T \left[\mathcal{W}(\beta) \right] \mathcal{X}\beta.$$

The difficulty in solving this is that $w_i(\beta)$ depend on β, so we can't just solve a linear system in β. We could use the following strategy. Use $\mathcal{W}$ tries to downweight points that are suspiciously inconsistent with our current estimate of β, then update

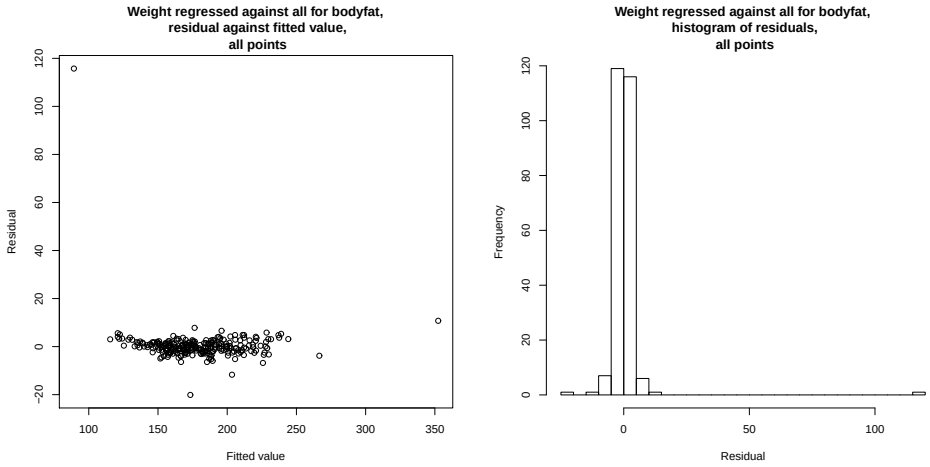


FIGURE 8.3: *A robust linear regression of weight against all variables for the bodyfat dataset, using the Huber loss and all data points. On the* **left**, *residual plotted against fitted value (the residual is not standardized). Notice that there are some points with very large residual, but most have much smaller residual; this wouldn't happen with a squared error. On the* **right**, *a histogram of the residual. If one ignores the extreme residual values, this looks normal. The robust process has been able to discount the effect of the outliers, without us needing to identify and reject outliers by hand.*

β using those weights. The strategy is known as **iteratively reweighted least squares**, and is very effective.

We assume we have an estimate of the correct parameters $\hat{\beta}^{(n)}$, and consider updating it to $\hat{\beta}^{(n+1)}$. We compute

$$w_i^{(n)} = w_i(\hat{\beta}^{(n)}) = \frac{\frac{\partial \rho}{\partial u}(y_i - \mathbf{x}_i^T\beta^{(n)}; \sigma)}{y_i - \mathbf{x}_i^T\hat{\beta}^{(n)}}.$$

We then estimate $\hat{\beta}^{(n+1)}$ by solving

$$\mathcal{X}^T\mathcal{W}^{(n)}\mathbf{Y} = \mathcal{X}^T\mathcal{W}^{(n)}\mathcal{X}\hat{\beta}^{(n+1)}.$$

The key to this algorithm is finding good start points for the iteration. One strategy is randomized search. We select a small subset of points uniformly at random, and fit some $\hat{\beta}$ to these points, then use the result as a start point. If we do this often enough, one of the start points will be an estimate that is not contaminated by outliers.

8.2.2 Scale for M-Estimators

The estimators require a sensible estimate of σ, which is often referred to as **scale**. Typically, the scale estimate is supplied at each iteration of the solution method.

One reasonable estimate is the **MAD** or **median absolute deviation**, given by

$$\sigma^{(n)} = 1.4826 \text{ median}_i | r_i^{(n)}(x_i; \hat{\beta}^{(n-1)}) |.$$

Another a popular estimate of scale is obtained with **Huber's proposal 2** (that is what everyone calls it!). Choose some constant $k_1 > 0$, and define $\Xi(u) = \min(|u|, k_1)^2$. Now solve the following equation for σ:

$$\sum_i \Xi(\frac{r_i^{(n)}(x_i; \hat{\beta}^{(n-1)})}{\sigma}) = Nk_2$$

where k_2 is another constant, usually chosen so that the estimator gives the right answer for a normal distribution (exercises). This equation needs to be solved with an iterative method; the MAD estimate is the usual start point. R provides `hubers`, which will compute this estimate of scale (and figures out k_2 for itself). The choice of k_1 depends somewhat on how contaminated you expect your data to be. As $k_1 \to \infty$, this estimate becomes more like the standard deviation of the data.

8.3 GENERALIZED LINEAR MODELS

We have used a linear regression to predict a value from a feature vector, but implicitly have assumed that this value is a real number. Other cases are important, and some of them can be dealt with using quite simple generalizations of linear regression. When we derived linear regression, I said one way to think about the model was

$$y = \mathbf{x}^T\beta + \xi$$

where ξ was a normal random variable with zero mean and variance σ_ξ^2. Another way to write this is to think of y as the value of a random variable Y. In this case, Y has mean $\mathbf{x}^T\beta$ and variance σ_ξ^2. This can be written as

$$Y \sim N(\mathbf{x}^T\beta, \sigma_\xi^2).$$

This offers a fruitful way to generalize: we replace the normal distribution with some other parametric distribution, and predict the parameter using $\mathbf{x}^T\beta$. Two examples are particularly important.

8.3.1 Logistic Regression

Assume the y values can be either 0 or 1. You could think of this as a two class classification problem, and deal with it using an SVM. There are sometimes advantages to seeing it as a regression problem. One is that we get to see a new classification method that explicitly models class posteriors, which an SVM doesn't do.

We build the model by asserting that the y values represent a draw from a Bernoulli random variable (definition below, for those who have forgotten). The parameter of this random variable is θ, the probability of getting a one. But $0 \leq \theta \leq 1$, so we can't just model θ as $\mathbf{x}^T\beta$. We will choose some **link function** g so that we can model $g(\theta)$ as $\mathbf{x}^T\beta$. This means that, in this case, g must map the interval between 0 and 1 to the whole line, and must be 1-1. The link function maps θ to $\mathbf{x}^T\beta$; the direction of the map is chosen by convention. We build our model by asserting that $g(\theta) = \mathbf{x}^T\beta$.

Definition: 8.1 *Bernoulli random variable*

A Bernoulli random variable with parameter θ takes the value 1 with probability θ and 0 with probability $1-\theta$. This is a model for a coin toss, among other things.

Notice that, for a Bernoulli random variable, we have that

$$\log\left[\frac{P(y=1|\theta)}{P(y=0|\theta)}\right] = \log\left[\frac{\theta}{1-\theta}\right]$$

and the **logit function** $g(u) = \log\left[\frac{u}{1-u}\right]$ meets our needs for a link function (it maps the interval between 0 and 1 to the whole line, and is 1-1). This means we can build our model by asserting that

$$\log\left[\frac{P(y=1|\mathbf{x})}{P(y=0|\mathbf{x})}\right] = \mathbf{x}^T\beta$$

then solving for the β that maximizes the log-likelihood of the data. Simple manipulation yields

$$P(y=1|\mathbf{x}) = \frac{e^{\mathbf{x}^T\beta}}{1+e^{\mathbf{x}^T\beta}} \text{ and } P(y=0|\mathbf{x}) = \frac{1}{1+e^{\mathbf{x}^T\beta}}.$$

In turn, this means the log-likelihood of a dataset will be

$$\mathcal{L}(\beta) = \sum_i \left[\mathbb{I}_{[y=1]}(y_i)\mathbf{x}_i^T\beta - \log\left(1+e^{\mathbf{x}_i^T\beta}\right)\right].$$

You can obtain β from this log-likelihood by gradient ascent (or rather a lot faster by Newton's method, if you know that).

A regression of this form is known as a **logistic regression**. It has the attractive property that it produces estimates of posterior probabilities. Another interesting property is that a logistic regression is a lot like an SVM. To see this, we replace the labels with new ones. Write $\hat{y}_i = 2y_i - 1$; this means that $\hat{y}_i$ takes the values -1 and 1, rather than 0 and 1. Now $\mathbb{I}_{[y=1]}(y_i) = \frac{\hat{y}_i+1}{2}$, so we can write

$$\begin{aligned}
-\mathcal{L}(\beta) &= -\sum_i \left[\frac{\hat{y}_i+1}{2}\mathbf{x}_i^T\beta - \log\left(1+e^{\mathbf{x}_i^T\beta}\right)\right] \\
&= \sum_i \left[\frac{\hat{y}_i+1}{2}\mathbf{x}_i^T\beta - \log\left(1+e^{\mathbf{x}_i^T\beta}\right)\right] \\
&= \sum_i \left[\log\left(\frac{1+e^{\mathbf{x}_i^T\beta}}{e^{\frac{\hat{y}_i+1}{2}\mathbf{x}_i^T\beta}}\right)\right] \\
&= \sum_i \left[\log\left(e^{\frac{-(\hat{y}_i+1)}{2}\mathbf{x}_i^T\beta} + e^{\frac{1-\hat{y}_i}{2}\mathbf{x}_i^T\beta}\right)\right]
\end{aligned}$$

and we can interpret the term in square brackets as a loss function. If you plot it, you will notice that it behaves rather like the hinge loss. When $\hat{y}_i = 1$, if $\mathbf{x}^T\beta$ is positive the loss is very small, but if $\mathbf{x}^T\beta$ is strongly negative, the loss grows linearly in $\mathbf{x}^T\beta$. There is similar behavior when $\hat{y}_i = -1$. The transition is smooth, unlike the hinge loss. Logistic regression should (and does) behave well for the same reasons the SVM behaves well.

Be aware that logistic regression has one annoying quirk. When the data are linearly separable (i.e. there exists some β such that $y_i\mathbf{x}_i^T\beta > 0$ for all data items), logistic regression will behave badly. To see the problem, choose the β that separates the data. Now it is easy to show that increasing the magnitude of β will increase the log likelihood of the data; there isn't any limit. These situations arise fairly seldom in practical data.

8.3.2 Multiclass Logistic Regression

Imagine $y \in [0, 1, \ldots, C-1]$. Then it is natural to model $p(y|\mathbf{x})$ with a discrete probability distribution on these values. This can be specified by choosing $(\theta_0, \theta_1, \ldots, \theta_{C-1})$ where each term is between 0 and 1 and $\sum_i \theta_i = 1$. Our link function will need to map this constrained vector of θ values to a $\Re^{C-1}$. We can do this with a fairly straightforward variant of the logit function, too. Notice that there are $C-1$ probabilities we need to model (the C'th comes from the constraint $\sum_i \theta_i = 1$). We choose one vector β for each probability, and write β_i for the vector used to model θ_i. Then we can write

$$\mathbf{x}^T\beta_i = \log\left(\frac{\theta_i}{1 - \sum_u \theta_u}\right)$$

and this yields the model

$$\begin{aligned}
P(y = 0|\mathbf{x}, \beta) &= \frac{e^{\mathbf{x}^T\beta_0}}{1 + \sum_i e^{\mathbf{x}^T\beta_i}} \\
P(y = 1|\mathbf{x}, \beta) &= \frac{e^{\mathbf{x}^T\beta_1}}{1 + \sum_i e^{\mathbf{x}^T\beta_i}} \\
&\cdots \\
P(y = C-1|\mathbf{x}, \beta) &= \frac{1}{1 + \sum_i e^{\mathbf{x}^T\beta_i}}
\end{aligned}$$

and we would fit this model using maximum likelihood. The likelihood is easy to write out, and gradient descent is a good strategy for actually fitting models.

8.3.3 Regressing Count Data

Now imagine that the y_i values are counts. For example, y_i might have the count of the number of animals caught in a small square centered on $\mathbf{x}_i$ in a study region. As another example, $\mathbf{x}_i$ might be a set of features that represent a customer, and y_i might be the number of times that customer bought a particular product. The natural model for count data is a Poisson model, with parameter θ representing the intensity (reminder below).

Definition: 8.2 *Poisson distribution*

A non-negative, integer valued random variable X has a Poisson distribution when its probability distribution takes the form

$$P(\{X = k\}) = \frac{\theta^k e^{-\theta}}{k!},$$

where $\theta > 0$ is a parameter often known as the **intensity** of the distribution.

Now we need $\theta > 0$. A natural link function is to use

$$\mathbf{x}^T\beta = \log\theta$$

yielding a model

$$P(\{X = k\}) = \frac{e^{k\mathbf{x}^T\beta}e^{-e^{k\mathbf{x}^T\beta}}}{k!}.$$

Now assume we have a dataset. The negative log-likelihood can be written as

$$\begin{aligned} -\mathcal{L}(\beta) &= -\sum_i \log\left(\frac{e^{y_i\mathbf{x}_i^T\beta}e^{-e^{y_i\mathbf{x}_i^T\beta}}}{y_i!}\right) \\ &= -\sum_i \left(y_i\mathbf{x}_i^T\beta - e^{y_i\mathbf{x}_i^T\beta} - \log(y_i!)\right). \end{aligned}$$

There isn't a closed form minimum available, but the log-likelihood is convex, and gradient descent (or Newton's method) are enough to find a minimum. Notice that the $\log(y_i!)$ term isn't relevant to the minimization, and is usually dropped.

8.3.4 Deviance

Cross-validating a model is done by repeatedly splitting a data set into two pieces, training on one, evaluating some score on the other, and averaging the score. But we need to keep track of *what* to score. For earlier linear regression models (eg section 10.1), we have used the squared error of predictions. This doesn't really make sense for a generalized linear model, because predictions are of quite different form. It is usual to use the **deviance** of the model. Write y_t for the true prediction at a point, $\mathbf{x}_p$ for the independent variables we want to obtain a prediction for, $\hat{\beta}$ for our estimated parameters; a generalized linear model yields $P(y|\mathbf{x}_p, \hat{\beta})$. For our purposes, you should think of the deviance as

$$-2\log P(y_t|\mathbf{x}_p, \hat{\beta})$$

(this expression is sometimes adjusted in software to deal with extreme cases, etc.). Notice that this is quite like the least squares error for the linear regression case, because there

$$-2\log P(y|\mathbf{x}_p, \hat{\beta}) = (\mathbf{x}_p^T\hat{\beta} - y_t)^2/\sigma^2 + K$$

for K some constant.

8.4 L1 REGULARIZATION AND SPARSE MODELS

Forward and backward stagewise regression were strategies for adding independent variables to, or removing independent variables from, a model. An alternative, and very powerful, strategy is to construct a model with a method that forces some coefficients to be zero. The resulting model ignores the corresponding independent variables. Models built this way are often called **sparse models**, because (one hopes) that many independent variables will have zero coefficients, and so the model is using a sparse subset of the possible predictors.

In some situations, we are forced to use a sparse model. For example, imagine there are more independent variables than there are examples. In this case, the matrix $\mathcal{X}^T\mathcal{X}$ will be rank deficient. We could use a ridge regression (Section 7.4.2) and the rank deficiency problem will go away, but it would be hard to trust the resulting model, because it will likely use all the predictors (more detail below). We really want a model that uses a small subset of the predictors. Then, because the model ignores the other predictors, there will be more examples than there are predictors *that we use*.

There is now quite a strong belief amongst practitioners that using sparse models is the best way to deal with high dimensional problems (although there are lively debates about *which* sparse model to use, etc.). This is sometimes called the "bet on sparsity" principle: use a sparse model for high dimensional data, because dense models don't work well for such problems.

8.4.1 Dropping Variables with L1 Regularization

We have a large set of explanatory variables, and we would like to choose a small set that explains most of the variance in the independent variable. We could do this by encouraging β to have many zero entries. In section 7.4.2, we saw we could regularize a regression by adding a term to the cost function that discouraged large values of β. Instead of solving for the value of β that minimized $\sum_i (y_i - \mathbf{x}_i^T\beta)^2 = (\mathbf{y} - \mathcal{X}\beta)^T(\mathbf{y} - \mathcal{X}\beta)$ (which I shall call the **error cost**), we minimized

$$\sum_i (y_i - \mathbf{x}_i^T\beta)^2 + \frac{\lambda}{2}\beta^T\beta = (\mathbf{y} - \mathcal{X}\beta)^T(\mathbf{y} - \mathcal{X}\beta) + \frac{\lambda}{2}\beta^T\beta$$

(which I shall call the **L2 regularized error**). Here $\lambda > 0$ was a constant chosen by cross-validation. Larger values of λ encourage entries of β to be small, but do not force them to be zero. The reason is worth understanding.

Write β_k for the k'th component of β, and write β_{-k} for all the other components. Now we can write the L2 regularized error as a function of β_k:

$$(a + \lambda)\beta_k^2 - 2b(\beta_{-k})\beta_k + c(\beta_{-k})$$

where a is a function of the data and b and c are functions of the data and of β_{-k}. Now notice that the best value of β_k will be

$$\beta_k = \frac{b(\beta_{-k})}{(a + \lambda)}.$$

Notice that λ doesn't appear in the numerator. This means that, to force β_k to zero by increasing λ, we may have to make λ arbitrarily large. This is because the improvement in the penalty obtained by going from a small β_k to $\beta_k = 0$ is tiny – the penalty is proportional to β_k^2.

To force some components of β to zero, we need a penalty that grows linearly around zero rather than quadratically. This means we should use the $\mathbf{L}_1$ **norm** of β, given by

$$\| \beta \|_1 = \sum_k | \beta_k |.$$

To choose β, we must now solve

$$(\mathbf{y} - \mathcal{X}\beta)^T(\mathbf{y} - \mathcal{X}\beta) + \lambda \| \beta \|_1$$

for an appropriate choice of λ. An equivalent problem is to solve a constrained minimization problem, where one minimizes

$$(\mathbf{y} - \mathcal{X}\beta)^T(\mathbf{y} - \mathcal{X}\beta) \text{ subject to } \| \beta \|_1 \leq t$$

where t is some value chosen to get a good result, typically by cross-validation. There is a relationship between the choice of t and the choice of λ (with some thought, a smaller t will correspond to a bigger λ) but it isn't worth investigating in any detail.

Actually solving this system is quite involved, because the cost function is not differentiable. You should *not* attempt to use stochastic gradient descent, because this will not compel zeros to appear in $\hat{\beta}$ (exercises). There are several methods, which are beyond our scope. As the value of λ increases, the number of zeros in $\hat{\beta}$ will increase too. We can choose λ in the same way we used for classification; split the training set into a training piece and a validation piece, train for different values of λ, and test the resulting regressions on the validation piece. However, one consequence of modern methods is that we can generate a very good approximation to the path $\hat{\beta}(\lambda)$ for all values of $\lambda \geq 0$ about as easily as we can choose $\hat{\beta}$ for a particular value of λ.

One way to understand the models that result is to look at the behavior of cross-validated error as λ changes. The error is a random variable, random because of the random split. It is a fair model of the error that would occur on a randomly chosen test example (assuming that the training set is "like" the test set, in a way that I do not wish to make precise yet). We could use multiple splits, and average over the splits. Doing so yields both an average error for each value of λ and an estimate of the standard deviation of error. Figure 8.4 shows the result of doing so for two datasets. Again, there is no λ that yields the smallest validation error, because the value of error depends on the random split cross-validation. A reasonable choice of λ lies between the one that yields the smallest error encountered (one vertical line in the plot) and the largest value whose mean error is within one standard deviation of the minimum (the other vertical line in the plot). It is informative to keep track of the number of zeros in $\hat{\beta}$ as a function of λ, and this is shown in Figure 8.4.

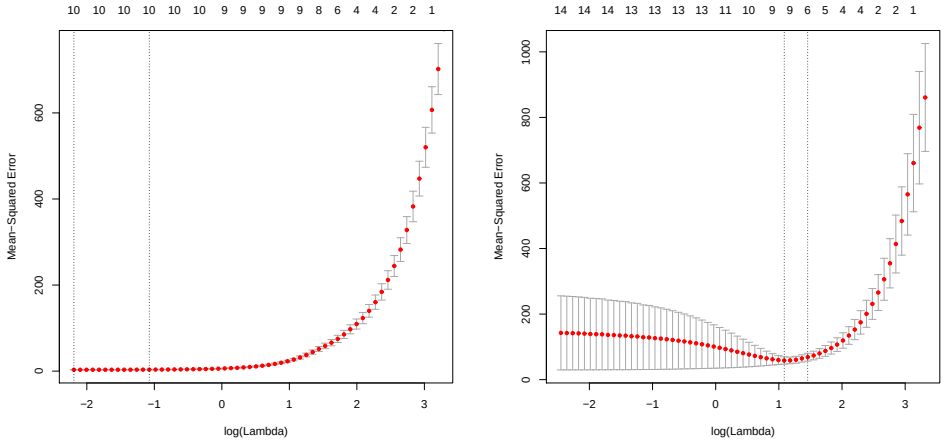


FIGURE 8.4: *Plots of mean-squared error as a function of log regularization parameter (i.e.* $\log \lambda$*) for a regression of weight against all variables for the bodyfat dataset using an L1 regularizer (i.e. a lasso). These plots show mean-squared error averaged over cross-validation folds with a vertical one standard deviation bar. On the* **left**, *the plot for the dataset with the six outliers identified in Figure 10.1 removed. On the* **right**, *the plot for the whole dataset. Notice how the outliers increase the variability of the error, and the best error. The top row of numbers gives the number of non-zero components in* $\hat{\beta}$*. Notice how as* λ *increases, this number falls (there are 15 explanatory variables, so the largest model would have 15 variables). The penalty ensures that explanatory variables with small coefficients are dropped as* λ *gets bigger.*

Worked example 8.3 *Building an L1 regularized regression*

Fit a linear regression to the bodyfat dataset, predicting weight as a function of all variables, and using the lasso to regularize. How good are the predictions? Do outliers affect the predictions?

Solution: I used the `glmnet` package, and I benefited a lot from example code by Trevor Hastie and Junyang Qian and published at https://web.stanford.edu/~hastie/glmnet/glmnet_alpha.html. You can see from Figure 8.7 that (a) for the case of outliers removed, the predictions are very good and (b) the outliers create problems. Note the magnitude of the error, and the low variance, for good cross validated choices. The main point of this example is to give you a start on producing R code, and I have put a code snippet in example 8.1.

Another way to understand the models is to look at how $\hat{\beta}$ changes as λ changes. We expect that, as λ gets smaller, more and more coefficients become non-zero. Figure 8.5 shows plots of coefficient values as a function of $\log \lambda$ for a

Listing 8.1: R code used for the lasso regression example of worked example 8.4

```
setwd('/users/daf/Current/courses/Probcourse/Regression/RCode/HeightWeight');
library(gdata)
bfd<-read.xls('BodyFat.xls')
library(glmnet)
xmat<-as.matrix(bfd[,-c(1, 5, 18)])
ymat<-as.matrix(bfd[, 5])
# keeping in the outliers
dmodel<-cv.glmnet(xmat, ymat, alpha=0)
plot(dmodel)
# without outliers
cbfd<-bfd[-c(216, 39, 41, 42, 221, 163),]
xmat<-as.matrix(cbfd[,-c(1, 5, 18)])
ymat<-as.matrix(cbfd[, 5])
model<-cv.glmnet(xmat, ymat, alpha=0)
plot(model)
```

regression of weight against all variables for the bodyfat dataset, penalised using the L_1 norm. For different values of λ, one gets different solutions for $\hat{\beta}$. When λ is very large, the penalty dominates, and so the norm of $\hat{\beta}$ must be small. In turn, most components of $\hat{\beta}$ are zero. As λ gets smaller, the norm of $\hat{\beta}$ falls and some components of become non-zero. At first glance, the variable whose coefficient grows very large seems important. Look more carefully; this is the last component introduced into the model. But Figure 8.4 implies that the right model has 7 components. This means that the right model has $\log \lambda \approx 1.3$, the vertical line shown in the detailed figure. In the best model, that coefficient is in fact zero.

The L_1 norm can sometimes produce an impressively small model from a large number of variables. In the UC Irvine Machine Learning repository, there is a dataset to do with the geographical origin of music (https://archive.ics.uci.edu/ml/datasets/Geographical+Original+of+Music). The dataset was prepared by Fang Zhou, and donors were Fang Zhou, Claire Q, and Ross D. King. Further details appear on that webpage, and in the paper: "Predicting the Geographical Origin of Music" by Fang Zhou, Claire Q and Ross. D. King, which appeared at ICDM in 2014. There are two versions of the dataset. One has 116 explanatory variables (which are various features representing music), and 2 independent variables (the latitude and longitude of the location where the music was collected). Figure 8.6 shows the results of a regression of latitude against the independent variables using L_1 regularization. Notice that the model that achieves the lowest cross-validated prediction error uses only 38 of the 116 variables.

Regularizing a regression with the L_1 norm is sometimes known as a **lasso**. A nuisance feature of the lasso is that, if several explanatory variables are correlated, it will tend to choose one for the model and omit the others (example in exercises). This can lead to models that have worse predictive error than models chosen using the L_2 penalty. One nice feature of good minimization algorithms for the lasso is

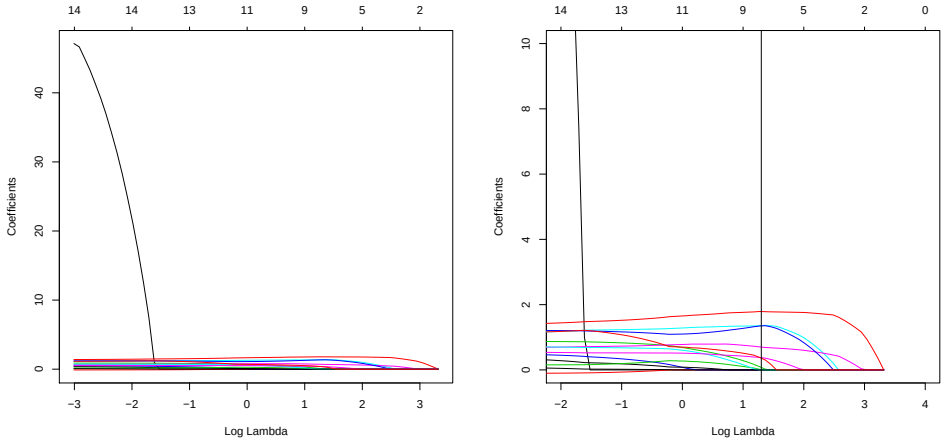


FIGURE 8.5: *Plots of coefficient values as a function of* $\log \lambda$ *for a regression of weight against all variables for the bodyfat dataset, penalised using the* L_1 *norm. In each case, the six outliers identified in Figure 10.1 were removed. On the* **left**, *the plot of the whole path for each coefficient (each curve is one coefficient). On the* **right**, *a detailed version of the plot. The vertical line shows the value of* $\log \lambda$ *the produces the model with smallest cross-validated error (look at Figure 8.4). Notice that the variable that appears to be important, because it would have a large weight with* $\lambda = 0$, *does not appear in this model.*

that it is easy to use both an L_1 penalty and an L_2 penalty together. One can form

$$\left(\frac{1}{N}\right)\left(\sum_i (y_i - \mathbf{x}_i^T \beta)^2\right) \quad + \quad \lambda\left(\frac{(1-\alpha)}{2}\|\beta\|_2{}^2 + \alpha\|\beta\|_1\right)$$
$$\text{Error} \quad + \quad \text{Regularizer}$$

where one usually chooses $0 \leq \alpha \leq 1$ by hand. Doing so can both discourage large values in β and encourage zeros. Penalizing a regression with a mixed norm like this is sometimes known as **elastic net**. It can be shown that regressions penalized with elastic net tend to produce models with many zero coefficients, while not omitting correlated explanatory variables. All the computation can be done by the `glmnet` package in R (see exercises for details).

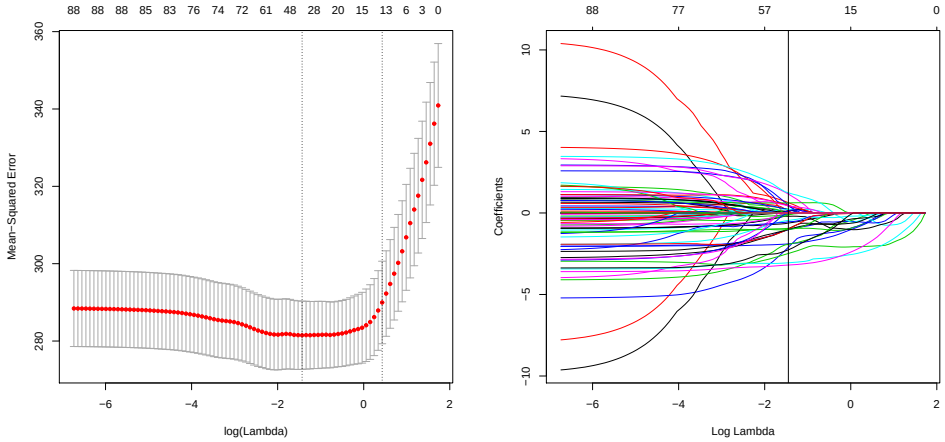


FIGURE 8.6: *Mean-squared error as a function of log regularization parameter (i.e.* $\log \lambda$*) for a regression of latitude against features describing music (details in text), using the dataset at* https://archive.ics.uci.edu/ml/datasets/Geographical+Original+of+Music *and penalized with the* L_1 *norm. The plot on the* **left** *shows mean-squared error averaged over cross-validation folds with a vertical one standard deviation bar. The top row of numbers gives the number of non-zero components in* $\hat{\beta}$*. Notice how as* λ *increases, this number falls. The penalty ensures that explanatory variables with small coefficients are dropped as* λ *gets bigger. On the* **right***, a plot of the coefficient values as a function of* $\log \lambda$ *for the same regression. The vertical line shows the value of* $\log \lambda$ *the produces the model with smallest cross-validated error. Only 38 of 116 explanatory variables are used by this model.*

Worked example 8.4 *Building an elastic net regression*

Fit a linear regression to the bodyfat dataset, predicting weight as a function of all variables, and using the elastic net to regularize. How good are the predictions? Do outliers affect the predictions?

Solution: I used the `glmnet` package, and I benefited a lot from example code by Trevor Hastie and Junyang Qian and published at https://web.stanford.edu/~hastie/glmnet/glmnet_alpha.html. The package will do ridge, lasso and elastic net regressions. One adjusts a parameter in the function call, α, that balances the terms; $\alpha = 0$ is ridge and $\alpha = 1$ is lasso. You can see from Figure 8.7 that (a) for the case of outliers removed, the predictions are very good and (b) the outliers create problems. Note the magnitude of the error, and the low variance, for good cross validated choices. The main point of this example is to give you a start on producing R code, and I have put a code snippet in example 8.2.

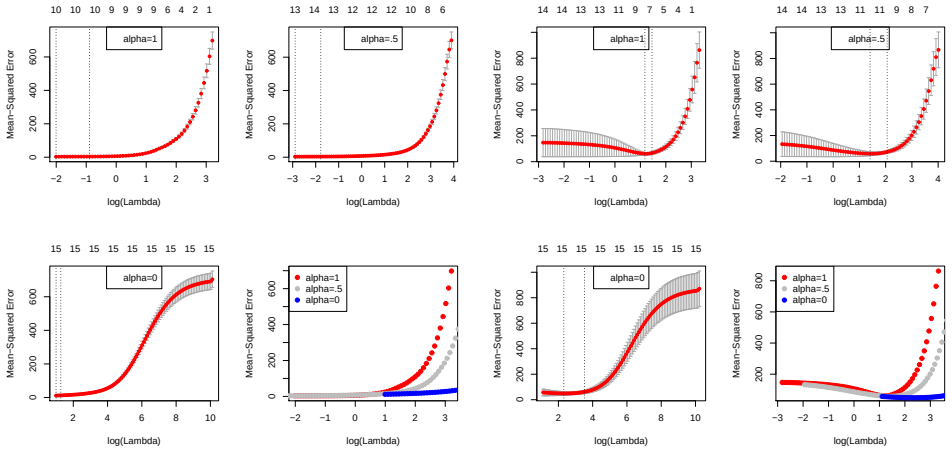


FIGURE 8.7: *Plots of mean-squared error as a function of log regularization parameter (i.e.* $\log \lambda$*) for a regression of weight against all variables for the bodyfat dataset using an elastic net regularizer for various choices of* α*. The case* $\alpha = 1$ *corresponds to a lasso;* $\alpha = 0$ *corresponds to a ridge; and* $\alpha = 0.5$ *is one possible choice yielding an elastic net. These plots show mean-squared error averaged over cross-validation folds with a vertical one standard deviation bar. On the* **left***, the plot for the dataset with the six outliers identified in Figure 10.1 removed. On the* **right***, the plot for the whole dataset. Notice how the outliers increase the variability of the error, and the best error. The top row of numbers gives the number of non-zero components in* $\hat{\beta}$*. Notice how as* λ *increases, this number falls (there are 15 explanatory variables, so the largest model would have 15 variables). The penalty ensures that explanatory variables with small coefficients are dropped as* λ *gets bigger.*

8.4.2 Wide Datasets

Now imagine we have more independent variables than examples (this is sometimes referred to as a "wide" dataset). This occurs quite often for a wide range of datasets; it's particularly common for biological datasets and natural language datasets. Unregularized linear regression must fail, because $\mathcal{X}^T\mathcal{X}$ must be rank deficient. Using an L2 (ridge) regularizer will produce an answer that should seem untrustworthy. The estimate of β is constrained by the data in some directions, but in other directions it is constrained only by the regularizer.

An estimate produced by L1 (lasso) regularization should look more reliable to you. Zeros in the estimate of β mean that the corresponding independent variables are ignored. Now if there are many zeros in the estimate of β, the model is being fit with a small subset of the independent variables. If this subset is small enough, then the number of independent variables that are actually being used is smaller than the number of examples. If the model gives low enough error, it should seem trustworthy in this case. There are some hard questions to face here (eg does the model choose the "right" set of variables?) that we can't deal with.

Listing 8.2: R code used for the elastic net regression example of worked example 8.4

```
setwd('/users/daf/Current/courses/Probcourse/Regression/RCode/HeightWeight');
library(gdata)
bfd<-read.xls('BodyFat.xls')
library(glmnet)
library(pls)
x<-as.matrix(bfd[,-c(1, 5, 18)])
y<-as.matrix(bfd[, 5])
foldid=sample(1:10,size=length(y),replace=TRUE)
cv1=cv.glmnet(x,y,foldid=foldid,alpha=1)
cv.5=cv.glmnet(x,y,foldid=foldid,alpha=.5)
cv0=cv.glmnet(x,y,foldid=foldid,alpha=0)
par(mfrow=c(2,2))
plot(cv1);
legend("top", legend="alpha=1")
plot(cv.5)
legend("top", legend="alpha=.5");
plot(cv0)
legend("top", legend="alpha=0")
plot(log(cv1$lambda),cv1$cvm,pch=19,col="red",
     xlab="log(Lambda)",ylab=cv1$name)
points(log(cv.5$lambda),cv.5$cvm,pch=19,col="grey")
points(log(cv0$lambda),cv0$cvm,pch=19,col="blue")
legend("topleft",
       legend=c("alpha=1","alpha=.5","alpha=0"),
       pch=19,col=c("red","grey","blue"))
```

Worked example 8.5 *L1 regularized regression for a "wide" dataset*

The gasoline dataset has 60 examples of near infrared spectra for gasoline of different octane ratings. The dataset is due to John H. Kalivas, and was originally described in the article "Two Data Sets of Near Infrared Spectra", in the journal *Chemometrics and Intelligent Laboratory Systems*, vol. 37, pp. 255259, 1997. Each example has measurements at 401 wavelengths. I found this dataset in the R library pls. Fit a regression of octane against infrared spectrum using L1 regularized logistic regression.

Solution: I used the `glmnet` package, and I benefited a lot from example code by Trevor Hastie and Junyang Qian and published at https://web.stanford.edu/~hastie/glmnet/glmnet_alpha.html. The package will do ridge, lasso and elastic net regressions. One adjusts a parameter in the function call, α, that balances the terms; $\alpha = 0$ is ridge and $\alpha = 1$ is lasso. Not surprisingly, the ridge isn't great. I tried $\alpha = 0.1$, $\alpha = 0.5$ and $\alpha = 1$. Results in Figure 8.8 suggest fairly strongly that very good predictions should be available with the lasso using quite a small regularization constant; there's no reason to believe that the best ridge models are better than the best elastic net models, or vice versa. The models are very sparse (look at the number of variables with non-zero weights, plotted on the top).

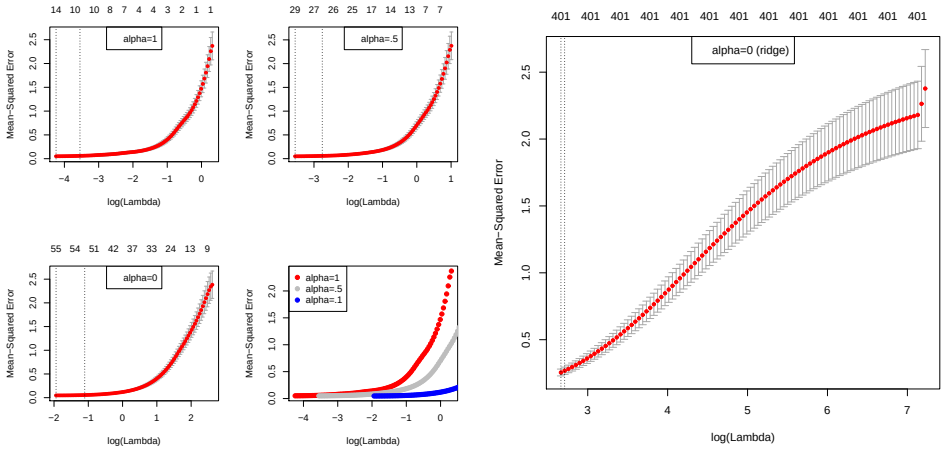


FIGURE 8.8: *On the* **left**, *a comparison between three values of* α *in a* `glmnet` *regression predicting octane from NIR spectra (see Example 8.5). The plots show cross-validated error against log regularization coefficient for* $\alpha = 1$ *(lasso) and two elastic-net cases,* $\alpha = 0.5$ *and* $\alpha = 0.1$. *I have plotted these curves separately, with error bars, and on top of each other but without error bars. The values at the top of each separate plot show the number of independent variables with non-zero coefficients in the best model with that regularization parameter. On the* **right**, *a ridge regression for comparison. Notice that the error is considerably larger, even at the best value of the regularization parameter.*

8.4.3 Using Sparsity Penalties with Other Models

A really nice feature of using an L1 penalty to enforce sparsity in a model is that it applies to a very wide range of models. For example, we can obtain a sparse SVM by replacing the L2 regularizer with an L1 regularizer. Most SVM packages will do this for you, although I'm not aware of any compelling evidence that this produces an improvement in most cases. All of the generalized linear models I described can be regularized with an L1 regularizer. For these cases, `glmnet` will do the computation required. The worked example shows using a multinomial (i.e. multiclass) logistic regression with an L1 regularizer.

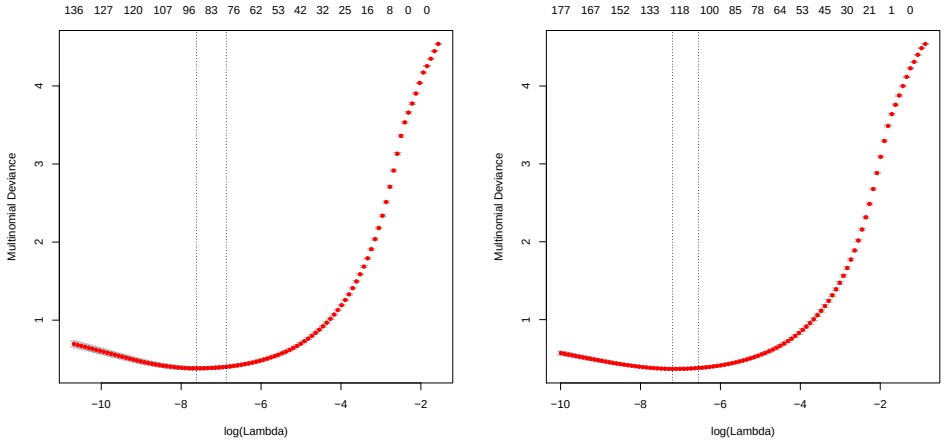


FIGURE 8.9: *Multiclass logistic regression on the MNIST data set, using a lasso and elastic net regularizers. On the* **left**, *deviance of held out data on the digit data set (worked example 8.6), for different values of the log regularization parameter in the lasso case. On the* **right**, *deviance of held out data on the digit data set (worked example 8.6), for different values of the log regularization parameter in the elastic net case,* $\alpha = 0.5$.

Worked example 8.6 *Multiclass logistic regression with an L1 regularizer*

The MNIST dataset consists of a collection of handwritten digits, which must be classified into 10 classes $(0, \ldots 9)$. There is a standard train/test split. This dataset is often called the zip code dataset because the digits come from zip codes, and has been quite widely studied. Yann LeCun keeps a record of the performance of different methods on this dataset at http://yann.lecun.com/exdb/mnist/. Obtain the Zip code dataset from http://statweb.stanford.edu/~tibs/ElemStatLearn/, and use a multiclass logistic regression with an L1 regularizer to classifiy it.

Solution: The dataset is rather large, and on my computer the fitting process takes a little time. Figure 8.9 shows what happens with the lasso, and with elasticnet with $\alpha = 0.5$ on the training set, using `glmnet` to predict and cross validation to select λ values. For the lasso, I found an error rate on the held out data of 8.5%, which is OK, but not great compared to other methods. For elastic net, I found a slightly better error rate (8.2%); I believe even lower error rates are possible with these codes.

Listing 8.3: R code used for the digit example of worked example 8.6

```
setwd('/users/daf/Current/courses/Probcourse/Regression/RCode/Digits');
library(glmnet)
digdat<-read.table('zip.train', sep=' ', header=FALSE)
y<-as.factor(digdat$V1)
x<-as.matrix(digdat[,-c(1, 258)])
mod<-cv.glmnet(x, y, family="multinomial", alpha=1)
plot(mod)
# ok now we need to predict
digtest<-read.table('zip.test', sep=' ', header=FALSE)
ytest<-as.factor(digtest$V1)
xtest<-as.matrix(digtest[,-c(1, 258)])
lmpredn<-predict(mod, xtest, type='class', s='lambda.min')
l1predn<-predict(mod, xtest, type='class', s='lambda.1se')
nmright<-sum(ytest==lmpredn)
errratem<-(1-nmright/dim(lmpredn))
n1right<-sum(ytest==lmpredn)
errrate1<-(1-n1right/dim(lmpredn))
mod.5<-cv.glmnet(x, y, family="multinomial", alpha=0.5)
plot(mod.5)
# ok now we need to predict
digtest<-read.table('zip.test', sep=' ', header=FALSE)
ytest<-as.factor(digtest$V1)
xtest<-as.matrix(digtest[,-c(1, 258)])
lmpredn.5<-predict(mod.5, xtest, type='class', s='lambda.min')
l1predn.5<-predict(mod.5, xtest, type='class', s='lambda.1se')
nmright.5<-sum(ytest==lmpredn.5)
errratem.5<-(1-nmright.5/dim(lmpredn.5))
n1right.5<-sum(ytest==lmpredn.5)
errrate1.5<-(1-n1right.5/dim(lmpredn.5))
```

8.5 YOU SHOULD

8.5.1 remember:

CHAPTER 9

Markov Chains and Hidden Markov Models

There are many situations where one must work with sequences. Here is a simple, and classical, example. We see a sequence of words, but the last word is missing. I will use the sequence "I had a glass of red wine with my grilled xxxx". What is the best guess for the missing word? You could obtain one possible answer by counting word frequencies, then replacing the missing word with the most common word. This is "the", which is not a particularly good guess because it doesn't fit with the previous word. Instead, you could find the most common pair of words matching "grilled xxxx", and then choose the second word. If you do this experiment (I used Google Ngram viewer, and searched for "grilled *"), you will find mostly quite sensible suggestions (I got "meats", "meat", "fish", "chicken", in that order). If you want to produce random sequences of words, the next word should depend on some of the words you have already produced.

9.1 MARKOV CHAINS

A sequence of random variables X_n is a **Markov chain** if it has the property that,

$$P(X_n = j|\text{values of all previous states}) = P(X_n = j|X_{n-1}),$$

or, equivalently, only the last state matters in determining the probability of the current state. The probabilities $P(X_n = j|X_{n-1} = i)$ are the **transition probabilities**. We will always deal with discrete random variables here, and we will assume that there is a finite number of states. For all our Markov chains, we will assume that

$$P(X_n = j|X_{n-1} = i) = P(X_{n-1} = j|X_{n-2} = i).$$

Formally, we focus on *discrete time, time homogenous Markov chains in a finite state space*. With enough technical machinery one can construct many other kinds of Markov chain.

One natural way to build Markov chains is to take a finite directed graph and label each directed edge from node i to node j with a probability. We interpret these probabilities as $P(X_n = j|X_{n-1} = i)$ (so the sum of probabilities over *outgoing* edges at any node must be 1). The Markov chain is then a **biased random walk** on this graph. A bug (or any other small object you prefer) sits on one of the graph's nodes. At each time step, the bug chooses one of the outgoing edges at random. The probability of choosing an edge is given by the probabilities on the drawing of the graph (equivalently, the transition probabilities). The bug then follows that edge. The bug keeps doing this until it hits an end state.

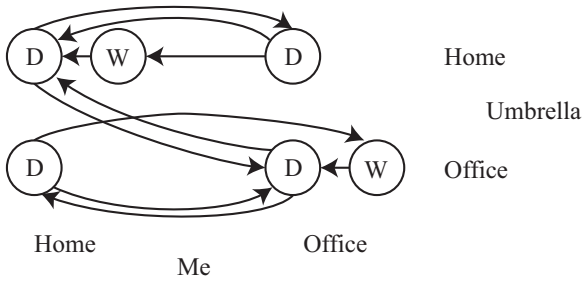


FIGURE 9.1: *A directed graph representing the umbrella example. Notice you can't arrive at the office wet with the umbrella at home (you'd have taken it), and so on. Labelling the edges with probabilities is left to the reader.*

Worked example 9.1 *Umbrellas*

I own one umbrella, and I walk from home to the office each morning, and back each evening. If it is raining (which occurs with probability p, and my umbrella is with me), I take it; if it is not raining, I leave the umbrella where it is. We exclude the possibility that it starts raining while I walk. Where I am, and whether I am wet or dry, forms a Markov chain. Draw a state machine for this Markov chain.

Solution: Figure 9.1 gives this chain. A more interesting question is with what probability I arrive at my destination wet? Again, we will solve this with simulation.

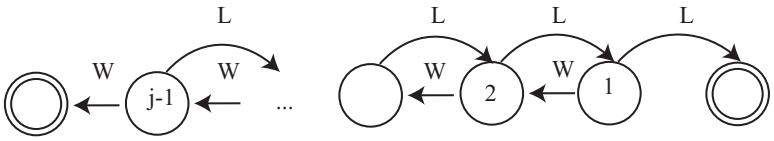


FIGURE 9.2: *A directed graph representing the gambler's ruin example. I have labelled each state with the amount of money the gambler has at that state. There are two end states, where the gambler has zero (is ruined), or has j and decides to leave the table. The problem we discuss is to compute the probability of being ruined, given the start state is s. This means that any state except the end states could be a start state. I have labelled the state transitions with "W" (for win) and "L" for lose, but have omitted the probabilities.*

Worked example 9.2 *The gambler's ruin*

Assume you bet 1 a tossed coin will come up heads. If you win, you get 1 and your original stake back. If you lose, you lose your stake. But this coin has the property that $P(H) = p < 1/2$. You have s when you start. You will keep betting until either (a) you have 0 (you are ruined; you can't borrow money) or (b) the amount of money you have accumulated is j, where $j > s$. The coin tosses are independent. The amount of money you have is a Markov chain. Draw the underlying state machine. Write $P(\text{ruined, starting with s}|p) = p_s$. It is straightforward that $p_0 = 1$, $p_j = 0$. Show that

$$p_s = pp_{s+1} + (1-p)p_{s-1}.$$

Solution: Figure 9.2 illustrates this example. The recurrence relation follows because the coin tosses are independent. If you win the first bet, you have $s+1$ and if you lose, you have $s-1$.

Notice an important difference between examples 9.1 and 9.2. For the gambler's ruin, the sequence of random variables can end (and your intuition likely tells you it should do so reliably). We say the Markov chain has an **absorbing state** – a state that it can never leave. In the example of the umbrella, there is an infinite sequence of random variables, each depending on the last. Each state of this chain is **recurrent** – it will be seen repeatedly in this infinite sequence. One way to have a state that is not recurrent is to have a state with outgoing but no incoming edges.

The gambler's ruin example illustrates some points that are quite characteristic of Markov chains. You can often write recurrence relations for the probability of various events. Sometimes you can solve them in closed form, though we will not pursue this thought further. It is often very helpful to think creatively about what the random variable is (example 9.3).

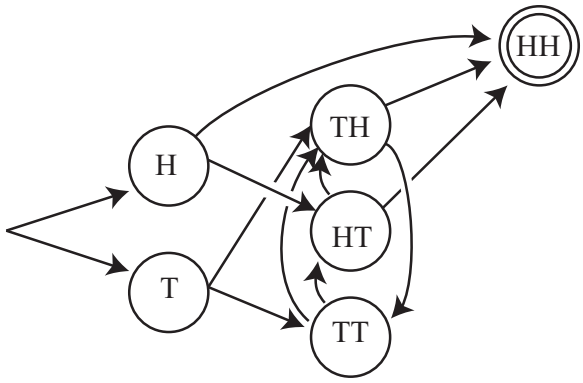


FIGURE 9.3: *A directed graph representing the coin flip example, using the pairs of random variables described in worked example 9.3. A sequence "HTHTHH" (where the last two H's are the last two flips) would be generated by transitioning to H, then to HT, then to TH, then to HT, then to TH, then to HH. By convention, the end state is a double circle. Each edge has probability 1/2.*

Worked example 9.3 *Multiple Coin Flips*

You choose to flip a fair coin until you see two heads in a row, and then stop. Represent the resulting sequence of coin flips with a Markov chain. What is the probability that you flip the coin four times?

Solution: You could think of the chain as being a sequence of independent coin flips. This is a Markov chain, but it isn't very interesting, and it doesn't get us anywhere. A better way to think about this problem is to have the X's be *pairs* of coin flips. The rule for changing state is that you flip a coin, then append the result to the state and drop the first item. Then you need a special state for stopping, and some machinery to get started. Figure 9.3 shows a drawing of the directed graph that represents the chain. The last three flips must have been THH (otherwise you'd go on too long, or end too early). But, because the second flip must be a T, the first could be either H or T. This means there are two sequences that work: $HTHH$ and $TTHH$. So $P(4 \text{ flips}) = 2/16 = 1/8$. We might want to answer significantly more interesting questions. For example, what is the probability that we must flip the coin more than 10 times? It is often possible to answer these questions by analysis, but we will use simulations.

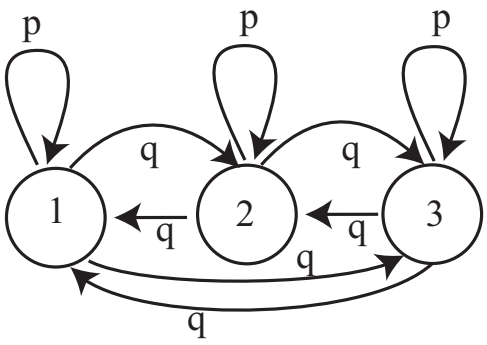


FIGURE 9.4: *A virus can exist in one of* 3 *strains. At the end of each year, the virus mutates. With probability* α*, it chooses uniformly and at random from one of the* 2 *other strains, and turns into that; with probability* $1 - \alpha$*, it stays in the strain it is in. For this figure, we have transition probabilities* $p = (1 - \alpha)$ *and* $q = (\alpha/2)$*.*

Useful Facts: 9.1 *Markov chains*

A Markov chain is a sequence of random variables X_n with the property that,

$$P(X_n = j|\text{values of all previous states}) = P(X_n = j|X_{n-1}).$$

9.1.1 Transition Probability Matrices

Define the matrix $\mathcal{P}$ with $p_{ij} = P(X_n = j|X_{n-1} = i)$. Notice that this matrix has the properties that $p_{ij} \geq 0$ and

$$\sum_j p_{ij} = 1$$

because at the end of each time step the model must be in some state. Equivalently, the sum of transition probabilities for outgoing arrows is one. Non-negative matrices with this property are **stochastic matrices**. By the way, you should look very carefully at the i's and j's here — Markov chains are usually written in terms of *row* vectors, and this choice makes sense in that context.

Worked example 9.4 *Viruses*

Write out the transition probability matrix for the virus of Figure 9.4, assuming that $\alpha = 0.2$.

Solution: We have $P(X_n = 1|X_{n-1} = 1) = (1 - \alpha) = 0.8$, and $P(X_n = 2|X_{n-1} = 1) = \alpha/2 = P(X_n = 3|X_{n-1} = 1)$; so we get

$$\begin{pmatrix} 0.8 & 0.1 & 0.1 \\ 0.1 & 0.8 & 0.1 \\ 0.1 & 0.1 & 0.8 \end{pmatrix}$$

Now imagine we do not know the initial state of the chain, but instead have a probability distribution. This gives $P(X_0 = i)$ for each state i. It is usual to take these k probabilities and place them in a k-dimensional *row vector*, which is usually written π. From this information, we can compute the probability distribution over the states at time 1 by

$$\begin{aligned} P(X_1 = j) &= \sum_i P(X_1 = j, X_0 = i) \\ &= \sum_i P(X_1 = j|X_0 = i)P(X_0 = i) \\ &= \sum_i p_{ij}\pi_i. \end{aligned}$$

If we write $\mathbf{p}^{(n)}$ for the row vector representing the probability distribution of the state at step n, we can write this expression as

$$\mathbf{p}^{(1)} = \pi\mathcal{P}.$$

Now notice that

$$\begin{aligned} P(X_2 = j) &= \sum_i P(X_2 = j, X_1 = i) \\ &= \sum_i P(X_2 = j|X_1 = i)P(X_1 = i) \\ &= \sum_i p_{ij} \left(\sum_{ki} p_{ki}\pi_k \right). \end{aligned}$$

so that

$$\mathbf{p}^{(n)} = \pi\mathcal{P}^n.$$

This expression is useful for simulation, and also allows us to deduce a variety of interesting properties of Markov chains.

Useful Facts: 9.2 *Transition probability matrices*

A finite state Markov chain can be represented with a matrix $\mathcal{P}$ of transition probabilities, where the i, j'th element $p_{ij} = P(X_n = j | X_{n-1} = i)$. This matrix is a stochastic matrix. If the probability distribution of state X_{n-1} is represented by π_{n-1}, then the probability distribution of state X_n is given by $\pi_{n-1}^T \mathcal{P}$.

9.1.2 Stationary Distributions

Worked example 9.5 *Viruses*

We know that the virus of Figure 9.4 started in strain 1. After two state transitions, what is the distribution of states when $\alpha = 0.2$? when $\alpha = 0.9$? What happens after 20 state transitions? If the virus starts in strain 2, what happens after 20 state transitions?

Solution: If the virus started in strain 1, then $\pi = [1, 0, 0]$. We must compute $\pi(\mathcal{P}(\alpha))^2$. This yields $[0.66, 0.17, 0.17]$ for the case $\alpha = 0.2$ and $[0.4150, 0.2925, 0.2925]$ for the case $\alpha = 0.9$. Notice that, because the virus with small α tends to stay in whatever state it is in, the distribution of states after two years is still quite peaked; when α is large, the distribution of states is quite uniform. After 20 transitions, we have $[0.3339, 0.3331, 0.3331]$ for the case $\alpha = 0.2$ and $[0.3333, 0.3333, 0.3333]$ for the case $\alpha = 0.9$; you will get similar numbers even if the virus starts in strain 2. After 20 transitions, the virus has largely "forgotten" what the initial state was.

In example 9.5, the distribution of virus strains after a long interval appears not to depend much on the initial strain. This property is true of many Markov chains. Assume that our chain has a finite number of states. Assume that any state can be reached from any other state, by some sequence of transitions. Such chains are called **irreducible**. Notice this means there is no absorbing state, and the chain cannot get "stuck" in a state or a collection of states. Then there is a unique vector $\mathbf{s}$, usually referred to as the **stationary distribution**, such that for *any* initial state distribution π,

$$\lim_{n \to \infty} \pi \mathcal{P}^{(n)} = \mathbf{s}.$$

Equivalently, if the chain has run through many steps, it no longer matters what the initial distribution is. The probability distribution over states will be $\mathbf{s}$.

The stationary distribution can often be found using the following property. Assume the distribution over states is $\mathbf{s}$, and the chain goes through one step. Then the new distribution over states must be $\mathbf{s}$ too. This means that

$$\mathbf{s}\mathcal{P} = \mathbf{s}$$

so that $\mathbf{s}$ is an eigenvector of $\mathcal{P}^T$, with eigenvalue 1. It turns out that, for an irreducible chain, there is exactly one such eigenvector.

The stationary distribution is a useful idea in applications. It allows us to answer quite natural questions, without conditioning on the initial state of the chain. For example, in the umbrella case, we might wish to know the probability I arrive home wet. This could depend on where the chain starts (example 9.6). If you look at the figure, the Markov chain is irreducible, so there is a stationary distribution and (as long as I've been going back and forth long enough for the chain to "forget" where it started), the probability it is in a particular state doesn't depend on where it started. So the most sensible interpretation of this probability is the probability of a particular state in the stationary distribution.

Worked example 9.6 *Umbrellas, but without a stationary distribution*

This is a different version of the umbrella problem, but with a crucial difference. When I move to town, I decide randomly to buy an umbrella with probability 0.5. I then go from office to home and back. If I have bought an umbrella, I behave as in example 9.1. If I have not, I just get wet. Illustrate this Markov chain with a state diagram.

Solution: Figure 9.5 does this. Notice this chain *isn't* irreducible. The state of the chain in the far future depends on where it started (i.e. did I buy an umbrella or not).

Useful Facts: 9.3 *Many Markov chains have stationary distributions*

If a Markov chain has a finite set of states, and if it is possible to get from any state to any other state, then the chain will have a stationary distribution. A sample state of the chain taken after it has been running for a long time will be a sample from that stationary distribution. Once the chain has run for long enough, it will visit states with a frequency corresponding to that stationary distribution, though it may take many state transitions to move from state to state.

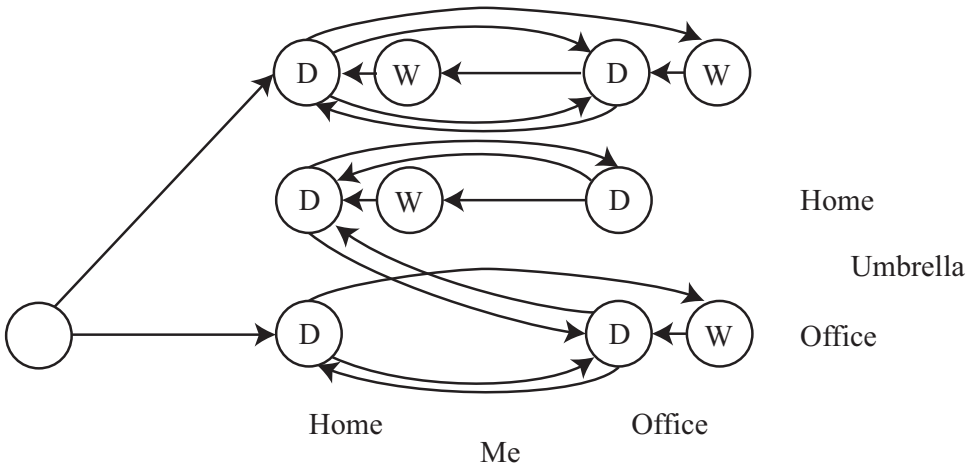


FIGURE 9.5: *In this umbrella example, there can't be a stationary distribution; what happens depends on the initial, random choice of buying/not buying an umbrella.*

9.1.3 Example: Markov Chain Models of Text

Imagine we wish to model English text. The very simplest model would be to estimate individual letter frequencies (most likely, by counting letters in a large body of example text). We might count spaces and punctuation marks as letters. We regard the frequencies as probabilities, then model a sequence by repeatedly drawing a letter from that probability model. You could even punctuate with this model by regarding punctuation signs as letters, too. We expect this model will produce sequences that are poor models of English text – there will be very long strings of 'a's, for example. This is clearly a (rather dull) Markov chain. It is sometimes referred to as a 0-th order chain or a 0-th order model, because each letter depends on the 0 letters behind it.

A slightly more sophisticated model would be to work with pairs of letters. Again, we would estimate the frequency of pairs by counting letter pairs in a body of text. We could then draw a first letter from the letter frequency table. Assume this is an 'a'. We would then draw the second letter by drawing a sample from the conditional probability of encountering each letter after 'a', which we could compute from the table of pair frequencies. Assume this is an 'n'. We get the third letter by drawing a sample from the conditional probability of encountering each letter after 'n', which we could compute from the table of pair frequencies, and so on. This is a first order chain (because each letter depends on the one letter behind it).

Second and higher order chains (or models) follow the general recipe, but the probability of a letter depends on more of the letters behind it. You may be concerned that conditioning a letter on the two (or k) previous letters means we don't have a Markov chain, because I said that the n'th state depends on only the $n-1$'th state. The cure for this concern is to use states that represent two (or k) letters, and adjust transition probabilities so that the states are consistent. So for

a second order chain, the string "abcde" is a sequence of four states, "ab", "bc", "cd", and "de".

Worked example 9.7 *Modelling short words*

Obtain a text resource, and use a trigram letter model to produce four letter words. What fraction of bigrams (resp. trigrams) do not occur in this resource? What fraction of the words you produce are actual words?

Solution: I used the text of a draft of this chapter. I ignored punctuation marks, and forced capital letters to lower case letters. I found 0.44 of the bigrams and 0.90 of the trigrams were not present. I built two models. In one, I just used counts to form the probability distributions (so there were many zero probabilities). In the other, I split a probability of 0.1 between all the cases that had not been observed. A list of 20 word samples from the first model is: "ngen", "ingu", "erms", "isso", "also", "plef", "trit", "issi", "stio", "esti", "coll", "tsma", "arko", "llso", "bles", "uati", "namp", "call", "riat", "eplu"; two of these are real English words (three if you count "coll", which I don't; too obscure), so perhaps 10% of the samples are real words. A list of 20 word samples from the second model is: "hate", "ther", "sout", "vect", "nces", "ffer", "msua", "ergu", "blef", "hest", "assu", "fhsp", "ults", "lend", "lsoc", "fysj", "uscr", "ithi", "prow", "lith"; four of these are real English words (you might need to look up "lith", but I refuse to count "hest" as being too archaic), so perhaps 20% of the samples are real words. In each case, the samples are too small to take the fraction estimates all that seriously.

Letter models can be good enough for (say) evaluating communication devices, but they're not great at producing words (example 9.7). More effective language models are obtained by working with words. The recipe is as above, but now we use words in place of letters. It turns out that this recipe applies to such domains as protein sequencing, dna sequencing and music synthesis as well, but now we use amino acids (resp. base pairs; notes) in place of letters. Generally, one decides what the basic item is (letter, word, amino acid, base pair, note, etc.). Then individual items are called **unigrams** and 0'th order models are **unigram models**; pairs are **bigrams** and first order models are **bigram models**; triples are **trigrams**, second order models **trigram models**; and for any other n, groups of n in sequence are **n-grams** and $n-1$'th order models are **n-gram models**.

Worked example 9.8 *Modelling text with n-grams of words*

Build a text model that uses bigrams (resp. trigrams, resp. n-grams) of words, and look at the paragraphs that your model produces.

Solution: This is actually a fairly arduous assignment, because it is hard to get good bigram frequencies without working with enormous text resources. Google publishes n-gram models for English words with the year in which the n-gram occurred and information about how many different books it occurred in. So, for example, the word "circumvallate" appeared 335 times in 1978, in 91 distinct books – some books clearly felt the need to use this term more than once. This information can be found starting at http://storage.googleapis.com/books/ngrams/books/datasetsv2.html. The raw dataset is huge, as you would expect. There are numerous n-gram language models on the web. Jeff Attwood has a brief discussion of some models at https://blog.codinghorror.com/markov-and-you/; Sophie Chou has some examples, and pointers to code snippets and text resources, at http://blog.sophiechou.com/2013/how-to-model-markov-chains/. Fletcher Heisler, Michael Herman, and Jeremy Johnson are authors of RealPython, a training course in Python, and give a nice worked example of a Markov chain language generator at https://realpython.com/blog/python/lyricize-a-flask-app-to-create-lyrics-using-markov-chains/. Markov chain language models are effective tools for satire. Garkov is Josh Millard's tool for generating comics featuring a well-known cat (at http://joshmillard.com/garkov/). There's a nice Markov chain for reviewing wines by Tony Fischetti at http://www.onthelambda.com/2014/02/20/how-to-fake-a-sophisticated-knowledge-of-wine-with-markov-chains/

It is usually straightforward to build a unigram model, because it is usually easy to get enough data to estimate the frequencies of the unigrams. There are many more bigrams than unigrams, many more trigrams than bigrams, and so on. This means that estimating frequencies can get tricky. In particular, you might need to collect an immense amount of data to see every possible n-gram several times. Without seeing every possible n-gram several times, you will need to deal with estimating the probability of encountering rare n-grams *that you haven't seen*. Assigning these n-grams a probability of zero is unwise, because that implies that they *never* occur, as opposed to occur seldom.

There are a variety of schemes for **smoothing** data (essentially, estimating the probability of rare items that have not been seen). The simplest one is to assign some very small fixed probability to every n-gram that has a zero count. It turns out that this is not a particularly good approach, because, for even quite small n, the fraction of n-grams that have zero count can be very large. In turn, you can find that most of the probability in your model is assigned to n-grams you have never seen. An improved version of this model assigns a fixed probability to unseen n-grams, then divides that probability up between all of the n-grams that

have never been seen before. This approach has its own characteristic problems. It ignores evidence that some of the unseen n-grams are more common than others. Some of the unseen n-grams have (n-1) leading terms that are (n-1)-grams that we *have* observed. These (n-1)-grams likely differ in frequency, suggesting that n-grams involving them should differ in frequency, too. More sophisticated schemes are beyond our scope, however.

9.2 ESTIMATING PROPERTIES OF MARKOV CHAINS

Many problems in probability can be worked out in closed form if one knows enough combinatorial mathematics, or can come up with the right trick. Textbooks are full of these, and we've seen some. Explicit formulas for probabilities are often extremely useful. But it isn't always easy or possible to find a formula for the probability of an event in a model. Markov chains are a particularly rich source of probability problems that might be too much trouble to solve in closed form. An alternative strategy is to build a simulation, run it many times, and count the fraction of outcomes where the event occurs. This is a simulation experiment.

9.2.1 Simulation

Imagine we have a random variable X with probability distribution $P(X)$ that takes values in some domain D. Assume that we can easily produce independent simulations, and that we wish to know $\mathbb{E}[f]$, the expected value of the function f under the distribution $P(X)$.

The weak law of large numbers tells us how to proceed. Define a new random variable $F = f(X)$. This has a probability distribution $P(F)$, which might be difficult to know. We want to estimate $\mathbb{E}[f]$, the expected value of the function f under the distribution $P(X)$. This is the same as $\mathbb{E}[F]$. Now if we have a set of IID samples of X, which we write x_i, then we can form a set of IID samples of F by forming $f(x_i) = f_i$. Write

$$F_N = \frac{\sum_{i=1}^{N} f_i}{N}.$$

This is a random variable, and the weak law of large numbers gives that, for any positive number ϵ

$$\lim_{N \to \infty} P(\{\|F_N - \mathbb{E}[F]\| > \epsilon\}) = 0.$$

You can interpret this as saying that, that for a set of IID random samples x_i, the probability that

$$\frac{\sum_{i=1}^{N} f(x_i)}{N}$$

is very close to $\mathbb{E}[f]$ is high for large N

Worked example 9.9 *Computing an Expectation*

Assume the random variable X is uniformly distributed in the range $[0-1]$, and the random variable Y is uniformly distributed in the range $[0-10]$. X and Z are independent. Write $Z = (Y - 5X)^3 - X^2$. What is $\mathsf{var}(\{Z\})$?

Solution: With enough work, one could probably work this out in closed form. An easy program will get a good estimate. We have that $\mathsf{var}(\{Z\}) = \mathbb{E}\left[Z^2\right] - \mathbb{E}[Z]^2$. My program computed 1000 values of Z (by drawing X and Y from the appropriate random number generator, then evaluating the function). I then computed $\mathbb{E}[Z]$ by averaging those values, and $\mathbb{E}[Z]^2$ by averaging their squares. For a run of my program, I got $\mathsf{var}(\{Z\}) = 2.76 \times 10^4$.

You can compute a probability using a simulation, too, because a probability can be computed by taking an expectation. Recall the property of indicator functions that

$$\mathbb{E}\left[\mathbb{I}_{[\mathcal{E}]}\right] = P(\mathcal{E})$$

(Section **??**). This means that computing the probability of an event $\mathcal{E}$ involves writing a function that is 1 when the event occurs, and 0 otherwise; we then estimate the expected value of that function.

Worked example 9.10 *Computing a Probability for Multiple Coin Flips*

You flip a fair coin three times. Use a simulation to estimate the probability that you see three H's.

Solution: You really should be able to work this out in closed form. But it's amusing to check with a simulation. I wrote a simple program that obtained a 1000x3 table of uniformly distributed random numbers in the range $[0-1]$. For each number, if it was greater than 0.5 I recorded an H and if it was smaller, I recorded a T. Then I counted the number of rows that had 3 H's (i.e. the expected value of the relevant indicator function). This yielded the estimate 0.127, which compares well to the right answer.

Worked example 9.11 *Computing a Probability*

Assume the random variable X is uniformly distributed in the range $[0-1]$, and the random variable Y is uniformly distributed in the range $[0-10]$. Write $Z = (Y - 5X)^3 - X^2$. What is $P(\{Z > 3\})$?

Solution: With enough work, one could probably work this out in closed form. An easy program will get a good estimate. My program computed 1000 values of Z (by drawing X and Y from the appropriate random number generator, then evaluating the function) and counted the fraction of Z values that was greater than 3 (which is the relevant indicator function). For a run of my program, I got $P(\{Z > 3\}) \approx 0.619$

For all the examples we will deal with, producing an IID sample of the relevant probability distribution will be straightforward. You should be aware that it can be very hard to produce an IID sample from an arbitrary distribution, particularly if that distribution is over a continuous variable in high dimensions.

9.2.2 Simulation Results as Random Variables

The estimate of a probability or of an expectation that comes out of a simulation experiment is a random variable, because it is a function of random numbers. If you run the simulation again, you'll get a different value, unless you did something silly with the random number generator. Generally, you should expect this random variable to have a normal distribution. You can check this by constructing a histogram over a large number of runs. The mean of this random variable is the parameter you are trying to estimate. It is useful to know that this random variable tends to be normal, because it means the standard deviation of the random variable tells you a lot about the likely values you will observe.

Another helpful rule of thumb, which is almost always right, is that the standard deviation of this random variable behaves like

$$\frac{C}{\sqrt{N}}$$

where C is a constant that depends on the problem and can be very hard to evaluate, and N is the number of runs of the simulation. What this means is that if you want to (say) double the accuracy of your estimate of the probability or the expectation, you have to run four times as many simulations. Very accurate estimates are tough to get, because they require immense numbers of simulation runs.

Figure 9.6 shows how the result of a simulation behaves when the number of runs changes. I used the simulation of example 9.11, and ran multiple experiments for each of a number of different samples (i.e. 100 experiments using 10 samples; 100 using 100 samples; and so on).

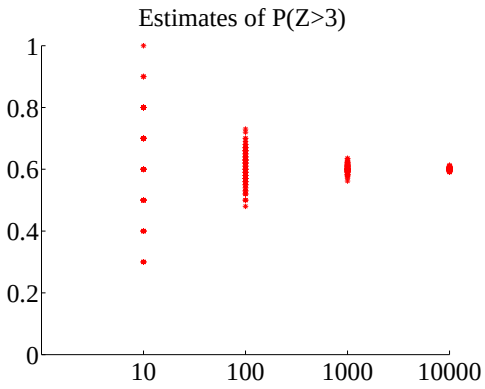


FIGURE 9.6: *Estimates of the probability from example 9.11, obtained from different runs of my simulator using different numbers of samples. In each case, I used 100 runs; the number of samples is shown on the horizontal axis. You should notice that the estimate varies pretty widely when there are only 10 samples in each run, but the variance (equivalently, the size of the spread) goes down sharply as the number of samples per run increases to 1000. Because we expect these estimates to be roughly normally distributed, the variance gives a good idea of how accurate the original probability estimate is.*

Worked example 9.12 *Getting 14's with 20-sided dice*

You throw 3 fair 20-sided dice. Estimate the probability that the sum of the faces is 14 using a simulation. Use $N = [1e1, 1e2, 1e3, 1e4, 1e5, 1e6]$. Which estimate is likely to be more accurate, and why?

Solution: You need a fairly fast computer, or this will take a long time. I ran ten versions of each experiment for $N = [1e1, 1e2, 1e3, 1e4, 1e5, 1e6]$, yielding ten probability estimates for each N. These were different for each version of the experiment, because the simulations are random. I got means of $[0, 0.0030, 0.0096, 0.0100, 0.0096, 0.0098]$, and standard deviations of $[00.00670.00330.00090.00020.0001]$. This suggests the true value is around 0.0098, and the estimate from $N = 1e6$ is best. The reason that the estimate with $N = 1e1$ is 0 is that the probability is very small, so you don't usually observe this case at all in only ten trials.

Small probabilities can be rather hard to estimate, as we would expect. In the case of example 9.11, let us estimate $P(\{Z > 950\})$. A few moments with a computer will show that this probability is of the order of 1e-3 to 1e-4. I obtained a million different simulated values of Z from my program, and saw 310 where $Z > 950$. This means that to know this probability to, say, three digits of numerical accuracy might involve a daunting number of samples. Notice that this does not contradict the rule of thumb that the standard deviation of the random variable

defined by a simulation estimate behaves like $\frac{C}{\sqrt{N}}$; it's just that in this case, C is very large indeed.

Useful Facts: 9.4 *The properties of simulations*

You should remember that

- The weak law of large numbers means you can estimate expectations and probabilities with a simulation.
- The result of a simulation is usually a normal random variable.
- The expected value of this random variable is usually the true value of the expectation or probability you are trying to simulate.
- The standard deviation of this random variable is usually $\frac{C}{\sqrt{N}}$, where N is the number of examples in the simulation and C is a number usually too hard to estimate.

Worked example 9.13 *Comparing simulation with computation*

You throw 3 fair six-sided dice. You wish to know the probability the sum is 3. Compare the true value of this probability with estimates from six runs of a simulation using $N = 10000$. What conclusions do you draw?

Solution: I ran six simulations with $N = 10000$, and got $[0.0038, 0.0038, 0.0053, 0.0041, 0.0056, 0.0049]$. The mean is 0.00458, and the standard deviation is 0.0007, which suggests the estimate isn't that great, but the right answer should be in the range $[0.00388, 0.00528]$ with probability about 0.68. The true value is $1/216 \approx 0.00463$. The estimate is tolerable, but not super accurate.

9.2.3 Simulating Markov Chains

We will always assume that we know the states and transition probabilities of the Markov chain. Properties that might be of interest in this case include: the probability of hitting an absorbing state; the expected time to go from one state to another; the expected time to hit an absorbing state; and which states have high probability under the stationary distribution.

Worked example 9.14 *Coin Flips with End Conditions*

I flip a coin repeatedly until I encounter a sequence HTHT, at which point I stop. What is the probability that I flip the coin nine times?

Solution: You might well be able to construct a closed form solution to this if you follow the details of example 10.1 and do quite a lot of extra work. A simulation is really straightforward to write; notice you can save time by not continuing to simulate coin flips once you've flipped past nine times. I got 0.0411 as the mean probability over 10 runs of a simulation of 1000 experiments each, with a standard deviation of 0.0056.

Worked example 9.15 *A Queue*

A bus is supposed to arrive at a bus stop every hour for 10 hours each day. The number of people who arrive to queue at the bus stop each hour has a Poisson distribution, with intensity 4. If the bus stops, everyone gets on the bus and the number of people in the queue becomes zero. However, with probability 0.1 the bus driver decides not to stop, in which case people decide to wait. If the queue is ever longer than 15, the waiting passengers will riot (and then immediately get dragged off by the police, so the queue length goes down to zero). What is the expected time between riots?

Solution: I'm not sure whether one could come up with a closed form solution to this problem. A simulation is completely straightforward to write. I get a mean time of 441 hours between riots, with a standard deviation of 391. It's interesting to play around with the parameters of this problem; a less conscientious bus driver, or a higher intensity arrival distribution, lead to much more regular riots.

Worked example 9.16 *Inventory*

A store needs to control its stock of an item. It can order stocks on Friday evenings, which will be delivered on Monday mornings. The store is old-fashioned, and open only on weekdays. On each weekday, a random number of customers comes in to buy the item. This number has a Poisson distribution, with intensity 4. If the item is present, the customer buys it, and the store makes 100; otherwise, the customer leaves. Each evening at closing, the store loses 10 for each unsold item on its shelves. The store's supplier insists that it order a fixed number k of items (i.e. the store must order k items each week). The store opens on a Monday with 20 items on the shelf. What k should the store use to maximise profits?

Solution: I'm not sure whether one could come up with a closed form solution to this problem, either. A simulation is completely straightforward to write. To choose k, you run the simulation with different k values to see what happens. I computed accumulated profits over 100 weeks for different k values, then ran the simulation five times to see which k was predicted. Results were $21, 19, 23, 20, 21$. I'd choose 21 based on this information.

For example 9.16, you should plot accumulated profits. If k is small, the store doesn't lose money by storing items, but it doesn't sell as much stuff as it could; if k is large, then it can fill any order but it loses money by having stock on the shelves. A little thought will convince you that k should be near 20, because that is the expected number of customers each week, so $k = 20$ means the store can expect to sell all its new stock. It may not be exactly 20, because it must depend a little on the balance between the profit in selling an item and the cost of storing it. For example, if the cost of storing items is very small compared to the profit, an very large k might be a good choice. If the cost of storage is sufficiently high, it might be better to never have anything on the shelves; this point explains the absence of small stores selling PC's.

Quite substantial examples are possible. The game "snakes and ladders" involves random walk on Markov chain. If you don't know this game, look it up; it's sometimes called "chutes and ladders", and there is an excellent Wikipedia page. The state is given by where each players' token is on the board, so on a 10x10 board one player involves 100 states, two players 100^2 states, and so on. The set of states is finite, though big. Transitions are random, because each player throws dice. The snakes (resp. ladders) represent extra edges in the directed graph. Absorbing states occur when a player hits the top square. It is straightforward to compute the expected number of turns for a given number of players by simulation, for example. For one commercial version, the Wikipedia page gives the crucial numbers: for two players, expected number of moves to a win is 47.76, and the first player wins with probability 0.509. Notice you might need to think a bit about how to write the program if there were, say, 8 players on a 12x12 board – you would likely avoid storing the entire state space.

9.3 EXAMPLE: RANKING THE WEB BY SIMULATING A MARKOV CHAIN

Perhaps the most valuable technical question of the last thirty years has been: Which web pages are interesting? Some idea of the importance of this question is that it was only really asked about 20 years ago, and at least one gigantic technology company has been spawned by a partial answer. This answer, due to Larry Page and Sergey Brin, and widely known as PageRank, revolves around simulating the stationary distribution of a Markov chain.

You can think of the world wide web as a directed graph. Each page is a state. Directed edges from page to page represent links. Count only the first link from a page to another page. Some pages are linked, others are not. We want to know how important each page is.

One way to think about importance is to think about what a random web surfer would do. The surfer can either (a) choose one of the outgoing links on a page at random, and follow it or (b) type in the URL of a new page, and go to that instead. This is a random walk on a directed graph. We expect that this random surfer should see a lot of pages that have lots of incoming links from other pages that have lots of incoming links that (and so on). These pages are important, because lots of pages have linked to them.

For the moment, ignore the surfer's option to type in a URL. Write $r(i)$ for the importance of the i'th page. We model importance as leaking from page to page across outgoing links (the same way the surfer jumps). Page i receives importance down each incoming link. The amount of importance is proportional to the amount of importance at the other end of the link, and inversely proportional to the number of links leaving that page. So a page with only one outgoing link transfers all its importance down that link; and the way for a page to receive a lot of importance is for it to have a lot of important pages link to it alone. We write

$$r(j) = \sum_{i \to j} \frac{r(i)}{|\, i \,|}$$

where $|\, i \,|$ means the total number of links pointing *out* of page i. We can stack the $r(j)$ values into a *row* vector $\mathbf{r}$, and construct a matrix $\mathcal{P}$, where

$$p_{ij} = \begin{cases} \frac{1}{|i|} & \text{if } i \text{ points to } j \\ 0 & \text{otherwise} \end{cases}$$

With this notation, the importance vector has the property

$$\mathbf{r} = \mathbf{r}\mathcal{P}$$

and should look a bit like the stationary distribution of a random walk to you, except that $\mathcal{P}$ isn't stochastic — there may be some rows where the row sum of $\mathcal{P}$ is zero, because there are *no* outgoing links from that page. We can fix this easily by replacing each row that sums to zero with $(1/n)\mathbf{1}$, where n is the total number of pages. Call the resulting matrix $\mathcal{G}$ (it's quite often called the **raw Google matrix**).

The web has pages with no outgoing links (which we've dealt with), pages with no incoming links, and even pages with no links at all. A random walk could

get trapped by moving to a page with no outgoing links. Allowing the surfer to randomly enter a URL sorts out all of these problems, because it inserts an edge of small weight from every node to every other node. Now the random walk cannot get trapped.

There are a variety of possible choices for the weight of these inserted edges. The original choice was to make each inserted edge have the same weight. Write $\mathbf{1}$ for the n dimensional column vector containing a 1 in each component, and let $0 < \alpha < 1$. We can write the matrix of transition probabilities as

$$\mathcal{G}(\alpha) = \alpha\frac{(\mathbf{1}\mathbf{1}^T)}{n} + (1-\alpha)\mathcal{G}$$

where $\mathcal{G}$ is the original Google matrix. An alternative choice is to choose a weight for each web page. This weight could come from a query; from advertising revenues; or from page visit statistics. Google keeps quiet about the details. Write this weight vector $\mathbf{v}$, and require that $\mathbf{1}^T\mathbf{v} = 1$ (i.e. the coefficients sum to one). Then we could have

$$\mathcal{G}(\alpha, \mathbf{v}) = \alpha\frac{(\mathbf{1}\mathbf{v}^T)}{n} + (1-\alpha)\mathcal{G}.$$

Now the importance vector $\mathbf{r}$ is the (unique, though I won't prove this) *row* vector $\mathbf{r}$ such that

$$\mathbf{r} = \mathbf{r}\mathcal{G}(\alpha, \mathbf{v}).$$

How do we compute this vector? One natural algorithm is to estimate $\mathbf{r}$ with a random walk, because $\mathbf{r}$ is the stationary distribution of a Markov chain. If we simulate this walk for many steps, the probability that the simulation is in state j should be $r(j)$, at least approximately.

This simulation is easy to build. Imagine our random walking bug sits on a web page. At each time step, it transitions to a new page by either (a) picking from all existing pages at random, using $\mathbf{v}$ as a probability distribution on the pages (which it does with probability α); or (b) chooses one of the outgoing links uniformly and at random, and follows it (which it does with probability $1-\alpha$). The stationary distribution of this random walk is $\mathbf{r}$. Another fact that I shall not prove is that, when α is sufficiently large, this random walk very quickly "forgets" it's initial distribution. As a result, you can estimate the importance of web pages by starting this random walk in a random location; letting it run for a bit; then stopping it, and collecting the page you stopped on. The pages you see like this are independent, identically distributed samples from $\mathbf{r}$; so the ones you see more often are more important, and the ones you see less often are less important.

9.4 HIDDEN MARKOV MODELS AND DYNAMIC PROGRAMMING

Imagine we wish to build a program that can transcribe speech sounds into text. Each small chunk of text can lead to one, or some, sounds, and some randomness is involved. For example, some people pronounce the word "fishing" rather like "fission". As another example, the word "scone" is sometimes pronounced rhyming with "stone", and sometimes rhyming with "gone". A Markov chain supplies a model of all possible text sequences, and allows us to compute the probability of any particular sequence. We will use a Markov chain to model text sequences, but

what we observe is sound. We must have a model of how sound is produced by text. With that model and the Markov chain, we want to produce text that (a) is a likely sequence of words and (b) is likely to have produced the sounds we hear.

Many applications contain the main elements of this example. We might wish to transcribe music from sound. We might wish to understand American sign language from video. We might wish to produce a written description of how someone moves from video observations. We might wish to break a substitution cipher. In each case, what we want to recover is a sequence that can be modelled with a Markov chain, but we don't see the states of the chain. Instead, we see noisy measurements that *depend* on the state of the chain, and we want to recover a state sequence that is (a) likely under the Markov chain model and (b) likely to have produced the measurements we observe.

9.4.1 Hidden Markov Models

Assume we have a finite state, time homogenous Markov chain, with S states. This chain will start at time 1, and the probability distribution $P(X_1 = i)$ is given by the vector π. At time u, it will take the state X_u, and its transition probability matrix is $p_{ij} = P(X_{u+1} = j | X_u = i)$. We do not observe the state of the chain. Instead, we observe some Y_u. We will assume that Y_u is also discrete, and there are a total of O possible states for Y_u for any u. We can write a probability distribution for these observations $P(Y_u | X_u = i) = q_i(Y_u)$. This distribution is the **emission distribution** of the model. For simplicity, we will assume that the emission distribution does not change with time.

We can arrange the emission distribution into a matrix $\mathcal{Q}$. A **hidden Markov model** consists of the transition probability distribution for the states, the relationship between the state and the probability distribution on Y_u, and the initial distribution on states, that is, $(\mathcal{P}, \mathcal{Q}, \pi)$. These models are often dictated by an application. An alternative is to build a model that best fits a collection of observed data, but doing so requires technical machinery we cannot expound here.

I will sketch how one might build a model for transcribing speech, but you should keep in mind this is just a sketch of a very rich area. We can obtain the probability of a word following some set of words using n-gram resources, as in section 9.1.3. We then build a model of each word in terms of small chunks of word that are likely to correspond to common small chunks of sound. We will call these chunks of sound **phonemes**. We can look up the different sets of phonemes that correspond to a word using a pronunciation dictionary. We can combine these two resources into a model of how likely it is one will pass from one phoneme inside a word to another, which might either be inside this word or inside another word. We now have $\mathcal{P}$. We will not spend much time on π, and might even model it as a uniform distribution. We can use a variety of strategies to build $\mathcal{Q}$. One is to build discrete features of a sound signal, then count how many times a particular set of features is produced when a particular phoneme is played.

9.4.2 Picturing Inference with a Trellis

Assume that we have a sequence of N measurements Y_i that we believe to be the output of a known hidden Markov model. We wish to recover the "best"

corresponding sequence of X_i. Doing so is inference. We will choose to recover a sequence X_i that maximises

$$\log P(X_1, X_2, \ldots, X_N | Y_1, Y_2, \ldots, Y_N, \mathcal{P}, \mathcal{Q}, \pi)$$

which is

$$\log \left(\frac{P(X_1, X_2, \ldots, X_N, Y_1, Y_2, \ldots, Y_N | \mathcal{P}, \mathcal{Q}, \pi)}{P(Y_1, Y_2, \ldots, Y_N)} \right)$$

and this is

$$\log P(X_1, X_2, \ldots, X_N, Y_1, Y_2, \ldots, Y_N | \mathcal{P}, \mathcal{Q}, \pi) - \log P(Y_1, Y_2, \ldots, Y_N).$$

Notice that $P(Y_1, Y_2, \ldots, Y_N)$ doesn't depend on the sequence of X_u we choose, and so the second term can be ignored. What is important here is that we can decompose $\log P(X_1, X_2, \ldots, X_N, Y_1, Y_2, \ldots, Y_N | \mathcal{P}, \mathcal{Q}, \pi)$ in a very useful way, because the X_u form a Markov chain. We want to maximise

$$\log P(X_1, X_2, \ldots, X_N, Y_1, Y_2, \ldots, Y_N | \mathcal{P}, \mathcal{Q}, \pi)$$

but this is

$$\begin{aligned} &\log P(X_1) + \log P(Y_1|X_1) + \\ &\log P(X_2|X_1) + \log P(Y_2|X_2) + \\ &\ldots \\ &\log P(X_N|X_{n-1}) + \log P(Y_N|X_N). \end{aligned}$$

Notice that this cost function has an important structure. It is a sum of terms. There are terms that depend on a single X_i (unary terms) and terms that depend on two (binary terms). Any state X_i appears in at most two binary terms.

We can illustrate this cost function in a structure called a **trellis**. This is a weighted, directed graph consisting of N copies of the state space, which we arrange in columns. There is a column corresponding to each measurement. We add a directed arrow from any state in the u'th column to any state in the $u+1$'th column if the transition probability between the states isn't 0. This represents the fact that there is a possible transition between these states. We then label the trellis with weights. We weight the node representing the case that state $X_u = j$ in the column corresponding to Y_u with $\log P(Y_u|X_u = j)$. We weight the arc from the node representing $X_u = i$ to that representing $X_{u+1} = j$ with $\log P(X_{u+1} = j|X_u = i)$.

The trellis has two crucial properties. Each directed path through the trellis from the start column to the end column represents a legal sequence of states. Now for some directed path from the start column to the end column, sum all the weights for the nodes and edges along this path. This sum is the log of the joint probability of that sequence of states with the measurements. You can verify each of these statements easily by reference to a simple example (try Figure 9.7)

There is an efficient algorithm for finding the path through a trellis which maximises the sum of terms. The algorithm is usually called **dynamic programming** or the **Viterbi algorithm**. I will describe this algorithm both in narrative, and as a recursion. We want to find the best path from each node in the first

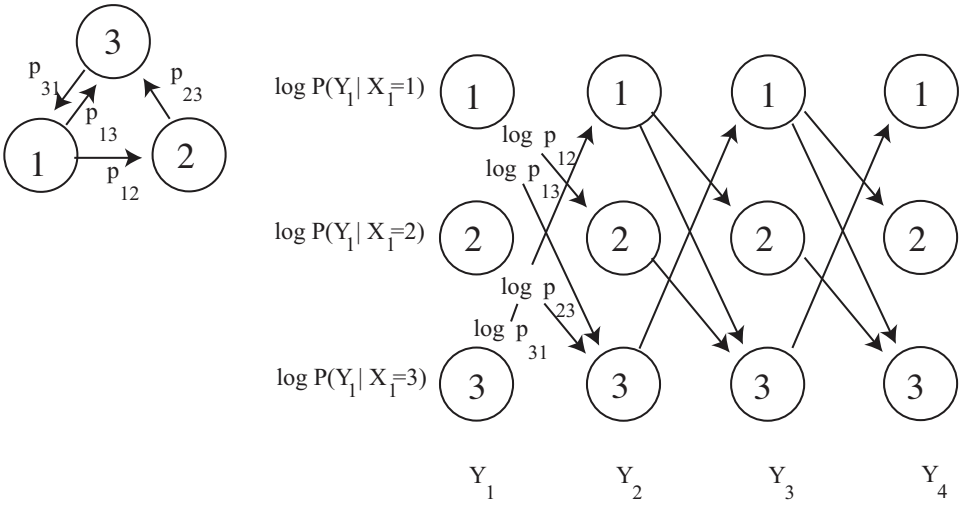


FIGURE 9.7: *At the* **top left**, *a simple state transition model. Each outgoing edge has some probability, though the topology of the model forces two of these probabilities to be* 1. *Below, the trellis corresponding to that model. Each path through the trellis corresponds to a legal sequence of states, for a sequence of three measurements. We weight the arcs with the log of the transition probabilities, and the nodes with the log of the emission probabilities. I have shown some weights*

column to each node in the last. There are S such paths, one for each node in the first column. Once we have these paths, we can choose the one with highest log joint probability. Now consider one of these paths. It passes through the i'th node in the u'th column. The path segment from this node to the end column must, itself, be the best path from this node to the end. If it wasn't, we could improve the original path by substituting the best. This is the key insight that gives us an algorithm.

Start at the *final* column of the tellis. We can evaluate the best path from each node in the final column to the final column, because that path is just the node, and the value of that path is the node weight. Now consider a two-state path, which will start at the second last column of the trellis (look at panel I in Figure 9.8). We can easily obtain the value of the best path leaving each node in this column. Consider a node: we know the weight of each arc leaving the node and the weight of the node at the far end of the arc, so we can choose the path segment with the largest value of the sum; this arc is the best we can do leaving that node. This sum is the best value obtainable on leaving that node—which is often known as the **cost to go function**.

Now, because we know the best value obtainable on leaving each node in the second-last column, we can figure out the best value obtainable on leaving each node in the third-last column (panel II in Figure 9.8). At each node in the third-last column, we have a choice of arcs. Each of these reaches a node *from which we know the value of the best path.* So we can choose the best path leaving a node in

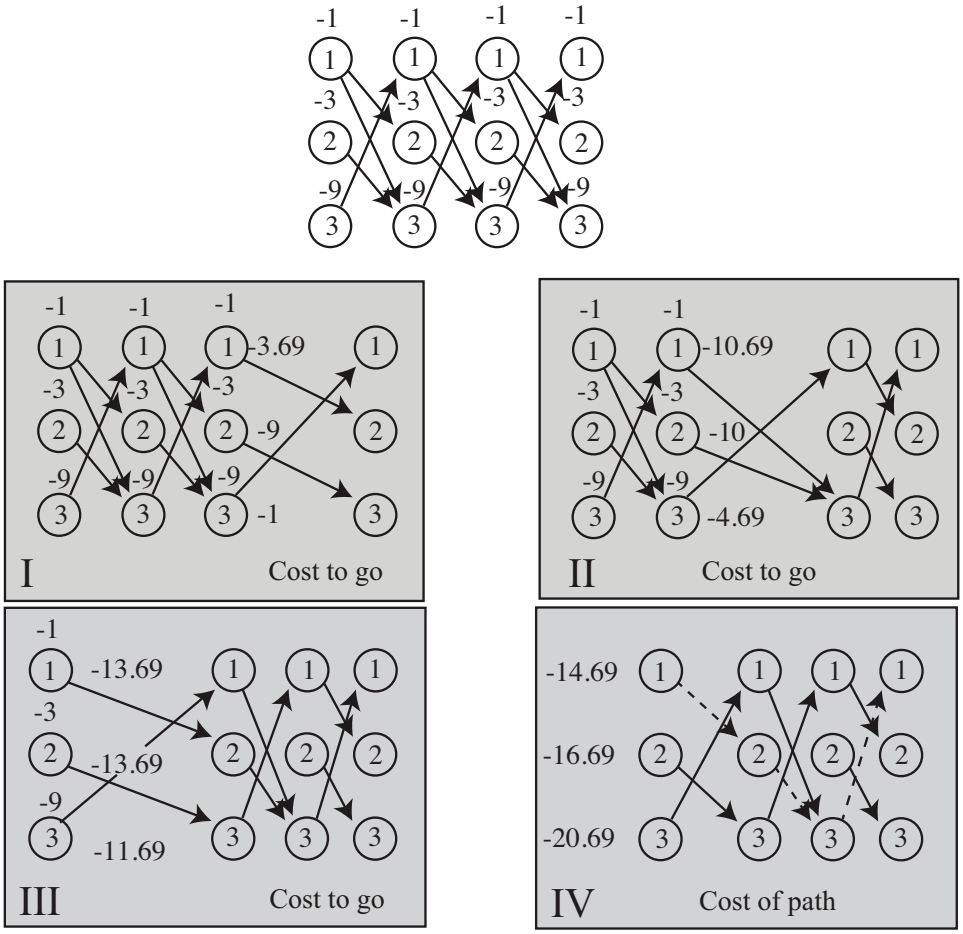


FIGURE 9.8: *An example of finding the best path through a trellis. The probabilities of leaving a node are uniform (and remember,* $\ln 2 \approx -0.69$*). Details in the text.*

the third-last column by finding the path that has the best value of: the arc weight leaving the node; the weight of the node in the second-last column the arc arrives at; and the value of the path leaving that node. This is much more easily done than described. All this works just as well for the fourth-last column, etc. (panel III in Figure 9.8) so we have a recursion. To find the value of the best path with $X_1 = i$, we go to the corresponding node in the first column, then add the value of the node to the value of the best path leaving that node (panel IV in Figure 9.8). Finally, to find the value of the best path leaving the first column, we compute the maximum value over all nodes in the first column.

We can also get the path with the maximum likelihood value. When we compute the value of a node, we erase all but the best arc leaving that node. Once we reach the first column, we simply follow the path from the node with the best value. This path is illustrated by dashed edges in Figure 9.8.

9.4.3 Dynamic Programming for HMM's: Formalities

We will formalize the recursion of the previous section with two ideas. First, we define $C_w(j)$ to be the cost of the best path segment to the end of the trellis *leaving* the node representing $X_w = j$. Second, we define $B_w(j)$ to be the node in column $w+1$ that lies on the best path *leaving* the node representing $X_w = j$. So $C_w(j)$ tells you the cost of the best path, and $B_w(j)$ tells you what node is next on the best path.

Now it is straightforward to find the cost of the best path leaving each node in the second last column, and also the path. In symbols, we have

$$C_{N-1}(j) = \max_u \left[\log P(X_N = u | X_{N-1} = j) + \log P(Y_N | X_N = u)\right]$$

and

$$B_{N-1}(j) = \operatorname*{argmax}_u \left[\log P(X_N = u | X_{N-1} = j) + \log P(Y_N | X_N = u)\right].$$

You should check this against step I of Figure 9.8

Once we have the best path leaving each node in the $w+1$'th column and its cost, it's straightforward to find the best path leaving the w'th column and its cost. In symbols, we have

$$C_w(j) = \max_u \left[\log P(X_{w+1} = u | X_w = j) + \log P(Y_{w+1} | X_{w+1} = u) + C_{w+1}(u)\right]$$

and

$$B_w(j) = \operatorname*{argmax}_u \left[\log P(X_{w+1} = u | X_w = j) + \log P(Y_{w+1} | X_{w+1} = u) + C_{w+1}(u)\right].$$

Check this against steps II and III in Figure 9.8.

9.4.4 Example: Simple Communication Errors

Hidden Markov models can be used to correct text errors. We will simplify somewhat, and assume we have text that has no punctuation marks, and no capital letters. This means there are a total of 27 symbols (26 lower case letters, and a space). We send this text down some communication channel. This could be a telephone line, a fax line, a file saving procedure or anything else. This channel makes errors independently at each character. For each location, with probability $1-p$ the output character at that location is the same as the input character. With probability p, the channel chooses randomly between the character one ahead or one behind in the character set, and produces that instead. You can think of this as a simple model for a mechanical error in one of those now ancient printers where a character strikes a ribbon to make a mark on the paper. We must reconstruct the transmission from the observations.

I built a unigram model, a bigram model, and a trigram model. I stripped the text of this chapter of punctuation marks and mapped the capital letters to lower case letters. I used an HMM package (in my case, for Matlab; but there's a good one for R as well) to perform inference. The main programming here is housekeeping

*	e	t	i	a	o	s	n	r	h
1.9e-1	9.7e-2	7.9e-2	6.6e-2	6.5e-2	5.8e-2	5.5e-2	5.2e-2	4.8e-2	3.7e-2

TABLE 9.1: *The most common single letters (unigrams) that I counted from a draft of this chapter, with their probabilities. The '*' stands for a space. Spaces are common in this text, because I have tended to use short words (from the probability of the '*', average word length is between five and six letters).*

Lead char					
*	*t (2.7e-2)	*a (1.7e-2)	*i (1.5e-2)	*s (1.4e-2)	*o (1.1e-2)
e	e* (3.8e-2)	er (9.2e-3)	es (8.6e-3)	en (7.7e-3)	el (4.9e-3)
t	th (2.2e-2)	t* (1.6e-2)	ti (9.6e-3)	te (9.3e-3)	to (5.3e-3)
i	in (1.4e-2)	is (9.1e-3)	it (8.7e-3)	io (5.6e-3)	im (3.4e-3)
a	at (1.2e-2)	an (9.0e-3)	ar (7.5e-3)	a* (6.4e-3)	al (5.8e-3)
o	on (9.4e-3)	or (6.7e-3)	of (6.3e-3)	o* (6.1e-3)	ou (4.9e-3)
s	s* (2.6e-2)	st (9.4e-3)	se (5.9e-3)	si (3.8e-3)	su (2.2e-3)
n	n* (1.9e-2)	nd (6.7e-3)	ng (5.0e-3)	ns (3.6e-3)	nt (3.6e-3)
r	re (1.1e-2)	r* (7.4e-3)	ra (5.6e-3)	ro (5.3e-3)	ri (4.3e-3)
h	he (1.4e-2)	ha (7.8e-3)	h* (5.3e-3)	hi (5.1e-3)	ho (2.1e-3)

TABLE 9.2: *The most common bigrams that I counted from a draft of this chapter, with their probabilities. The '*' stands for a space. For each of the 10 most common letters, I have shown the five most common bigrams with that letter in the lead. This gives a broad view of the bigrams, and emphasizes the relationship between unigram and bigram frequencies. Notice that the first letter of a word has a slightly different frequency than letters (top row; bigrams starting with a space are first letters). About* 40% *of the possible bigrams do not appear in the text.*

to make sure the transition and emission models are correct. About 40% of the bigrams and 86% of the trigrams did not appear in the text. I smoothed the bigram and trigram probabilities by dividing the probability 0.01 evenly between all unobserved bigrams (resp. trigrams). The most common unigrams, bigrams and trigrams appear in the tables. As an example sequence, I used

> "the trellis has two crucial properties each directed path through the trellis from the start column to the end column represents a legal sequence of states now for some directed path from the start column to the end column sum all the weights for the nodes and edges along this path this sum is the log of the joint probability of that sequence of states with the measurements you can verify each of these statements easily by reference to a simple example"

(which is text you could find in a draft of this chapter). There are 456 characters in this sequence.

When I ran this through the noise process with $p = 0.0333$, I got

> "theztrellis has two crucial properties each directed path through the tqdllit from the start column to the end coluln represents a legal se-

*th	the	he*	is*	*of	of*	on*	es*	*a*	ion
1.7e-2	1.2e-2	9.8e-3	6.2e-3	5.6e-3	5.4e-3	4.9e-3	4.9e-3	4.9e-3	4.9e-3
tio	e*t	in*	*st	*in	at*	ng*	ing	*to	*an
4.6e-3	4.5e-3	4.2e-3	4.1e-3	4.1e-3	4.0e-3	3.9e-3	3.9e-3	3.8e-3	3.7e-3

TABLE 9.3: *The most frequent 10 trigrams in a draft of this chapter, with their probabilities. Again, '*' stands for space. You can see how common 'the' and '*a*' are; 'he*' is common because '*the*' is common. About* 80% *of possible trigrams do not appear in the text.*

> quencezof states now for some directed path from the start column to thf end column sum aml the veights for the nodes and edges along this path this sum is the log of the joint probability oe that sequence of states wish the measurements youzcan verify each of these statements easily by reference to a simple examqle"

which is mangled but not too badly (13 of the characters are changed, so 443 locations are the same).

The unigram model produces

> "the trellis has two crucial properties each directed path through the tqdllit from the start column to the end coluln represents a legal sequence of states now for some directed path from the start column to thf end column sum aml the veights for the nodes and edges along this path this sum is the log of the joint probability oe that sequence of states wish the measurements you can verify each of these statements easily by reference to a simple examqle"

which fixes three errors. The unigram model only changes an observed character when the probability of encountering that character on its own is less than the probability it was produced by noise. This occurs only for "z", which is unlikely on its own and is more likely to have been a space. The bigram model produces

> "she trellis has two crucial properties each directed path through the trellit from the start column to the end coluln represents a legal sequence of states now for some directed path from the start column to the end column sum aml the veights for the nodes and edges along this path this sum is the log of the joint probability oe that sequence of states wish the measurements you can verify each of these statements easily by reference to a simple example"

This is the same as the correct text in 449 locations, so somewhat better than the noisy text. The trigram model produces

> "the trellis has two crucial properties each directed path through the trellit from the start column to the end column represents a legal sequence of states now for some directed path from the start column to the end column sum all the weights for the nodes and edges along this path this sum is the log of the joint probability of that sequence of states

with the measurements you can verify each of these statements easily by reference to a simple example"

which corrects all but one of the errors (look for "trellit").

9.5 YOU SHOULD

9.5.1 remember these definitions:

9.5.2 remember these terms:

9.5.3 remember these facts:

9.5.4 be able to:

- Estimate various probabilities and expectations for a Markov chain by simulation.
- Evaluate the results of multiple runs of a simple simulation.

- Set up a simple HMM and use it to solve problems.

PROBLEMS

9.1. Multiple die rolls: You roll a fair die until you see a 5, then a 6; after that, you stop. Write $P(N)$ for the probability that you roll the die N times.
- **(a)** What is $P(1)$?
- **(b)** Show that $P(2) = (1/36)$.
- **(c)** Draw a directed graph encoding all the sequences of die rolls that you could encounter. Don't write the events on the edges; instead, write their probabilities. There are 5 ways not to get a 5, but only one probability, so this simplifies the drawing.
- **(d)** Show that $P(3) = (1/36)$.
- **(e)** Now use your directed graph to argue that $P(N) = (5/6)P(N-1) + (25/36)P(N-2)$.

9.2. More complicated multiple coin flips: You flip a fair coin until you see either HTH or THT, and then you stop. We will compute a recurrence relation for $P(N)$.
- **(a)** Draw a directed graph for this chain.
- **(b)** Think of the directed graph as a finite state machine. Write Σ_N for some string of length N accepted by this finite state machine. Use this finite state machine to argue that $Sigma_N$ has one of four forms:

  1. $TT\Sigma_{N-2}$
  2. $HH\Sigma_{N-3}$
  3. $THH\Sigma_{N-2}$
  4. $HTT\Sigma_{N-3}$

- **(c)** Now use this argument to show that $P(N) = (1/2)P(N-2)+(1/4)P(N-3)$.

9.3. For the umbrella example of worked example 9.1, assume that with probability 0.7 it rains in the evening, and 0.2 it rains in the morning. I am conventional, and go to work in the morning, and leave in the evening.
- **(a)** Write out a transition probability matrix.
- **(b)** What is the stationary distribution? (you should use a simple computer program for this).
- **(c)** What fraction of evenings do I arrive at home wet?
- **(d)** What fraction of days do I arrive at my destination dry?

PROGRAMMING EXERCISES

9.4. A dishonest gambler has two dice and a coin. The coin and one die are both fair. The other die is unfair. It has $P(n) = [0.5, 0.1, 0.1, 0.1, 0.1, 0.1]$ (where n is the number displayed on the top of the die). The gambler starts by choosing a die. Choosing a die is by flipping a coin; if the coin comes up heads, the gambler chooses the fair die, otherwise, the unfair die. The gambler rolls the chosen die repeatedly until a 6 comes up. When a 6 appears, the gambler chooses again (by flipping a coin, etc), and continues.
- **(a)** Model this process with a hidden markov model. The emitted symbols should be $1, \ldots, 6$. Doing so requires only two hidden states (which die is

in hand). Simulate a long sequence of rolls using this model. What is the probability the emitted symbol is 1?

(b) Use your simulation to produce 10 sequences of 100 symbols. Record the hidden state sequence for each of these. Now recover the hidden state using dynamic programming (you should likely use a software package for this; there are many good ones for R and Matlab). What fraction of the hidden states is correctly identified by your inference procedure?

(c) Does inference accuracy improve when you use sequences of 1000 symbols?

9.5. Warning: this exercise is fairly elaborate, though straightforward. We will correct text errors using a hidden Markov model.

(a) Obtain the text of a copyright-free book in plain characters. One natural source is Project Gutenberg, at https://www.gutenberg.org. Simplify this text by dropping all punctuation marks except spaces, mapping capital letters to lower case, and mapping groups of many spaces to a single space. The result will have 27 symbols (26 lower case letters and a space). From this text, count unigram, bigram and trigram letter frequencies.

(b) Use your counts to build models of unigram, bigram and trigram letter probabilities. You should build both an unsmoothed model, and at least one smoothed model. For the smoothed models, choose some small amount of probability ϵ and split this between all events with zero count. Your models should differ only by the size of ϵ.

(c) Construct a corrupted version of the text by passing it through a process that, with probability p_c, replaces a character with a randomly chosen character, and otherwise reports the original character.

(d) For a reasonably sized block of corrupted text, use an HMM inference package to recover the best estimate of your true text. Be aware that your inference will run more slowly as the block gets bigger, but you won't see anything interesting if the block is (say) too small to contain any errors.

(e) For $p_c = 0.01$ and $p_c = 0.1$, estimate the error rate for the corrected text for different values of ϵ. Keep in mind that the corrected text could be worse than the corrupted text.

CHAPTER 10

Clustering using Probability Models

10.1 THE MULTIVARIATE NORMAL DISTRIBUTION

All the nasty facts about high dimensional data, above, suggest that we need to use quite simple probability models. By far the most important model is the multivariate normal distribution, which is quite often known as the multivariate gaussian distribution. There are two sets of parameters in this model, the mean μ and the covariance Σ. For a d-dimensional model, the mean is a d-dimensional column vector and the covariance is a $d \times d$ dimensional matrix. The covariance is a symmetric matrix. For our definitions to be meaningful, the covariance matrix must be positive definite.

The form of the distribution $p(\mathbf{x}|\mu, \Sigma)$ is

$$p(\mathbf{x}|\mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^d \det(\Sigma)}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right),$$

where Σ is a positive definite matrix. Notice that if Σ is not positive definite, then we cannot have a probability distribution, because there are some directions $\mathbf{d}$ such that $\exp\left(-\frac{1}{2}(t\mathbf{d}-\mu)^T\Sigma^{-1}(t\mathbf{d}-\mu)\right)$ does not fall off to zero as t limits to infinity. In turn, this means we can't compute the integral, and so can't normalize.

The following facts explain the names of the parameters:

Useful Facts: 10.1 *Parameters of a Multivariate Normal Distribution*

Assuming a multivariate normal distribution, we have

- $\mathbb{E}[\mathbf{x}] = \mu$, meaning that the mean of the distribution is μ.
- $\mathbb{E}\left[(\mathbf{x}-\mu)(\mathbf{x}-\mu)^T\right] = \Sigma$, meaning that the entries in Σ represent covariances.

Assume I know have a dataset of items $\mathbf{x}_i$, where i runs from 1 to N, and we wish to model this data with a multivariate normal distribution. The maximum likelihood estimate of the mean, $\hat{\mu}$, is

$$\hat{\mu} = \frac{\sum_i \mathbf{x}_i}{N}$$

(which is quite easy to show). The maximum likelihood estimate of the covariance, $\hat{\Sigma}$, is

$$\hat{\Sigma} = \frac{\sum_i (\mathbf{x}_i - \hat{\mu})(\mathbf{x}_i - \hat{\mu})^T}{N}$$

(which is rather a nuisance to show, because you need to know how to differentiate a determinant). You should be aware that this estimate is not guaranteed to be positive definite, even though the covariance matrix of a gaussian must be positive definite. We deal with this problem by checking the estimate. If its smallest eigenvalue is too close to zero, then we add some small positive constant times the identity to get a positive definite matrix.

10.1.1 Affine Transformations and Gaussians

Gaussians behave very well under affine transformations. In fact, we've already worked out all the math. Assume I have a dataset $\{\mathbf{x}\}$ with N data points $\mathbf{x}_i$, each a d-dimensional vector. The mean of the maximum likelihood gaussian model is $\mathsf{mean}\left(\{\mathbf{x}\}\right)$, and the covariance is $\mathsf{Covmat}\left(\{\mathbf{x}\}\right)$. We assume that this is positive definite, or adjust it as above.

I can now transform the data with an affine transformation, to get $\mathbf{y}_i = \mathcal{A}\mathbf{x}_i + \mathbf{b}$. We assume that $\mathcal{A}$ is a square matrix with full rank, so that this transformation is 1-1. The mean of the maximum likelihood gaussian model for the transformed dataset is $\mathsf{mean}\left(\{\mathbf{y}\}\right) = \mathcal{A}\mathsf{mean}\left(\{\mathbf{x}\}\right) + \mathbf{b}$. Similarly, the covariance is $\mathsf{Covmat}\left(\{\mathbf{y}\}\right) = \mathcal{A}\mathsf{Covmat}\left(\{\mathbf{x}\}\right)\mathcal{A}^T$.

A very important point follows in an obvious way. I can apply an affine transformation to any multivariate gaussian to obtain one with (a) zero mean and (b) independent components. In turn, this means that, *in the right coordinate system*, any gaussian is a product of zero mean, unit standard deviation, one-dimensional normal distributions. This fact is quite useful. For example, it means that simulating multivariate normal distributions is quite straightforward — you could simulate a standard normal distribution for each component, then apply an affine transformation.

10.1.2 Plotting a 2D Gaussian: Covariance Ellipses

There are some useful tricks for plotting a 2D Gaussian, which are worth knowing both because they're useful, and they help to understand Gaussians. Assume we are working in 2D; we have a Gaussian with mean μ (which is a 2D vector), and covariance Σ (which is a 2x2 matrix). We could plot the collection of points $\mathbf{x}$ that has some fixed value of $p(\mathbf{x}|\mu, \Sigma)$. This set of points is given by:

$$\frac{1}{2}\left((\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right) = c^2$$

where c is some constant. I will choose $c^2 = \frac{1}{2}$, because the choice doesn't matter, and this choice simplifies some algebra. You might recall that a set of points $\mathbf{x}$ that satisfies a quadratic like this is a conic section. Because Σ (and so Σ^{-1}) is positive definite, the curve is an ellipse. There is a useful relationship between the geometry of this ellipse and the Gaussian.

This ellipse — like all ellipses — has a major axis and a minor axis. These are at right angles, and meet at the center of the ellipse. We can determine the properties of the ellipse in terms of the Gaussian quite easily. The geometry of the ellipse isn't affected by rotation or translation, so we will translate the ellipse so that $\mu = \mathbf{0}$ (i.e. the mean is at the origin) and rotate it so that Σ^{-1} is diagonal.

Writing $\mathbf{x} = [x, y]$ we get that the set of points on the ellipse satisfies

$$\frac{1}{2}(\frac{1}{k_1^2}x^2 + \frac{1}{k_2^2}y^2) = \frac{1}{2}$$

where $\frac{1}{k_1^2}$ and $\frac{1}{k_2^2}$ are the diagonal elements of Σ^{-1}. We will assume that the ellipse has been rotated so that $k_1 < k_2$. The points $(k_1, 0)$ and $(-k_1, 0)$ lie on the ellipse, as do the points $(0, k_2)$ and $(0, -k_2)$. The major axis of the ellipse, in this coordinate system, is the x-axis, and the minor axis is the y-axis. In this coordinate system, x and y are independent. If you do a little algebra, you will see that the standard deviation of x is $\mathsf{abs}(k_1)$ and the standard deviation of y is $\mathsf{abs}(k_2)$. So the ellipse is longer in the direction of largest standard deviation and shorter in the direction of smallest standard deviation.

Now rotating the ellipse is means we will pre- and post-multiply the covariance matrix with some rotation matrix. Translating it will move the origin to the mean. As a result, the ellipse has its center at the mean, its major axis is in the direction of the eigenvector of the covariance with largest eigenvalue, and its minor axis is in the direction of the eigenvector with smallest eigenvalue. A plot of this ellipse, which can be coaxed out of most programming environments with relatively little effort, gives us a great deal of information about the underlying Gaussian. These ellipses are known as **covariance ellipses**.

10.2 MIXTURE MODELS AND CLUSTERING

It is natural to think of clustering in the following way. The data was created by a collection of distinct models (one per cluster). For each data item, something (nature?) chose which model was to produce a point, and then the model produced a point. We see the results: crucially, we'd like to know what the models were, but we don't know which model produced which point. If we knew the models, it would be easy to decide which model produced which point. Similarly, if we knew which point went to which model, we could determine what the models were.

One encounters this situation – or problems that can be mapped to this situation – again and again. It is very deeply embedded in clustering problems. It is pretty clear that a natural algorithm is to iterate between estimating which model gets which point, and the model parameters. We have seen this approach before, in the case of k-means.

A particularly interesting case occurs when the models are probabilistic. There is a standard, and very important, algorithm for estimation here, called **EM** (or **expectation maximization**, if you want the long version). I will develop this algorithm in two simple cases, and we will see it in a more general form later.

Notation: This topic lends itself to a glorious festival of indices, limits of sums and products, etc. I will do one example in quite gory detail; the other follows the same form, and for that we'll proceed more expeditiously. Writing the limits of sums or products explicitly is usually even more confusing than adopting a compact notation. When I write $\sum_i$ or $\prod_i$, I mean a sum (or product) over all values of i. When I write $\sum_{i,\hat{j}}$ or $\prod_{i,\hat{j}}$, I mean a sum (or product) over all values of i *except* for the j'th item. I will write vectors, as usual, as $\mathbf{x}$; the i'th such vector in a collection is $\mathbf{x}_i$, and the k'th component of the i'th vector in a collection is x_{ik}.

In what follows, I will construct a vector δ_i corresponding to the i'th data item $\mathbf{x}_i$ (it will tell us what cluster that item belongs to). I will write δ to mean all the δ_i (one for each data item). The j'th component of this vector is δ_{ij}. When I write $\sum_{\delta_u}$, I mean a sum over all values that δ_u can take. When I write $\sum_{\delta}$, I mean a sum over all values that each δ can take. When I write $\sum_{\delta,\hat{\delta_v}}$, I mean a sum over all values that all δ can take, *omitting* all cases for the v'th vector δ_v.

10.2.1 A Finite Mixture of Blobs

A blob of data points is quite easily modelled with a single normal distribution. Obtaining the parameters is straightforward (estimate the mean and covariance matrix with the usual expressions). Now imagine I have t blobs of data, and I know t. A normal distribution is likely a poor model, but I could think of the data as being produced by t normal distributions. I will assume that each normal distribution has a fixed, *known* covariance matrix Σ, but the mean of each is unknown. Because the covariance matrix is fixed, and *known*, we can compute a factorization $\Sigma = \mathcal{A}\mathcal{A}^T$. The factors must have full rank, because the covariance matrix must be positive definite. This means that we can apply $\mathcal{A}^{-1}$ to all the data, so that each blob covariance matrix (and so each normal distribution) is the identity.

Write μ_j for the mean of the j'th normal distribution. We can model a distribution that consists of t distinct blobs by forming a weighted sum of the blobs, where the j'th blob gets weight π_j. We ensure that $\sum_j \pi_j = 1$, so that we can think of the overall model as a probability distribution. We can then model the data as samples from the probability distribution

$$p(\mathbf{x}|\mu_1, \ldots, \mu_k, \pi_1, \ldots, \pi_k) = \sum_j \pi_j \left[\frac{1}{\sqrt{(2\pi)^d}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mu_j)^T(\mathbf{x} - \mu_j) \right) \right].$$

The way to think about this probability distribution is that a point is generated by first choosing one of the normal distributions (the j'th is chosen with probability π_j), then generating a point from that distribution. This is a pretty natural model of clustered data. Each mean is the center of a blob. Blobs with many points in them have a high value of π_j, and blobs with few points have a low value of π_j. We must now use the data points to estimate the values of π_j and μ_j (again, I am assuming that the blobs – and the normal distribution modelling each – have the identity as a covariance matrix). A distribution of this form is known as a **mixture of normal distributions**.

Writing out the likelihood will reveal a problem: we have a product of many sums. The usual trick of taking the log will not work, because then you have a sum of logs of sums, which is hard to differentiate and hard to work with. A much more productive approach is to think about a set of hidden variables which tell us which blob each data item comes from. For the i'th data item, we construct a vector δ_i. The j'th component of this vector is δ_{ij}, where $\delta_{ij} = 1$ if $\mathbf{x}_i$ comes from blob (equivalently, normal distribution) j and zero otherwise. Notice there is exactly one 1 in δ_i, because each data item comes from one blob. I will write δ to mean all the δ_i (one for each data item). Assume we know the values of these terms. I will

write $\theta = (\mu_1, \ldots, \mu_k, \pi_1, \ldots, \pi_k)$ for the unknown parameters. Then we can write

$$p(\mathbf{x}_i|\delta_i, \theta) = \prod_j \left[\frac{1}{\sqrt{(2\pi)^d}} \exp\left(-\frac{1}{2}(\mathbf{x}_i - \mu_j)^T(\mathbf{x}_i - \mu_j) \right) \right]^{\delta_{ij}}$$

(because $\delta_{ij} = 1$ means that $\mathbf{x}_i$ comes from blob j, so the terms in the product are a collection of 1's and the probability we want). We also have

$$p(\delta_{ij} = 1|\theta) = \pi_j$$

allowing us to write

$$p(\delta_i|\theta) = \prod_j [\pi_j]^{\delta_{ij}}$$

(because this is the probability that we select blob j to produce a data item; again, the terms in the product are a collection of 1's and the probability we want). This means that

$$p(\mathbf{x}_i, \delta_i|\theta) = \prod_j \left\{ \left[\frac{1}{\sqrt{(2\pi)^d}} \exp\left(-\frac{1}{2}(\mathbf{x}_i - \mu_j)^T(\mathbf{x}_i - \mu_j) \right) \right] \pi_j \right\}^{\delta_{ij}}$$

and we can write a log-likelihood. The data are the observed values of $\mathbf{x}$ and δ (remember, we pretend we know these; I'll fix this in a moment), and the parameters are the unknown values of $\mu_1, \ldots, \mu_k$ and $\pi_1, \ldots, \pi_k$. We have

$$\begin{aligned} \mathcal{L}(\mu_1, \ldots, \mu_k, \pi_1, \ldots, \pi_k; \mathbf{x}, \delta) &= \mathcal{L}(\theta; \mathbf{x}, \delta) \\ &= \sum_{ij} \left\{ \left[\left(-\frac{1}{2}(\mathbf{x}_i - \mu_j)^T(\mathbf{x}_i - \mu_j) \right) \right] + \log \pi_j \right\} \delta_{ij} + K \end{aligned}$$

where K is a constant that absorbs the normalizing constants for the normal distributions. You should check this expression gives the right answer. I have used the δ_{ij} as a "switch" – for one term, $\delta_{ij} = 1$ and the term in curly brackets is "on", and for all others that term is multiplied by zero. The problem with all this is that we don't know δ. I will deal with this when we have another example.

10.2.2 Topics and Topic Models

We cluster documents together if they come from the same topic. Imagine we know which document comes from which topic. Then we could estimate the word probabilities using the documents in each topic. Now imagine we know the word probabilities for each topic. Then we could tell (at least in principle) which topic a document comes from by looking at the probability each topic generates the document, and choosing the topic with the highest probability. This should strike you as being a circular argument. It has a form you should recognize from k-means, though the details of the distance have changed.

To construct a probabilistic model, we will assume that a document is generated in two steps. We will have t topics. First, we choose a topic, choosing the j'th topic with probability π_j. Then we will obtain a set of words by repeatedly drawing

IID samples from that topic, and record the count of each word in a count vector. Assume we have N vectors of word counts, and write $\mathbf{x}_i$ for the i'th such vector. Each topic is a multinomial probability distribution. Write $\mathbf{p}_j$ for the vector of word probabilities for the j'th topic. We assume that words are generated independently, conditioned on the topic. Write x_{ik} for the k'th component of $\mathbf{x}_i$, and so on. Notice that $\mathbf{x}_i^T\mathbf{1}$ is the sum of entries in $\mathbf{x}_i$, and so the number of words in document i. Then the probability of observing the counts in $\mathbf{x}_i$ when the document was generated by topic j is

$$p(\mathbf{x}_i|\mathbf{p}_j) = \left(\frac{(\mathbf{x}_i^T\mathbf{1})!}{\prod_v x_{iv}!}\right)\prod_u p_{ju}^{x_{iu}}.$$

We can now write the probability of observing a document. Again, we write $\theta = (\mathbf{p}_1, \ldots, \mathbf{p}_t, \pi_1, \ldots, \pi_t)$ for the vector of unknown parameters. We have

$$\begin{aligned} p(\mathbf{x}_i|\theta) &= \sum_l p(\mathbf{x}_i|\text{topic is l})p(\text{topic is l}|\theta) \\ &= \sum_l \left[\left(\frac{(\mathbf{x}_i^T\mathbf{1})!}{\prod_v x_{iv}!}\right)\prod_u p_{lu}^{x_{iu}}\right]\pi_l. \end{aligned}$$

This model is widely called a **topic model**; be aware that there are many kinds of topic model, and this is a simple one. The expression should look unpromising, in a familiar way. If you write out a likelihood, you will see a product of sums; and if you write out a log-likelihood, you will see a sum of logs of sums. Neither is enticing. We could use the same trick we used for a mixture of normals. Write $\delta_{ij} = 1$ if $\mathbf{x}_i$ comes from topic j, and $\delta_{ij} = 0$ otherwise. Then we have

$$p(\mathbf{x}_i|\delta_{ij} = 1, \theta) = \left[\left(\frac{(\mathbf{x}_i^T\mathbf{1})!}{\prod_v x_{iv}!}\right)\prod_u p_{ju}^{x_{iu}}\right]$$

(because $\delta_{ij} = 1$ means that $\mathbf{x}_i$ comes from topic j). This means we can write

$$p(\mathbf{x}_i|\delta_i, \theta) = \prod_j \left\{\left[\left(\frac{(\mathbf{x}_i^T\mathbf{1})!}{\prod_v x_{iv}!}\right)\prod_u p_{ju}^{x_{iu}}\right]\right\}^{\delta_{ij}}$$

(because $\delta_{ij} = 1$ means that $\mathbf{x}_i$ comes from topic j, so the terms in the product are a collection of 1's and the probability we want). We also have

$$p(\delta_{ij} = 1|\theta) = \pi_j$$

(because this is the probability that we select topic j to produce a data item), allowing us to write

$$p(\delta_i|\theta) = \prod_j [\pi_j]^{\delta_{ij}}$$

(again, the terms in the product are a collection of 1's and the probability we want). This means that

$$p(\mathbf{x}_i, \delta_i|\theta) = \prod_j \left[\left(\frac{(\mathbf{x}_i^T\mathbf{1})!}{\prod_v x_{iv}!}\right)\prod_u \left(p_{ju}^{x_{iu}}\right)\pi_j\right]^{\delta_{ij}}$$

and we can write a log-likelihood. The data are the observed values of $\mathbf{x}$ and δ (remember, we pretend we know these for the moment), and the parameters are the unknown values collected in θ. We have

$$\mathcal{L}(\theta; \mathbf{x}, \delta) = \sum_i \left\{ \sum_j \left[\sum_u x_{iu} \log p_{ju} + \log \pi_j \right] \delta_{ij} \right\} + K$$

where K is a term that contains all the

$$\log \left(\frac{(\mathbf{x}_i^T \mathbf{1})!}{\prod_v x_{iv}!} \right)$$

terms. This is of no interest to us, because it doesn't depend on any of our parameters. It takes a fixed value for each dataset. You should check this expression, noticing that, again, I have used the δ_{ij} as a "switch" – for one term, $\delta_{ij} = 1$ and the term in curly brackets is "on", and for all others that term is multiplied by zero. The problem with all this, as before, is that we don't know δ_{ij}. But there is a recipe.

10.3 THE EM ALGORITHM

There is a straightforward, natural, and very powerful recipe. In essence, we will average out the things we don't know. But this average will depend on our estimate of the parameters, so we will average, then re-estimate parameters, then re-average, and so on. If you lose track of what's going on here, think of the example of k-means with soft weights (section 10.1; this is what the equations for the case of a mixture of normals will boil down to). In this analogy, the δ tell us which cluster center a data item came from. Because we don't know the values of the δ, we assume we have a set of cluster centers; these allow us to make a soft estimate of the δ; then we use this estimate to re-estimate the centers; and so on.

This is an instance of a general recipe. Recall we wrote θ for a vector of parameters. In the mixture of normals case, θ contained the means and the mixing weights; in the topic model case, it contained the topic distributions and the mixing weights. Assume we have an estimate of the value of this vector, say $\theta^{(n)}$. We could then compute $p(\delta|\theta^{(n)}, \mathbf{x})$. In the mixture of normals case, this is a guide to which example goes to which cluster. In the topic case, it is a guide to which example goes to which topic.

We could use this to compute the expected value of the likelihood with respect to δ. We compute

$$\mathcal{Q}(\theta; \theta^{(n)}) = \sum_{\delta} \mathcal{L}(\theta; \mathbf{x}, \delta) p(\delta|\theta^{(n)}, \mathbf{x})$$

(where the sum is over all values of δ). Notice that $Q(\theta; \theta^{(n)})$ is a *function* of θ (because $\mathcal{L}$ was), but now does not have any unknown δ terms in it. This $Q(\theta; \theta^{(n)})$ encodes what we know about δ.

For example, assume that $p(\delta|\theta^{(n)}, \mathbf{x})$ has a single, narrow peak in it, at (say) $\delta = \delta^0$. In the mixture of normals case, this would mean that there is one allocation of points to clusters that is significantly better than all others, given $\theta^{(n)}$. For this example, $Q(\theta; \theta^{(n)})$ will be approximately $\mathcal{L}(\theta; \mathbf{x}, \delta^0)$.

Now assume that $p(\delta|\theta^{(n)}, \mathbf{x})$ is about uniform. In the mixture of normals case, this would mean that any particular allocation of points to clusters is about as good as any other. For this example, $Q(\theta; \theta^{(n)})$ will average $\mathcal{L}$ over all possible δ values with about the same weight for each.

We obtain the next estimate of θ by computing

$$\theta^{(n+1)} = \underset{\theta}{\text{argmax}}\ Q(\theta; \theta^{(n)})$$

and iterate this procedure until it converges (which it does, though I shall not prove that). The algorithm I have described is extremely general and powerful, and is known as **expectation maximization** or (more usually) **EM**. The step where we compute $Q(\theta; \theta^{(n)})$ is called the **E step**; the step where we compute the new estimate of θ is known as the **M step**.

One trick to be aware of: it is quite usual to ignore additive constants in the log-likelihood, because they have no effect. When you do the E-step, taking the expectation of a constant gets you a constant; in the M-step, the constant can't change the outcome. As a result, I will tend to be careless about it. In the mixture of normals example, below, I've tried to keep track of it; for the mixture of multinomials, I've ignored it.

10.3.1 Example: Mixture of Normals: The E-step

Now let us do the actual calculations for a mixture of normal distributions. The E step requires a little work. We have

$$\mathcal{Q}(\theta; \theta^{(n)}) = \sum_{\delta} \mathcal{L}(\theta; \mathbf{x}, \delta) p(\delta|\theta^{(n)}, \mathbf{x})$$

If you look at this expression, it should strike you as deeply worrying. There are a very large number of different possible values of δ. In this case, there are $N \times t$ cases (there is one δ_i for each data item, and each of these can have a one in each of t locations). It isn't obvious how we could compute this average.

But notice

$$p(\delta|\theta^{(n)}, \mathbf{x}) = \frac{p(\delta, \mathbf{x}|\theta^{(n)})}{p(\mathbf{x}|\theta^{(n)})}$$

and let us deal with numerator and denominator separately. For the numerator, notice that the $\mathbf{x}_i$ and the δ_i are independent, identically distributed samples, so that

$$p(\delta, \mathbf{x}|\theta^{(n)}) = \prod_i p(\delta_i, \mathbf{x}_i|\theta^{(n)}).$$

The denominator is slightly more work. We have

$$\begin{aligned} p(\mathbf{x}|\theta^{(n)}) &= \sum_{\delta} p(\delta, \mathbf{x}|\theta^{(n)}) \\ &= \sum_{\delta} \left[\prod_i p(\delta_i, \mathbf{x}_i|\theta^{(n)}) \right] \\ &= \prod_i \left[\sum_{\delta_i} p(\delta_i, \mathbf{x}_i|\theta^{(n)}) \right]. \end{aligned}$$

You should check the last step; one natural thing to do is check with $N = 2$ and $t = 2$. This means that we can write

$$\begin{aligned} p(\delta|\theta^{(n)}, \mathbf{x}) &= \frac{p(\delta, \mathbf{x}|\theta^{(n)})}{p(\mathbf{x}|\theta^{(n)})} \\ &= \frac{\prod_i p(\delta_i, \mathbf{x}_i|\theta^{(n)})}{\prod_i \left[\sum_{\delta_i} p(\delta_i, \mathbf{x}_i|\theta^{(n)})\right]} \\ &= \prod_i \frac{p(\delta_i, \mathbf{x}_i|\theta^{(n)})}{\sum_{\delta_i} p(\delta_i, \mathbf{x}_i|\theta^{(n)})} \\ &= \prod_i p(\delta_i|\mathbf{x}_i, \theta^{(n)}) \end{aligned}$$

Now we need to look at the log-likelihood. We have

$$\mathcal{L}(\theta; \mathbf{x}, \delta) = \sum_{ij} \left\{ \left[\left(-\frac{1}{2}(\mathbf{x}_i - \mu_j)^T (\mathbf{x}_i - \mu_j) \right) \right] + \log \pi_j \right\} \delta_{ij} + K.$$

The K term is of no interest – it will result in a constant – but we will try to keep track of it. To simplify the equations we need to write, I will construct a t dimensional vector $\mathbf{c}_i$ for the i'th data point. The j'th component of this vector will be

$$\left\{ \left[\left(-\frac{1}{2}(\mathbf{x}_i - \mu_j)^T (\mathbf{x}_i - \mu_j) \right) \right] + \log \pi_j \right\}$$

so we can write

$$\mathcal{L}(\theta; \mathbf{x}, \delta) = \sum_i \mathbf{c}_i^T \delta_i + K.$$

Now all this means that

$$\begin{aligned} \mathcal{Q}(\theta; \theta^{(n)}) &= \sum_{\delta} \mathcal{L}(\theta; \mathbf{x}, \delta) p(\delta|\theta^{(n)}, \mathbf{x}) \\ &= \sum_{\delta} \left(\sum_i \mathbf{c}_i^T \delta_i + K \right) p(\delta|\theta^{(n)}, \mathbf{x}) \\ &= \sum_{\delta} \left(\sum_i \mathbf{c}_i^T \delta_i + K \right) \prod_u p(\delta_u|\theta^{(n)}, \mathbf{x}) \\ &= \sum_{\delta} \left(\mathbf{c}_1^T \delta_1 \prod_u p(\delta_u|\theta^{(n)}, \mathbf{x}) + \ldots \mathbf{c}_N^T \delta_N \prod_u p(\delta_u|\theta^{(n)}, \mathbf{x}) \right). \end{aligned}$$

We can simplify further. We have that $\sum_{\delta_i} p(\delta_i|\mathbf{x}_i, \theta^{(n)}) = 1$, because this is a

probability distribution. Notice that, for any index v,

$$\begin{aligned}\sum_{\delta}\left(\mathbf{c}_v^T\delta_v\prod_u p(\delta_u|\theta^{(n)},\mathbf{x})\right) &= \sum_{\delta_v}\left(\mathbf{c}_v^T\delta_v p(\delta_v|\theta^{(n)},\mathbf{x})\right)\left[\sum_{\delta,\,\hat{\delta_v}}\prod_{u,\hat{v}} p(\delta_u|\theta^{(n)},\mathbf{x})\right]\\ &= \sum_{\delta_v}\left(\mathbf{c}_v^T\delta_v p(\delta_v|\theta^{(n)},\mathbf{x})\right)\end{aligned}$$

So we can write

$$\begin{aligned}\mathcal{Q}(\theta;\theta^{(n)}) &= \sum_{\delta}\mathcal{L}(\theta;\mathbf{x},\delta)p(\delta|\theta^{(n)},\mathbf{x})\\ &= \sum_i\left[\sum_{\delta_i}\mathbf{c}_i^T\delta_i p(\delta_i|\theta^{(n)},\mathbf{x})\right]+K\\ &= \sum_i\left[\left(\sum_j\left\{\left[\left(-\frac{1}{2}(\mathbf{x}_i-\mu_j)^T(\mathbf{x}_i-\mu_j)\right)+\log\pi_j\right]w_{ij}\right\}\right)\right]+K\end{aligned}$$

where

$$\begin{aligned}w_{ij} &= 1p(\delta_{ij}=1|\theta^{(n)},\mathbf{x})+0p(\delta_{ij}=0|\theta^{(n)},\mathbf{x})\\ &= p(\delta_{ij}=1|\theta^{(n)},\mathbf{x}).\end{aligned}$$

Now

$$\begin{aligned}p(\delta_{ij}=1|\theta^{(n)},\mathbf{x}) &= \frac{p(\mathbf{x},\delta_{ij}=1|\theta^{(n)})}{p(\mathbf{x}|\theta^{(n)})}\\ &= \frac{p(\mathbf{x},\delta_{ij}=1|\theta^{(n)})}{\sum_l p(\mathbf{x},\delta_{il}=1|\theta^{(n)})}\\ &= \frac{p(\mathbf{x}_i,\delta_{ij}=1|\theta^{(n)})\prod_{u,\hat{i}}p(\mathbf{x}_u,\delta_u|\theta)}{\left(\sum_l p(\mathbf{x},\delta_{il}=1|\theta^{(n)})\right)\prod_{u,\hat{i}}p(\mathbf{x}_u,\delta_u|\theta)}\\ &= \frac{p(\mathbf{x}_i,\delta_{ij}=1|\theta^{(n)})}{\sum_l p(\mathbf{x},\delta_{il}=1|\theta^{(n)})}\end{aligned}$$

If the last couple of steps puzzle you, remember we obtained $p(\mathbf{x},\delta|\theta)=\prod_i p(\mathbf{x}_i,\delta_i|\theta)$. Also, look closely at the denominator; it expresses the fact that the data must have come from somewhere. So the main question is to obtain $p(\mathbf{x}_i,\delta_{ij}=1|\theta^{(n)})$. But

$$\begin{aligned}p(\mathbf{x}_i,\delta_{ij}=1|\theta^{(n)}) &= p(\mathbf{x}_i|\delta_{ij}=1,\theta^{(n)})p(\delta_{ij}=1|\theta^{(n)})\\ &= = \left[\frac{1}{\sqrt{(2\pi)^d}}\exp\left(-\frac{1}{2}(\mathbf{x}_i-\mu_j)^T(\mathbf{x}_i-\mu_j)\right)\right]\pi_j.\end{aligned}$$

Substituting yields

$$p(\delta_{ij}=1|\theta^{(n)},\mathbf{x})=\frac{\left[\exp\left(-\frac{1}{2}(\mathbf{x}_i-\mu_j)^T(\mathbf{x}_i-\mu_j)\right)\right]\pi_j}{\sum_k\left[\exp\left(-\frac{1}{2}(\mathbf{x}_i-\mu_k)^T(\mathbf{x}_i-\mu_k)\right)\right]\pi_k}=w_{ij}.$$

10.3.2 Example: Mixture of Normals: The M-step

The M-step is more straightforward. Recall

$$\mathcal{Q}(\theta;\theta^{(n)}) = \left(\sum_{ij}\left\{\left[\left(-\frac{1}{2}(\mathbf{x}_i-\mu_j)^T(\mathbf{x}_i-\mu_j)\right)\right]+\log\pi_j\right\}w_{ij}+K\right)$$

and we have to maximise this with respect to μ and π, and the terms w_{ij} are known. This maximization is easy. We compute

$$\mu_j^{(n+1)} = \frac{\sum_i x_i w_{ij}}{\sum_i w_{ij}}$$

and

$$\pi_j^{(n+1)} = \frac{\sum_i w_{ij}}{N}.$$

You should check these expressions by differentiating and setting to zero. When you do so, remember that, because π is a probability distribution, $\sum_j \pi_j = 1$ (otherwise you'll get the wrong answer).

10.3.3 Example: Topic Model: The E-Step

We need to work out two steps. The E step requires a little calculation. We have

$$\begin{aligned}
\mathcal{Q}(\theta;\theta^{(n)}) &= \sum_\delta \mathcal{L}(\theta;\mathbf{x},\delta)p(\delta|\theta^{(n)},\mathbf{x}) \\
&= \sum_\delta\left(\sum_{ij}\left\{\left[\sum_u x_{iu}\log p_{ju}\right]+\log\pi_j\right\}\delta_{ij}\right)p(\delta|\theta^{(n)},\mathbf{x}) \\
&= \left(\sum_{ij}\left\{\left[\sum_k x_{i,k}\log p_{j,k}\right]+\log\pi_j\right\}w_{ij}\right)
\end{aligned}$$

Here the last two steps follow from the same considerations as in the mixture of normals. The $\mathbf{x}_i$ and δ_i are IID samples, and so the expectation simplifies as in that case. If you're uncertain, rewrite the steps of section 10.3.1. The form of this Q function is the same as that (a sum of $\mathbf{c}_i^T\delta_i$ terms, but using a different expression for $\mathbf{c}_i$). In this case, as above,

$$\begin{aligned}
w_{ij} &= 1p(\delta_{ij}=1|\theta^{(n)},\mathbf{x}) + 0p(\delta_{ij}=0|\theta^{(n)},\mathbf{x}) \\
&= p(\delta_{ij}=1|\theta^{(n)},\mathbf{x}).
\end{aligned}$$

Again, we have

$$\begin{aligned}
p(\delta_{ij}=1|\theta^{(n)},\mathbf{x}) &= \frac{p(\mathbf{x}_i,\delta_{ij}=1|\theta^{(n)})}{p(\mathbf{x}_i|\theta^{(n)})} \\
&= \frac{p(\mathbf{x}_i,\delta_{ij}=1|\theta^{(n)})}{\sum_l p(\mathbf{x}_i,\delta_{il}=1|\theta^{(n)})}
\end{aligned}$$

and so the main question is to obtain $p(\mathbf{x}_i, \delta_{ij} = 1|\theta^{(n)})$. But

$$\begin{aligned} p(\mathbf{x}_i, \delta_{ij} = 1|\theta^{(n)}) &= p(\mathbf{x}_i|\delta_{ij} = 1, \theta^{(n)})p(\delta_{ij} = 1|\theta^{(n)}) \\ &= = \left[\prod_k p_{j,k}^{x_k}\right] \pi_j. \end{aligned}$$

Substituting yields

$$p(\delta_{ij} = 1|\theta^{(n)}, \mathbf{x}) = \frac{\left[\prod_k p_{j,k}^{x_k}\right] \pi_j}{\sum_l \left[\prod_k p_{l,k}^{x_k}\right] \pi_l}$$

10.3.4 Example: Topic Model: The M-step

The M-step is more straightforward. Recall

$$\mathcal{Q}(\theta; \theta^{(n)}) = \left(\sum_{ij} \left\{ \left[\sum_k x_{i,k} \log p_{j,k} \right] + \log \pi_j \right\} w_{ij} \right)$$

and we have to maximise this with respect to μ and π, and the terms w_{ij} are known. This maximization is easy, but remember that the probabilities sum to one, so you need either to use a Lagrange multiplier or to set one probability to $(1 -$ all others$)$. You should get

$$\mathbf{p}_j^{(n+1)} = \frac{\sum_i \mathbf{x}_i w_{ij}}{\sum_i \mathbf{x}_i^T \mathbf{1} w_{ij}}$$

and

$$\pi_j^{(n+1)} = \frac{\sum_i w_{ij}}{N}.$$

You should check these expressions by differentiating and setting to zero.

10.3.5 EM in Practice

The algorithm we have seen is amazingly powerful; I will use it again, ideally with less notation. One could reasonably ask whether it produces a "good" answer. Slightly surprisingly, the answer is yes. The algorithm produces a local minimum of $p(\mathbf{x}|\theta)$, the likelihood of the data conditioned on parameters. This is rather surprising because we engaged in all the activity with δ to avoid directly dealing with this likelihood (which in our cases was an unattractive product of sums). I did not prove this, but it's true anyway.

There are some practical issues. First, how many cluster centers should there be? Mostly, the answer is a practical one. We are usually clustering data for a reason (vector quantization is a really good reason), and then we search for a k that yields the best results.

Second, how should one start the iteration? This depends on the problem you want to solve, but for the two cases I have described, a rough clustering using k-means usually provides an excellent start. In the mixture of normals problem, you can take the cluster centers as initial values for the means, and the fraction of points in each cluster as initial values for the mixture weights. In the topic model

problem, you can cluster the count vectors with k-means, use the overall counts within a cluster to get an initial estimate of the multinomial model probabilities, and use the fraction of documents within a cluster to get mixture weights. You need to be careful here, though. You really don't want to initialize a topic probability with a zero value for any word (otherwise no document containing that word can ever go into the cluster, which is a bit extreme). For our purposes, it will be enough to allocate a small value to each zero count, then adjust all the word probabilies to be sure they sum to one. More complicated approaches are possible.

Third, we need to avoid numerical problems in the implementation. Notice that you will be evaluating terms that look like

$$\frac{\pi_k e^{-(\mathbf{x}_i-\mu_k)^T(\mathbf{x}_i-\mu_k)/2}}{\sum_u \pi_u e^{-(\mathbf{x}_i-\mu_u)^T(\mathbf{x}_i-\mu_u)/2}}.$$

Imagine you have a point that is far from all cluster means. If you just blithely exponentiate the negative distances, you could find yourself dividing zero by zero, or a tiny number by a tiny number. This can lead to trouble. There's an easy alternative. Find the center the point is closest to. Now subtract the square of this distance (d^2_{min} for concreteness) from all the distances. Then evaluate

$$\frac{\pi_k e^{-\left[(\mathbf{x}_i-\mu_k)^T(\mathbf{x}_i-\mu_k)-d^2_{\text{min}}\right]/2}}{\sum_u \pi_u e^{-\left[(\mathbf{x}_i-\mu_u)^T(\mathbf{x}_i-\mu_u)-d^2_{\text{min}}\right]/2}}$$

which is a better way of estimating the same number (notice the $e^{-d^2_{\text{min}}/2}$ terms cancel top and bottom).

The last problem is more substantial. EM will get me to a local minimum of $p(\mathbf{x}|\theta)$, but there might be more than one local minimum. For clustering problems, the usual case is there are lots of them. One doesn't really expect a clustering problem to have a single best solution, as opposed to a lot of quite good solutions. Points that are far from all clusters are a particular source of local minima; placing these points in different clusters yields somewhat different sets of cluster centers, each about as good as the other. It's not usual to worry much about this point. A natural strategy is to start the method in a variety of different places (use k means with different start points), and choose the one that has the best value of Q when it has converged.

However, EM isn't magic. There are problems where computing the expectation is hard, typically because you have to sum over a large number of cases which don't have the nice independence structure that helped in the examples I showed. There are strategies for dealing with this problem — essentially, you can get away with an approximate expectation – but they're beyond our reach at present.

10.4 YOU SHOULD

10.4.1 remember:

PROGRAMMING EXERCISES

10.1. Obtain the activities of daily life dataset from the UC Irvine machine learning website (https://archive.ics.uci.edu/ml/datasets/Dataset+for+ADL+Recognition+with+Wrist-worn+Acceleron data provided by Barbara Bruno, Fulvio Mastrogiovanni and Antonio Sgorbissa).

(a) Build a classifier that classifies sequences into one of the 14 activities provided. To make features, you should vector quantize, then use a histogram of cluster centers (as described in the subsection; this gives a pretty explicit set of steps to follow). You will find it helpful to use hierarchical k-means to vector quantize. You may use whatever multi-class classifier you wish, though I'd start with R's decision forest, because it's easy to use and effective. You should report (a) the total error rate and (b) the class confusion matrix of your classifier.

(b) Now see if you can improve your classifier by (a) modifying the number of cluster centers in your hierarchical k-means and (b) modifying the size of the fixed length samples that you use.

CHAPTER 11

Mean Field Inference

Bayesian inference is an important and useful tool, but it comes with a serious practical problem. It will help to have some notation. Write X for a set of observed values, H for the unknown (hidden) values of interest, and recall Bayes' rule has

$$P(H|X) = \frac{P(X|H)P(H)}{P(X)} = \frac{\text{Likelihood} \times \text{Prior}}{\text{Normalizing constant}}.$$

The problem is that it is usually very difficult to form posterior distributions, because the normalizing constant is hard to evaluate for almost every model. This point is easily dodged in first courses. For MAP inference, we can ignore the normalizing constant. A careful choice of problem and of conjugate prior can make things look easy (or, at least, hide the real difficulty). But most of the time we cannot compute

$$P(X) = \int P(X|H)P(H)dX.$$

Either the integral is too hard, or – in the case of discrete models – the marginalization requires an unmanageable sum. In such cases, we must approximate.

Warning: The topics of this chapter allow a great deal of room for mathematical finicking, which I shall try to avoid. Generally, when I define something I'm going to leave out the information that it's only meaningful under some circumstances, etc. None of the background detail I'm eliding is difficult or significant for anything we do. Those who enjoy this sort of thing can supply the ifs ands and buts without trouble; those who don't won't miss them. I will usually just write an integral sign for marginalization, and I'll assume that, when the variables are discrete, everyone's willing to replace with a sum.

11.1 USEFUL BUT INTRACTABLE EXAMPLES

11.1.1 Boltzmann Machines

Here is a formal model we can use. A **Boltzmann machine** is a distribution model for a set of binary random variables. Assume we have N binary random variables U_i, which take the values 1 or -1. The values of these random variables are not observed (the true values of the pixels). These binary random variables are not independent. Instead, we will assume that some (but not all) pairs are coupled. We could draw this situation as a graph (Figure 11.1), where each node represents a U_i and each edge represents a coupling. The edges are weighted, so the coupling strengths vary from edge to edge.

Write $\mathcal{N}(i)$ for the set of random variables whose values are coupled to that

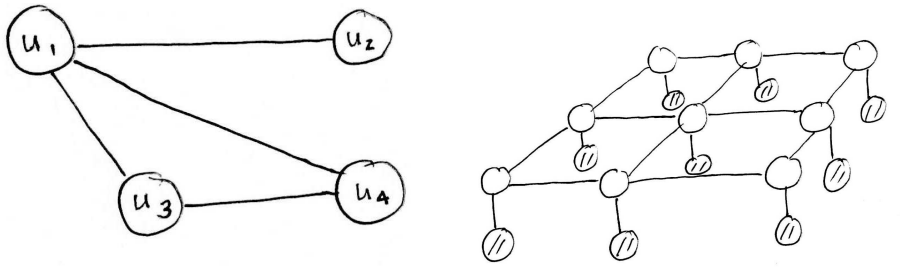


FIGURE 11.1: *On the* **left**, *a simple Boltzmann machine. Each U_i has two possible states, so the whole thing has 16 states. Different choices of the constants coupling the U's along each edge lead to different probability distributions. On the* **right**, *this Boltzmann machine adapted to denoising binary images. The shaded nodes represent the known pixel values (X_i in the text) and the open nodes represent the (unknown, and to be inferred) true pixel values H_i. Notice that pixels depend on their neighbors in the grid.*

of i – these are the neighbors of i in the graph. The joint probability model is

$$\log P(U|\theta) = \left[\sum_i \sum_{j \in \mathcal{N}(i)} \theta_{ij} U_i U_j\right] - \log Z(\theta) = -E(U|\theta) - \log Z(\theta).$$

Now $U_i U_j$ is 1 when U_i and U_j agree, and -1 otherwise (this is why we chose U_i to take values 1 or -1). The θ_{ij} are the edge weights; notice if $\theta_{ij} > 0$, the model generally prefers U_i and U_j to agree (as in, it will assign higher probability to states where they agree, unless other variables intervene), and if $\theta_{ij} < 0$, the model prefers they disagree.

Here $E(U|\theta)$ is sometimes referred to as the **energy** (notice the sign - higher energy corresponds to lower probability) and $Z(\theta)$ ensures that the model normalizes to 1, so that

$$Z(\theta) = \sum_{\text{all values of U}} \left[\exp\left(-E(U|\theta)\right)\right].$$

11.1.2 Denoising Binary Images with Boltzmann Machines

Here is a simple model for a binary image that has been corrupted by noise. At each pixel, we observe the corrupted value, which is binary. Hidden from us are the true values of each pixel. The observed value at each pixel is random, but depends only on the true value. This means that, for example, the value at a pixel can change, but the noise doesn't cause blocks of pixels to, say, shift left. This is a fairly good model for many kinds of transmission noise, scanning noise, and so on. The true value at each pixel is affected by the true value at each of its neighbors – a reasonable model, as image pixels tend to agree with their neighbors.

We can apply a Boltzmann machine. We split the U into two groups. One group represents the observed value at each pixel (I will use X_i, and the convention that i chooses the pixel), and the other represents the hidden value at each pixel (I will use H_i). Each observation is either 1 or -1. We arrange the graph so that the edges between the H_i form a grid, and there is a link between each X_i and its corresponding H_i (but no other - see Figure 11.1).

Assume we know good values for θ. We have

$$P(H|X,\theta) = \frac{\exp(-E(H,X|\theta))/Z(\theta)}{\Sigma_H \left[\exp(-E(H,X|\theta))/Z(\theta)\right]} = \frac{\exp\left(-E(H,X|\theta)\right)}{\Sigma_H \exp\left(-E(H,X|\theta)\right)}$$

so posterior inference doesn't require evaluating the normalizing constant. This isn't really good news. Posterior inference still requires a sum over an exponential number of values. Unless the underlying graph is special (a tree or a forest) or very small, posterior inference is intractable.

11.1.3 MAP Inference for Boltzmann Machines is Hard

You might think that focusing on MAP inference will solve this problem. Recall that MAP inference seeks the values of H to maximize $P(H|X,\theta)$ or equivalently, maximizing the log of this function. We seek

$$\underset{H}{\text{argmax}} \ \log P(H|X,\theta) = (-E(H,X|\theta)) - \log\left[\Sigma_H \exp\left(-E(H,X|\theta)\right)\right]$$

but the second term is not a function of H, so we could avoid the intractable sum. This doesn't mean the problem is tractable. Some pencil and paper work will establish that there is some set of constants a_{ij} and b_j so that the solution is obtained by solving

$$\underset{H}{\text{argmax}} \ \left(\sum_{ij} a_{ij} h_i h_j\right) + \sum_j b_j h_j \quad \text{subject to } h_i \in \{-1, 1\}.$$

This is a combinatorial optimization problem with considerable potential for unpleasantness. How nasty it is depends on some details of the a_{ij}, but with the right choice of weights a_{ij}, the problem is **max-cut**, which is NP-complete.

11.1.4 A Discrete Markov Random Field

Boltzmann machines are a simple version of a much more complex device widely used in computer vision and other applications. In a Boltzmann machine, we took a graph and associated a binary random variable with each node and a coupling weight with each edge. This produced a probability distribution. We obtain a **Markov random field** by placing a random variable (doesn't have to be binary, or even discrete) at each node, and a coupling function (almost anything works) at each edge. Write U_i for the random variable at the i'th node, and $\theta(U_i, U_j)$ for the coupling function associated with the edge from i to j (the arguments tell you which function; you can have different functions on different edges).

We will ignore the possibility that the random variables are continuous. A **discrete Markov random field** has all U_i discrete random variables with a finite set of possible values. Write U_i for the random variable at each node, and $\theta(U_i, U_j)$ for the coupling function associated with the edge from i to j (the arguments tell you which function; you can have different functions on different edges). For a discrete Markov random field, we have

$$\log P(U|\theta) = \left[\sum_i \sum_{j \in \mathcal{N}(i)} \theta(U_i, U_j)\right] - \log Z(\theta).$$

It is usual – and a good idea – to think about the random variables as indicator functions, rather than values. So, for example, if there were three possible values at node i, we represent U_i with a 3D vector containing one indicator function for each value. One of the components must be one, and the other two must be zero. Vectors like this are sometimes know as **one-hot vectors**. The advantage of this representation is that it helps keep track of the fact that the *values* that each random variable can take are not really to the point; it's the *interaction* between assignments that matters. Another advantage is that we can easily keep track of the parameters that matter. I will adopt this convention in what follows.

I will write $\mathbf{u}_i$ for the random variable at location i represented as a vector. All but one of the components of this vector are zero, and the remaining component is 1. If there are $\#(U_i)$ possible values for U_i and $\#(U_j)$ possible values for U_j, we can represent $\theta(U_i, U_j)$ as a $\#(U_i) \times \#(U_j)$ table of values. I will write $\Theta^{(ij)}$ for the table representing $\theta(U_i, U_j)$, and $\theta^{(ij)}_{mn}$ for the m, n'th entry of that table. This entry is the value of $\theta(U_i, U_j)$ when U_i takes its m'th value and U_j takes its n'th value. I write $\Theta^{(ij)}$ for a matrix whose m, n'th component is $\theta^{(ij)}_{mn}$. In this notation, I write

$$\theta(U_i, U_j) = \mathbf{u}_i^T \Theta^{(ij)} \mathbf{u}_j.$$

All this does not simplify computation of the normalizing constant. We have

$$Z(\theta) = \sum_{\text{all values of } \mathbf{u}} \left[\exp\left(\sum_i \sum_{j \in \mathcal{N}(i)} \mathbf{u}_i^T \Theta^{(ij)} \mathbf{u}_j\right)\right].$$

Note that the collection of all values of $\mathbf{u}$ has rather nasty structure, and is very big – it consists of all possible one-hot vectors representing each U.

11.1.5 Denoising and Segmenting with Discrete MRF's

A simple denoising model for images that aren't binary is just like the binary denoising model. We now use a discrete MRF. We split the U into two groups, H and X. We observe a noisy image (the X values) and we wish to reconstruct the true pixel values (the H). For example, if we are dealing with grey level images with 256 different possible grey values at each pixel, then each H has 256 possible values. The graph is a grid for the H and one link from an X to the corresponding H (like Figure 11.1). Now we think about $P(H|X, \theta)$. As you would expect, the model is intractable – the normalizing constant can't be computed.

Worked example 11.1 *A simple discrete MRF for image denoising.*

Set up an MRF for grey level image denoising.

Solution: Construct a graph that is a grid. The grid represents the true value of each pixel, which we expect to be unknown. Now add an extra node for each grid element, and connect that node to the grid element. These nodes represent the observed value at each pixel. As before, we will separate the variables U into two sets, X for observed values and H for hidden values (Figure 11.1). In most grey level images, pixels take one of 256 ($= 2^8$) values. For the moment, we work with a grey level image, so each variable takes one of 256 values. There is no reason to believe that any one pixel behaves differently from any other pixel, so we expect the $\theta(H_i, H_j)$ not to depend on the pixel location; there'll be one copy of the same function at each grid edge. By far the most usual case has

$$\theta(H_i, H_j) = \begin{bmatrix} 0 & \text{if } H_i = H_j \\ c & \text{otherwise} \end{bmatrix}$$

where $c > 0$. Representing this function using one-hot vectors is straightforward. There is no reason to believe that the relationship between observed and hidden values depends on the pixel location. However, large differences between observed and hidden values should be more expensive than small differences. Write X_j for the observed value at node j, where j is the observed value node corresponding to H_i. We usually have

$$\theta(H_i, X_j) = (H_i - X_j)^2.$$

If we think of H_i as an indicator function, then this function can be represented as a vector of values; one of these values is picked out by the indicator. Notice there is a different vector at each H_i node (because there may be a different X_i at each).

Now write $\mathbf{h}_i$ for the hidden variable at location i represented as a vector, etc. Remember, all but one of the components of this vector are zero, and the remaining component is 1. The one-hot vector representing an observed value at location i is $\mathbf{x}_i$. I write $\Theta^{(ij)}$ for a matrix who's m, n'th component is $\theta^{(ij)}_{mn}$. In this notation, I write

$$\theta(H_i, H_j) = \mathbf{h}_i^T \Theta^{(ij)} \mathbf{h}_j$$

and

$$\theta(H_i, X_j) = \mathbf{h}_i^T \Theta^{(ij)} \mathbf{x}_j = \mathbf{h}_i^T \beta_i.$$

In turn, we have

$$\log p(H|X) = \left[\left(\sum_{ij} \mathbf{h}_i^T \Theta^{(ij)} \mathbf{h}_j \right) + \sum_i \mathbf{h}_i^T \beta_i \right] + \log Z.$$

Worked example 11.2 *Denoising MRF - II*

Write out $\Theta^{(ij)}$ for the $\theta(H_i, H_j)$ with the form given in example 11.1 using the one-hot vector notation.

Solution: This is more a check you have the notation. $c\mathcal{I}$ is the answer.

Worked example 11.3 *Denoising MRF - III*

Assume that we have $X_1 = 128$ and $\theta(H_i, X_j) = (H_i - X_j)^2$. What is β_1 using the one-hot vector notation? Assume pixels take values in the range $[0, 255]$.

Solution: Again, a check you have the notation. We have

$$\beta_1 = \begin{pmatrix} 128^2 & \text{first component} \\ \dots & \\ (i-128)^2 & i\text{'th component} \\ \dots & \\ 127^2 & \end{pmatrix}$$

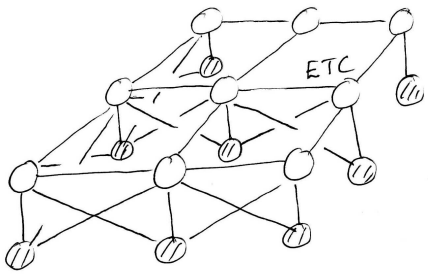


FIGURE 11.2: *The graph of an MRF adapted to image segmentation. The shaded nodes represent the known pixel values (X_i in the text) and the open nodes represent the (unknown, and to be inferred) labels H_i. A particular hidden node may depend on many pixels, because we will use all these pixel values to compute the cost of labelling that node in a particular way.*

Segmentation is another application that fits this recipe. We now want to break the image into a set of regions. Each region will have a label (eg "grass", "sky", "tree", etc.). The X_i are the observed values of each pixel value, and the H_i are the labels. In this case, the graph may have quite complex structure (eg

figure 11.2). We must come up with a process that computes the cost of labelling a given pixel location in the image with a given label. Notice this process could look at many other pixel values in the image to come up with the label, but not at other labels. There are many possibilities. For example, we could build a logistic regression classifier that predicts the label at a pixel from image features around that pixel (if you don't know any image feature constructions, assume we use the pixel color; if you do, you can use anything that pleases you). We then model the cost of a having a particular label at a particular point as the negative log probability of the label under that model. We obtain the $\theta(H_i, H_j)$ by assuming that labels on neighboring pixels should agree with one another, as in the case of denoising.

11.1.6 MAP Inference in Discrete MRF's can be Hard

As you should suspect, focusing on MAP inference doesn't make the difficulty go away for discrete Markov random fields.

Worked example 11.4 *Useful facts about MRF's.*

Show that, using the notation of the text, we have: (a) for any i, $\mathbf{1}^T\mathbf{h}_i = 1$; (b) the MAP inference problem can be expressed as a quadratic program, with linear constraints, on discrete variables.

Solution: For (a) the equation is true because exactly one entry in $\mathbf{h}_i$ is 1, the others are zero. But (b) is more interesting. MAP inference is equivalent to maximizing $\log p(H|X)$. Recall $\log Z$ does not depend on the $\mathbf{h}$. We seek

$$\max_{\mathbf{h}_1,\ldots,\mathbf{h}_N} \left[\left(\sum_{ij} \mathbf{h}_i^T \Theta^{(ij)} \mathbf{h}_j \right) + \sum_i \mathbf{h}_i^T \beta_i \right] + \log Z$$

subject to very important constraints. We must have $\mathbf{1}^T\mathbf{h}_i = 1$ for all i. Furthermore, any component of any $\mathbf{h}_i$ must be either 0 or 1. So we have a quadratic program (because the cost function is quadratic in the variables), with linear constraints, on discrete variables.

Example 11.4 is a bit alarming, because it implies (correctly) that MAP inference in MRF's can be very hard. You should remember this. Gradient descent is no use here because the idea is meaningless. You can't take a gradient with respect to discrete variables. If you have the background, it's quite easy to prove by producing (eg from example 11.4) an MRF where inference is equivalent to max-cut, which is NP hard.

Worked example 11.5 *MAP inference for MRF's is a linear program*

Show that, using the notation of the text, the MAP inference for an MRF problem can be expressed as a linear program, with linear constraints, on discrete variables.

Solution: If you have two binary variables z_i and z_j both in $\{0, 1\}$, then write $q_{ij} = z_i z_j$. We have that $q_{ij} \leq z_i$, $q_{ij} \leq z_j$, $q_{ij} \in \{0, 1\}$, and $q_{ij} \geq z_i + z_j - 1$. You should check (a) these inequalities and (b) that q_{ij} is uniquely identified by these inequalities. Now notice that each $\mathbf{h}_i$ is just a bunch of binary variables, and the quadratic term $\mathbf{h}_i^T \Theta^{(ij)} \mathbf{h}_j$ is linear in q_{ij}.

Example 11.5 is the start of an extremely rich vein of approximation mathematics, which we shall not mine. If you are of a deep mathematical bent, you can phrase everything in what follows in terms of approximate solutions of linear programs. For example, this makes it possible to identify MRF's for which MAP inference can be done in polynomial time; the family is more than just trees. We won't go there.

11.2 VARIATIONAL INFERENCE

We could just ignore intractable models, and stick to tractable models. This isn't a good idea, because intractable models are often quite natural. The discrete Markov random field model of an image is a fairly natural model. Image labels *should* depend on pixel values, and on neighboring labels. It is better to try and deal with the intractable model. One really successful strategy for doing so is to choose a tractable parametric family of probability models $Q(H; \theta)$, then adjust θ to find an element that is "close" in the right sense to $P(H|X)$. This process is known as **variational inference**.

11.2.1 The KL Divergence: Measuring the Closeness of Probability Distributions

Assume we have two probability distributions $P(X)$ and $Q(X)$. A measure of their similarity is the **KL-divergence** (or sometimes **Kullback-Leibler divergence**) written

$$\mathbb{D}(P \parallel Q) = \int P(X) \log \frac{P(X)}{Q(X)} dX$$

(you've clearly got to be careful about zeros in P and Q here). This likely strikes you as an odd measure of similarity, because it isn't symmetric. It is not the case that $\mathbb{D}(P \parallel Q)$ is the same as $\mathbb{D}(Q \parallel P)$, which means you have to watch your P's and Q's. Furthermore, some work will demonstrate that it does not satisfy the triangle inequality, so KL divergence lacks two of the three important properties of a metric.

KL divergence has some nice properties, however. First, we have

$$\mathbb{D}(P \parallel Q) \geq 0$$

with equality only if P and Q are equal almost everywhere (i.e. except on a set of measure zero).

Second, there is a suggestive relationship between KL divergence and maximum likelihood. Assume that X_i are IID samples from some *unknown* $P(X)$, and we wish to fit a parametric model $Q(X|\theta)$ to these samples. This is the usual situation we deal with when we fit a model. Now write $H(P)$ for the entropy of $P(X)$, defined by

$$H(P) = -\int P(X) \log P(X) dx = -\mathbb{E}_P[\log P].$$

The distribution P is unknown, and so is its entropy, but it is a constant. Now we can write

$$\mathbb{D}(P \parallel Q) = \mathbb{E}_P[\log P] - \mathbb{E}_P[\log Q]$$

Then

$$\begin{aligned}\mathcal{L}(\theta) = \sum_i \log Q(X_i|\theta) \approx \int P(X) \log Q(X|\theta) dX &= \mathbb{E}_{P(X)}[\log Q(X|\theta)] \\ &= -H(P) - \mathbb{D}(P \parallel Q)(\theta).\end{aligned}$$

Equivalently, we can write

$$\mathcal{L}(\theta) + \mathbb{D}(P \parallel Q)(\theta) = -H(P).$$

Recall P doesn't change (though it's unknown), so $H(P)$ is also constant (though unknown). This means that when $\mathcal{L}(\theta)$ goes up, $\mathbb{D}(P \parallel Q)(\theta)$ must go down. When $\mathcal{L}(\theta)$ is at a maximum, $\mathbb{D}(P \parallel Q)(\theta)$ must be at a minimum. All this means that, when you choose θ to maximize the likelihood of some dataset given θ for a parametric family of models, you are choosing the model in that family with smallest KL divergence from the (unknown) $P(X)$.

11.2.2 The Variational Free Energy

We have a $P(H|X)$ that is hard to work with (usually because we can't evaluate $P(X)$) and we want to obtain a $Q(H)$ that is "close to" $P(H|X)$. A good choice of "close to" is to require that

$$\mathbb{D}(Q(H) \parallel P(H|X))$$

is small. Expand the expression for KL divergence, to get

$$\begin{aligned}\mathbb{D}(Q(H) \parallel P(H|X)) &= \mathbb{E}_Q[\log Q] - \mathbb{E}_Q[\log P(H|X)] \\ &= \mathbb{E}_Q[\log Q] - \mathbb{E}_Q[\log P(H,X)] + \mathbb{E}_Q[\log P(X)] \\ &= \mathbb{E}_Q[\log Q] - \mathbb{E}_Q[\log P(H,X)] + \log P(X)\end{aligned}$$

which at first glance may look unpromising, because we can't evaluate $P(X)$. But $\log P(X)$ is fixed (although unknown). Now rearrange to get

$$\begin{aligned}\log P(X) &= \mathbb{D}(Q(H) \parallel P(H|X)) - (\mathbb{E}_Q[\log Q] - \mathbb{E}_Q[\log P(H,X)]) \\ &= \mathbb{D}(Q(H) \parallel P(H|X)) - \mathsf{F}_Q.\end{aligned}$$

Here

$$\mathsf{F}_Q = (\mathbb{E}_Q[\log Q] - \mathbb{E}_Q[\log P(H, X)])$$

is referred to as the **variational free energy**. We can't evaluate $\mathbb{D}(Q(H) \| P(H|X))$. But, because $\log P(X)$ is fixed, when F_Q goes down, $\mathbb{D}(Q(H) \| P(H|X))$ must go down too. Furthermore, a minimum of F_Q will correspond to a minimum of $\mathbb{D}(Q(H) \| P(H|X))$. And we can evaluate F_Q.

We now have a strategy for building approximate $Q(H)$. We choose a family of approximating distributions. From that family, we obtain the $Q(H)$ that minimises F_Q (which will take some work). The result is the $Q(H)$ in the family that minimizes $\mathbb{D}(Q(H) \| P(H|X))$. We use that $Q(H)$ as our approximation to $P(H|X)$, and extract whatever information we want from $Q(H)$.

11.3 EXAMPLE: VARIATIONAL INFERENCE FOR BOLTZMANN MACHINES

We want to construct a $Q(H)$ that approximates the posterior for a Boltzmann machine. We will choose $Q(H)$ to have one factor for each hidden variable, so $Q(H) = q_1(H_1)q_2(H_2)\ldots q_N(H_N)$. We will then assume that all but one of the terms in Q are known, and adjust the remaining term. We will sweep through the terms doing this until nothing changes.

The i'th factor in Q is a probability distribution over the two possible values of H_i, which are 1 and -1. There is only one possible choice of distribution. Each q_i has one parameter $\pi_i = P(\{H_i = 1\})$. We have

$$q_i(H_i) = (\pi_i)^{\frac{(1+H_i)}{2}} (1 - \pi_i)^{\frac{(1-H_i)}{2}}.$$

Notice the trick; the power each term is raised to is either 1 or 0, and I have used this trick as a switch to turn on or off each term, depending on whether H_i is 1 or -1. So $q_i(1) = \pi_i$ and $q_i(-1) = (1 - \pi_i)$. This is a standard, and quite useful, trick. We wish to minimize the variational free energy, which is

$$\mathsf{F}_Q = (\mathbb{E}_Q[\log Q] - \mathbb{E}_Q[\log P(H, X)]).$$

We look at the $\mathbb{E}_Q[\log Q]$ term first. We have

$$\begin{aligned} \mathbb{E}_Q[\log Q] &= \mathbb{E}_{q_1(H_1)\ldots q_N(H_N)}[\log q_1(H_1) + \ldots \log q_N(H_N)] \\ &= \mathbb{E}_{q_1(H_1)}[\log q_1(H_1)] + \ldots \mathbb{E}_{q_N(H_N)}[\log q_N(H_N)] \end{aligned}$$

where we get the second step by noticing that

$$\mathbb{E}_{q_1(H_1)\ldots q_N(H_N)}[\log q_1(H_1)] = \mathbb{E}_{q_1(H_1)}[\log q_1(H_1)]$$

(write out the expectations and check this if you're uncertain).

Now we need to deal with $\mathbb{E}_Q[\log P(H|X)]$. We have

$$\begin{aligned} \log p(H, X) &= -E(H, X) - \log Z \\ &= \sum_{i \in H} \sum_{j \in \mathcal{N}(i) \cap H} \theta_{ij} H_i H_j + \sum_{i \in H} \sum_{j \in \mathcal{N}(i) \cap X} \theta_{ij} H_i X_j + K \end{aligned}$$

(where K doesn't depend on any H and is so of no interest). Assume all the q's are known except the i'th term. Write $Q_{\hat{i}}$ for the distribution obtained by omitting q_i from the product, so $Q_{\hat{1}} = q_2(H_2)q_3(H_3)\dots q_N(H_N)$, etc. Notice that

$$\mathbb{E}_Q[\log P(H|X)] = \begin{pmatrix} q_i(-1)\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1,\dots,H_i=-1,\dots,H_N|X)] + \\ q_i(1)\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1,\dots,H_i=1,\dots,H_N|X)] \end{pmatrix}.$$

This means that if we fix all the q terms *except* $q_i(H_i)$, we must choose q_i to minimize

$$\begin{array}{rl} q_i(-1)\log q_i(-1) + q_i(1)\log q_i(1) & - \\ q_i(-1)\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1,\dots,H_i=-1,\dots,H_N|X)] & + \\ q_i(1)\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1,\dots,H_i=1,\dots,H_N|X)] & \end{array}$$

subject to the constraint that $q_i(1) + q_i(-1) = 1$. Introduce a Lagrange multiplier to deal with the constraint, differentiate and set to zero, and get

$$\begin{aligned} q_i(1) &= \frac{1}{c}\exp\left(\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1,\dots,H_i=1,\dots,H_N|X)]\right) \\ q_i(-1) &= \frac{1}{c}\exp\left(\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1,\dots,H_i=-1,\dots,H_N|X)]\right) \\ \text{where } c &= \exp\left(\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1,\dots,H_i=-1,\dots,H_N|X)]\right) + \\ &\quad \exp\left(\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1,\dots,H_i=1,\dots,H_N|X)]\right). \end{aligned}$$

In turn, this means we need to know $\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1,\dots,H_i=-1,\dots,H_N|X)]$, etc. only up to a constant. Equivalently, we need to compute only $\log q_i(H_i) + K$ for K some unknown constant (because $q_i(1) + q_i(-1) = 1$). Now we compute

$$\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1,\dots,H_i=-1,\dots,H_N|X)].$$

This is equal to

$$\mathbb{E}_{Q_{\hat{i}}}\left[\sum_{j\in\mathcal{N}(i)\cap H}\theta_{ij}(-1)H_j + \sum_{j\in\mathcal{N}(i)\cap X}\theta_{ij}(-1)X_j + \text{terms not containing } H_i\right]$$

which is the same as

$$\sum_{j\in\mathcal{N}(i)\cap H}\theta_{ij}(-1)\mathbb{E}_{Q_{\hat{i}}}[H_j] + \sum_{j\in\mathcal{N}(i)\cap X}\theta_{ij}(-1)X_j + K$$

and this is the same as

$$\sum_{j\in\mathcal{N}(i)\cap H}\theta_{ij}(-1)((\pi_j)(1) + (1-\pi_j)(-1)) + \sum_{j\in\mathcal{N}(i)\cap X}\theta_{ij}(-1)X_j + K$$

and this is

$$\sum_{j\in\mathcal{N}(i)\cap H}\theta_{ij}(-1)(2\pi_j - 1) + \sum_{j\in\mathcal{N}(i)\cap X}\theta_{ij}(-1)X_j + K.$$

If you thrash through the case for

$$\mathbb{E}_{Q_i}[\log P(H_1, \ldots, H_i = 1, \ldots, H_N|X)]$$

(which works the same) you will get

$$\begin{aligned}
\log q_i(1) &= \mathbb{E}_{Q_i}[\log P(H_1, \ldots, H_i = 1, \ldots, H_N, X)] + K \\
&= \sum_{j \in \mathcal{N}(i) \cap H} [\theta_{ij}(2\pi_j - 1)] + \sum_{j \in \mathcal{N}(i) \cap X} [\theta_{ij} X_j] + K \\
\text{and} & \\
\log q_i(-1) &= \mathbb{E}_{Q_i}[\log P(H_1, \ldots, H_i = -1, \ldots, H_N, X)] + K \\
&= \sum_{j \in \mathcal{N}(i) \cap H} [-\theta_{ij}(2\pi_j - 1)] + \sum_{j \in \mathcal{N}(i) \cap X} [-\theta_{ij} X_j] + K
\end{aligned}$$

All this means that

$$\pi_i = \frac{e^{\left(\sum_{j \in \mathcal{N}(i) \cap H} [\theta_{ij}(2\pi_j - 1)] + \sum_{j \in \mathcal{N}(i) \cap X} [\theta_{ij} X_j]\right)}}{e^{\left(\sum_{j \in \mathcal{N}(i) \cap H} [\theta_{ij}(2\pi_j - 1)] + \sum_{j \in \mathcal{N}(i) \cap X} [\theta_{ij} X_j]\right)} + e^{\left(\sum_{j \in \mathcal{N}(i) \cap H} [-\theta_{ij}(2\pi_j - 1)] + \sum_{j \in \mathcal{N}(i) \cap X} [-\theta_{ij} X_j]\right)}}.$$

After this blizzard of calculation, our inference algorithm is straightforward. We visit each hidden node in turn, set the associated π_i to the value of the expression above *assuming all the other π_j are fixed at their current values*, and repeat until convergence. We can test convergence by evaluating the variational free energy; an alternative is to check the size of the change in each π_j.

We can now do anything to $Q(H)$ that we would have done to $P(H|X)$. For example, we might compute the values of H that maximize $Q(H)$ for MAP inference. It is wise to limit ones ambition here, because $Q(H)$ is an approximation. It's straightforward to set up and describe, but it isn't particularly good. The main problem is that the variational distribution is unimodal. Furthermore, we chose a variational distribution by assuming that each H_i was independent of all others. This means that computing, say, covariances will likely lead to the wrong numbers (although it's easy — almost all are zero, and the remainder are easy). Obtaining an approximation by assuming that H_i is independent of all others is often called a **mean field method**.

CHAPTER 12

Classification with Neural Networks

12.1 UNITS AND CLASSIFICATION

We will build complex classification systems out of simple units. A **unit** takes a vector $\mathbf{x}$ of inputs and uses a vector $\mathbf{w}$ of parameters (known as the **weights**), a scalar b (known as the **bias**), and a nonlinear function F to form its output, which is

$$F(\mathbf{w}^T\mathbf{x} + b).$$

Over the years, a wide variety of nonlinear functions have been tried. Current best practice is to use the **RELU** (for rectified linear unit), where

$$F(u) = \mathsf{max}(0, u).$$

For example, if $\mathbf{x}$ was a point on the plane, then a single unit would represent a line, chosen by the choice of $\mathbf{w}$ and b. The output for all points on one side of the line would be zero. The output for points on the other side would be a positive number that is larger for points that are further from the line.

Units are sometimes referred to as **neurons**, and there is a large and rather misty body of vague speculative analogy linking devices built out of units to neuroscience. I deprecate this practice; what we are doing here is quite useful and interesting enough to stand on its own without invoking biological authority. Also, if you want to see a real neuroscientist laugh, explain to them how your neural network is really based on some gobbet of brain tissue or other.

12.1.1 Building a Classifier out of Units: The Cost Function

We will build a multiclass classifier out of units by modelling the class posterior probabilities using the outputs of the units. Each class will get the output of a single unit. Write o_i for the output of the i'th unit, and θ for all the parameters in all the units. We will organize these units into a vector $\mathbf{o}$, whose i'th component is o_i. We want to use that unit to model the probability that the input is of class j, which I will write $p(\text{class} = j|\mathbf{x}, \theta)$. To build this model, I will use the **softmax function**. This is a function that takes a C dimensional vector and returns a C dimensional vector. I will write $\mathbf{s}(\mathbf{u})$ for the softmax function, and the dimension C will always be the number of classes. We have

$$\mathbf{s}(\mathbf{u}) = \left(\frac{1}{\sum_k e^{u_k}}\right) \begin{bmatrix} e^{u_1} \\ e^{u_2} \\ \dots \\ e^{u_C} \end{bmatrix}$$

(recall u_i is the i'th component of $\mathbf{u}$). We then use the model

$$p(\text{class} = i|\mathbf{x}, \theta) = s_i(\mathbf{o}(\mathbf{x}, \theta)).$$

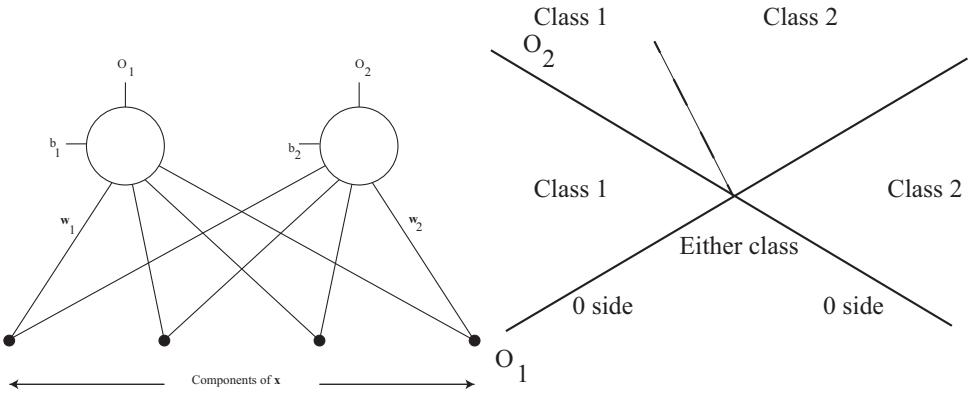


FIGURE 12.1: *On the* **left**, *two units observing an input vector, and providing outputs. On the* **right**, *the decision boundary for two units classifying a point on the plane into one of two classes. The angle of the dashed line depends on the magnitudes of* $\mathbf{w}_1$ *and* $\mathbf{w}_2$.

Notice that this expression passes important tests for a probability model. Each value is between 0 and 1, and the sum over classes is 1.

In this form, the classifier is not super interesting. For example, imagine that the features $\mathbf{x}$ are points on the plane, and we have two classes. Then we have two units, one for each class. There is a line corresponding to each unit; on one side of the line, the unit produces a zero, and on the other side, the unit produces a positive number that increases as with perpendicular distance from the line. We can get a sense of what the decision boundary will be like from this. When a point is on the 0 side of both lines, the class probabilities will be equal (and so both $\frac{1}{2}$ – two classes, remember). When a point is on the positive side of the i'th line, but the zero side of the other, the class probability for class i will be

$$\frac{e^{o_i(\mathbf{x},\theta)}}{1 + e^{o_i(\mathbf{x},\theta)}},$$

and the point will always be classified in the i'th class (remember, $o_i \geq 0$). Finally, when a point is on the positive side of both lines, the classifier boils down to choosing the i that has the largest value of $o_i(\mathbf{x}, \theta)$. All this leads to the decision boundary shown in figure **??**. Notice that this is piecewise linear, and somewhat more complex than the boundary of an SVM. It's quite helpful to try and draw what would happen for three or more classes with $\mathbf{x}$ a 2D point.

12.1.2 Building a Classifier out of Units: Strategy

The essential difficulty here is to choose θ that results in the best behavior. We will do so by writing a cost function that estimates the error rate of the classification, then choosing a value $\hat{\theta}$ that minimises that function. We have a set of N examples $\mathbf{x}_i$ and for each example we know the class. There are a total of C classes. We encode the class of an example using a **one hot** vector $\mathbf{y}_i$, which is C dimensional.

If the i'th example is from class j, then the j'th component of $\mathbf{y}_i$ is 1, and all other components in the vector are 0. I will write y_{ij} for the j'th component of $\mathbf{y}_i$.

A natural cost function looks at the log likelihood of the data under the probability model produced from the outputs of the units. If the i'th example is from class j, we would like $-\log p(\text{class} = j|\mathbf{x}_i, \theta)$ to be small (notice the sign here; it's usual to minimize negative log likelihood). I will write $\log \mathbf{s}$ to mean the vector whose components are the logarithms of the components of $\mathbf{s}$. This yields a loss function

$$\frac{1}{N} \sum_{i \in \text{data}} \left[\sum_{j \in \text{classes}} \left\{ -\mathbf{y}_i^T \log \mathbf{s}(\mathbf{o}(\mathbf{x}_i, \theta)) \right\} \right].$$

Notice that this loss function is written in a clean way that may lead to a poor implementation. I have used the y_{ij} values as "switches", as in the discussion of EM. This leads to clean notation, but hides fairly obvious computational efficiencies (when taking the gradient, you need to deal with only one term in the sum over classes). As in the case of the linear SVM (section 10.1), we would like to achieve a low cost with a "small" θ, and so form an overall cost function that will have loss and penalty terms.

There are a variety of possible penalties. For now, we will penalize large sets of weights, but we'll look at other possibilities below. Remember, we have C units (one per class) and so there are C distinct sets of weights. Write the weights for the u'th unit $\mathbf{w}_u$. Our penalty becomes

$$\sum_{u \in \text{units}} \mathbf{w}_u^T \mathbf{w}_u.$$

As in the case of the linear SVM (section 10.1), we write λ for a weight applied to the penalty. Our cost function is then

$$S(\theta, \mathbf{x}; \lambda) = \underbrace{\frac{1}{N} \sum_{i \in \text{data}} \left[\sum_{j \in \text{classes}} \left\{ -\mathbf{y}_i^T \log \mathbf{s}(\mathbf{o}(\mathbf{x}_i, \theta)) \right\} \right]}_{\text{(misclassification loss)}} + \underbrace{\frac{\lambda}{2} \sum_{u \in \text{units}} \mathbf{w}_u^T \mathbf{w}_u}_{\text{(penalty)}}$$

12.1.3 Building a Classifier out of Units: Training

I have described a simple classifier built out of units. We must now train this classifier, by choosing a value of θ that results in a small loss. It may be quite hard to get the true minimum, and we may need to settle for a small value. We use stochastic gradient descent, because we have seen it before; because it is effective; and because it is the algorithm of choice when training more complex classifiers built out of units.

For the SVM, we selected one example at random, computed the gradient at that example, updated the parameters, and went again. For neural nets, it is more usual to use **minibatch training**, where we select a subset of the data uniformly and at random, compute a gradient using that subset, update and go again. This is because in the best implementations many operations are vectorized, and using a minibatch can provide a gradient estimate that is clearly better than

that obtained using only one example, but doesn't take longer to compute. The size of the minibatch is usually determined by memory or architectural considerations. It is often a power of two, for this reason.

Now imagine we have chosen a minibatch of M examples. We must compute the gradient of the cost function. This is mainly an exercise in notation, but there's a lot of notation. Write θ_u for a vector containing all the parameters for the u'th unit, so that $\theta_u = [\mathbf{w}_u, b_u]^T$. Recall $s_k(\mathbf{o}(\mathbf{x}_i, \theta_k))$ is the output of the softmax function for the k'th unit for input $\mathbf{x}_i$. This represents the probability that example i is of class k under the current model. Then we must compute

$$\nabla_{\theta_u} \frac{1}{M} \sum_{i \in \text{minibatch}} \left[\left\{ -\mathbf{y}_i^T \log \mathbf{s}(\mathbf{o}(\mathbf{x}_i, \theta)) \right\} \right] + \frac{\lambda}{2} \sum_{j \in \text{classes}} \mathbf{w}_j^T \mathbf{w}_j.$$

The gradient is easily computed using the chain rule. The term

$$\frac{\lambda}{2} \sum_{j \in \text{classes}} \mathbf{w}_j^T \mathbf{w}_j$$

presents no challenge, but the other term is more interesting. We must differentiate the softmax function by its inputs, then the units by their parameters. More notation: assume we have a vector valued function of vector inputs, for example, $\mathbf{s}(\mathbf{o})$. Here $\mathbf{s}$ is the function and $\mathbf{o}$ are the inputs. I will write $\#(\mathbf{o})$ to mean the number of components of $\mathbf{o}$, and o_i for the i'th component. The matrix of first partial derivatives is extremely important (we will see a lot of these; pay attention). I will write $\mathcal{J}_{\mathbf{s};\mathbf{o}}$ to mean

$$\begin{pmatrix} \frac{\partial s_1}{\partial o_1} & \cdots & \frac{\partial s_1}{\partial o_{\#(\mathbf{o})}} \\ \cdots & \cdots & \cdots \\ \frac{\partial s_{\#(\mathbf{s})}}{\partial o_1} & \cdots & \frac{\partial s_{\#(\mathbf{s})}}{\partial o_{\#(\mathbf{o})}} \end{pmatrix}$$

and refer to such a matrix of first partial derivatives as a **Jacobian**.

Now we can use the chain rule to write

$$\nabla_{\theta_u} \frac{1}{M} \sum_{i \in \text{minibatch}} \left[\left\{ -\mathbf{y}_i^T \log \mathbf{s}(\mathbf{x}_i, \theta) \right\} \right] = \frac{1}{M} \sum_{i \in \text{minibatch}} \left[\left\{ -\mathbf{y}_i^T \mathcal{J}_{\log \mathbf{s};\mathbf{o}} \mathcal{J}_{\mathbf{o};\theta_u} \right\} \right].$$

This isn't particularly helpful without knowing the relevant Jacobians. They're quite straightforward.

Write $\mathbb{I}_{[u=v]}(u, v)$ for the indicator function that is 1 when $u = v$ and zero otherwise. We have

$$\begin{aligned} \frac{\partial \log s_u}{\partial o_v} &= \mathbb{I}_{[u=v]} - \frac{e^{o_v}}{\sum_k e^{o_k}} \\ &= \mathbb{I}_{[u=v]} - s_v. \end{aligned}$$

To get the other Jacobian, we need yet more notation (but this isn't new, it's a reminder). I will write $w_{u,i}$ for the i'th component of $\mathbf{w}_u$, and $\mathbb{I}_{[o_u>0]}(o_u)$ for the indicator function that is 1 if its argument is greater than zero. Then

$$\frac{\partial o_u}{\partial w_{u,i}} = x_i \mathbb{I}_{[o_u>0]}(o_u)$$

and

$$\frac{\partial o_u}{\partial b_u} = \mathbb{I}_{[o_u > 0]}(o_u).$$

Notice that if $v \neq u$,

$$\frac{\partial o_u}{\partial w_{v,i}} = 0 \text{ and } \frac{\partial o_u}{\partial b_v} = 0.$$

At least in principle, we can build a multiclass classifier in a straightforward way using minibatch gradient descent. We use one unit per class, each one using each component of the feature vector. We obtain training data, and then iterate computing a gradient from a minibatch, and taking a step along the negative of the gradient. If you try, you may run into some of the important small practical problems that cause networks to work badly. Here are some of the ones you may encounter.

Initialization: You need to choose the initial values of all of the parameters. There are many parameters; in our case, with a d dimensional $\mathbf{x}$ and C classes, we have $(d+1) \times C$ parameters. If you initialize each parameter to zero, you will find that the gradient is also zero, which is not helpful. This occurs because all the o_u will be zero (because the $w_{u,i}$ and the b_u are zero). It is usual to initialize to draw a sample of a zero mean normal random variable for each initial value (appropriate choices of variance get interesting; more below).

Learning rate: Each step will look like $\theta^{(n+1)} = \theta^{(n)} - \eta_n \nabla_\theta \text{cost}$. You need to choose η_n for each step. This is widely known as the **learning rate**; an older term is **steplength** (neither term is a super-accurate description). It is not usual for the learning rate to be the same throughout learning. We would like to take "large" steps early, and "small" steps late, in learning, so we would like η_n to be "large" for small n, and "small" for large n. It is tough to be precise about a good choice. As in stochastic gradient descent for a linear SVM, breaking learning into epochs ($e(n)$ is the epoch of the n'th iteration), then choosing two constants a and b to obtain

$$\eta_n = \frac{1}{a + be(n)}$$

is quite a good choice. The constants, and the epoch size, will need to be chosen by experiment. As we build more complex collections of units, the need for a better process will become pressing; two options appear below.

Ensuring learning is proceeding: We need to keep track of what is going on inside the system as we train it. One way is to plot the loss as a function of the number of steps. These plots can be very informative (Figure **??**). If the learning rate is small, the system will make very slow progress but may (eventually) end up in a good state. If the learning rate is large, the system will make fast progress initially, but will then stop improving, because the state will change too quickly to find a good solution. If the learning rate is very large, the system might even diverge. If the learning rate is just right, you should get fast descent to a good value, and then slow but fairly steady improvement. Of course, just as in the case of SVMs, the plot of loss against step isn't a smooth curve, but rather noisy. There is an amusing collection of examples of training problems at lossfunctions.tumblr.com It is quite usual to plot the error rate, or the accuracy, on a validation dataset

while training. This will allow you to compare the training error with the validation error. If these are very different, you have a problem: the system is overfitting, or not generalizing well. You should increase the regularization constant.

Dead units: Imagine the system gets into a state where for some unit u, $o_u = 0$ for every training data item. This could happen, for example, if the learning rate was too large. Then it can't get out of this state, because the gradient for that unit will be zero for every training data item, too. Such units are referred to as **dead units**. This problem can be contained by keeping the learning rate small enough. In more complex architectures (below), it is also contained by having a large number of units.

Gradient problems: There are a variety of important ways to have gradient problems. By far the most important is making a simple error in code (i.e you compute the Jacobian elements wrong). This is surprisingly common; everybody does it at least once; and one learns to check gradients. Checking is fairly straightforward. You compute a numerical derivative, and compare that to the exact derivative. If they're too different, you have a gradient problem you need to fix. We will see a second important gradient problem when we see more complex architectures.

Choosing the regularization constant: This follows the recipe we saw for a linear SVM. Hold out a validation dataset. Train for several different values of λ. Evaluate each system on the validation dataset, and choose the best. Notice this involves many rounds of training, which could make things slow.

Does it work? Evaluating the classifier we have described is like evaluating any other classifier. You evaluate the error on a held-out data set that *wasn't* used to choose the regularization constant, or during training.

12.2 LAYERS AND NETWORKS

We have built a multiclass classifier out of units by using one unit per class, then interpreting the outputs of the units as probabilities using a softmax function. This classifier is at best only mildly interesting. The way to get something really interesting is to ask what the features for this classifier should be. To date, we have not looked closely at features. Instead, we've assumed they "come with the dataset" or should be constructed from domain knowledge. Remember that, in the case of regression, we could improve predictions by forming non-linear functions of features. We can do better than that; we could *learn* what non-linear functions to apply, by using the output of one set of units to form the inputs of the next set.

We will focus on systems built by organizing the units into **layers**; these layers form a **neural network** (a term I dislike, for the reasons above, but use because everybody else does). There is an input layer, consisting of the units that receive feature inputs from outside the network. There is an output layer, consisting of units whose outputs are passed outside the network. These two might be the same, as they were in the previous section. The most interesting cases occur when they are not the same. There may be **hidden layers**, whose inputs come from other layers and whose outputs go to other layers. In our case, the layers are ordered, and outputs of a given layer act as inputs to the next layer only (as in Figure 12.2 - we don't allow connections to wander all over the network). For the moment, assume that each unit in a layer receives an input from every unit in the previous layer;

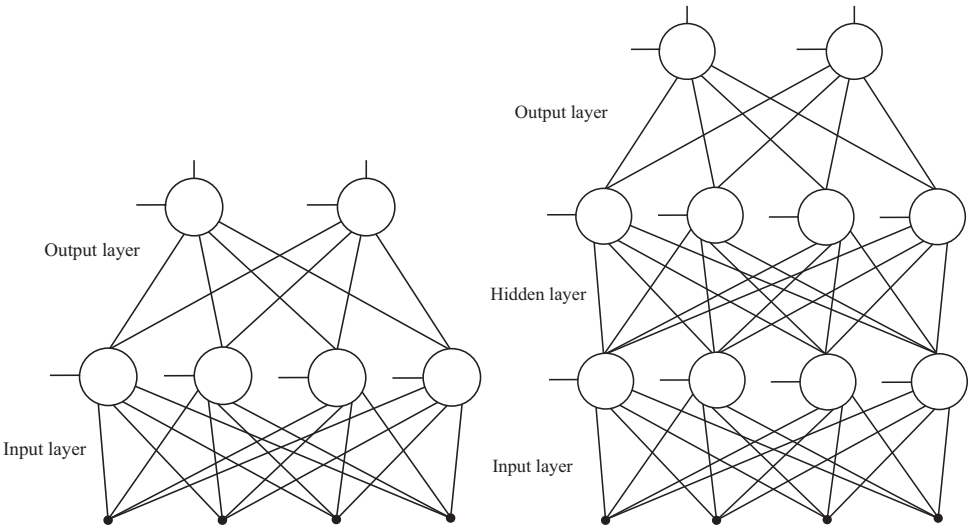


FIGURE 12.2: *On the* **left**, *an input layer connected to an output layer. The units in the input layer take the inputs and compute features; the output layer turns these features into output values that will be turned into class probabilities with the softmax function. On the* **right**, *there is a hidden layer between input and output layer. This architecture means that the features seen by the output layer can be trained to be a significantly more complex function of the inputs.*

this means that our network is **fully connected**. Other architectures are possible, but right now the most important question is how to train the resulting object.

12.2.1 Notation

Inevitably, we need yet more notation. There will be L layers. The input layer is layer 1, and the output layer is L. I will write u_i^l for the i'th unit in the l'th layer. This unit has output o_i^l and parameters $\mathbf{w}_i^l$ and b_i^l, which I will stack into a vector θ_i^l. I write θ^l to refer to all the parameters of layer l. If I do not need to identify the layer in which a unit sits (for example, if I am summing over all units) I will drop the superscript. The vector of inputs to this unit is $\mathbf{x}_i^l$. These inputs are formed by choosing from the outputs of layer $l-1$. I will write $\mathbf{o}^l$ for all the outputs of the l'th layer, stacked into a vector. I will represent the connections by a matrix $\mathcal{C}_i^l$, so that $\mathbf{x}_i^l = \mathcal{C}_i^l \mathbf{o}^{l-1}$. The matrix $\mathcal{C}_i^l$ contains only 1 or 0 entries, and in the case of fully connected layers, it is the identity. Notice that every unit has its own $\mathcal{C}_i^l$.

I will write $L(\mathbf{y}_i, \mathbf{s}(\mathbf{o}^L(\mathbf{x}_i, \theta)))$ for the loss of classifying the i'th example using softmax. We will continue to use

$$L(\mathbf{y}_i, \mathbf{s}(\mathbf{o}^L(\mathbf{x}_i, \theta))) = -\mathbf{y}_i^T \log \mathbf{s}(\mathbf{o}^L(\mathbf{x}_i, \theta)))$$

but in other applications, other losses might arise.

Generally, we will train by mini batch gradient descent, though I will describe some tricks that can speed up training and improve results. But we must compute

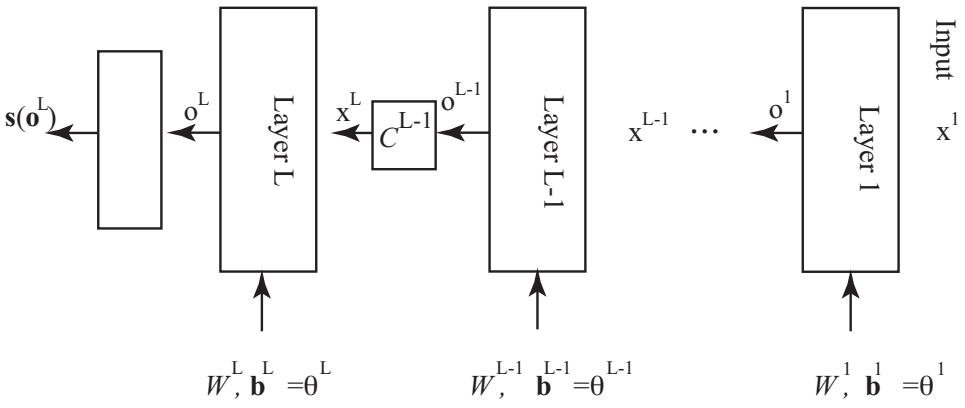


FIGURE 12.3: *Notation for layers, inputs, etc.*

the gradient. The output layer of our network has C units, one per class. We will apply the softmax to these outputs, as before. Writing E for the cost of error on training examples and R for the regularization term, we can write the cost of using the network as

$$\text{cost} = E + R = (1/N) \sum_{i \in \text{examples}} L(\mathbf{y}_i, \mathbf{s}(\mathbf{o}^L(\mathbf{x}_i, \theta))) + \frac{\lambda}{2} \sum_{k \in \text{units}} \mathbf{w}_k^T \mathbf{w}_k.$$

You should not let this compactified notation let you lose track of the fact that $\mathbf{o}^L$ depends on $\mathbf{x}_i$ through $\mathbf{o}^{L-1}, \ldots, \mathbf{o}^1$. What we really should write is

$$\mathbf{o}^L(\mathbf{o}^{L-1}(\ldots(\mathbf{o}^1(\mathbf{x}, \theta^1), \theta^2), \ldots), \theta^L).$$

Equivalently, we could stack all the $\mathcal{C}_i^l$ into one linear operator $\mathcal{C}^l$ and write

$$\begin{aligned} \mathbf{o}^L(\mathbf{x}^L, \theta^L) & \quad \text{where} \\ \mathbf{x}^L &= \mathcal{C}^L \mathbf{o}^{L-1}(\mathbf{x}^{L-1}, \theta^{L-1}) \\ \ldots &= \ldots \\ \mathbf{x}^2 &= \mathcal{C}^2 \mathbf{o}^1(\mathbf{x}^1, \theta^1) \\ \mathbf{x}^1 = \mathcal{C}^1 \mathbf{x} & \end{aligned}$$

This is important, because it allows us to write an expression for the gradient.

12.2.2 Training, Gradients and Backpropagation

Now consider $\nabla_\theta E$. We have that E is a sum over examples. The gradient of the loss at a particular example is of most interest, because we will usually train with minibatches. So we are interested in

$$\nabla_\theta E_i = \nabla_\theta L(\mathbf{y}_i, \mathbf{s}(\mathbf{o}^L(\mathbf{x}_i, \theta))) = \nabla_\theta \left[-\mathbf{y}_i^T \log \mathbf{s}(\mathbf{o}^L(\mathbf{x}_i, \theta))) \right]$$

and we can extend our use of the chain rule from section 12.1.3, very aggressively. We have

$$\nabla_{\theta^L} L(\mathbf{y}_i, \mathbf{s}(\mathbf{o}^L(\mathbf{x}_i, \theta))) = -\mathbf{y}_i^T \mathcal{J}_{\log \mathbf{s}; \mathbf{o}^L} J_{\mathbf{o}^L; \theta^L}$$

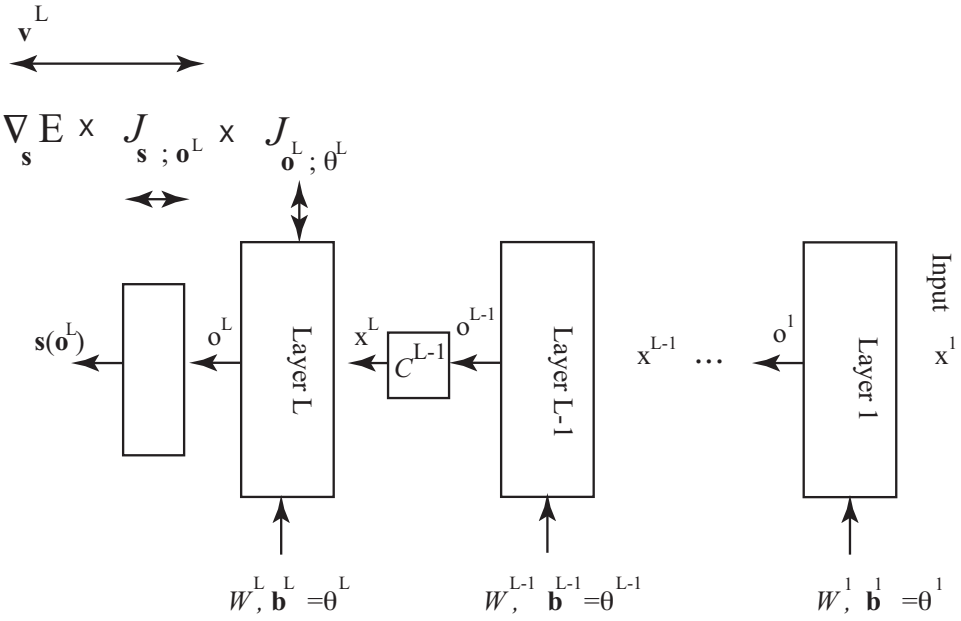


FIGURE 12.4: *Constructing the gradient with respect to θ^L.*

as in that section. Differentiating with respect to θ^{L-1} is more interesting. Layer L depends on θ^{L-1} in a somewhat roundabout way; layer $L-1$ uses θ^{L-1} to produce its outputs, and these are fed into layer L *as its inputs*. So we must have

$$\nabla_{\theta^{L-1}} E_i = -\mathbf{y}_i^T \mathcal{J}_{\log \mathbf{s};\mathbf{o}^L} J_{\mathbf{o}^L;\mathbf{x}^L} J_{\mathbf{x}^L;\theta^{L-1}}$$

(look carefully at the subscripts on the Jacobians). These Jacobians have about the same form as those in section 12.1.3 if you recall that $\mathbf{x}^L = \mathcal{C}^L \mathbf{o}^{L-1}$. In turn, this means that

$$J_{\mathbf{x}^L;\theta^{L-1}} = \mathcal{C}^L J_{\mathbf{o}^{L-1};\theta^{L-1}}$$

and the form of that Jacobian appears in section 12.1.3. But $\mathbf{o}^L$ depends on θ^{L-2} through *its* inputs (which are $\mathbf{x}^{L-1}$), so that

$$\nabla_{\theta^{L-2}} E_i = -\mathbf{y}_i^T \mathcal{J}_{\log \mathbf{s};\mathbf{o}^L} J_{\mathbf{o}^L;\mathbf{x}^L} J_{\mathbf{o}^{L-1};\mathbf{x}^{L-1}} J_{\mathbf{x}^{L-1};\theta^{L-2}}$$

(again, look carefully at the subscripts on each of the Jacobians).

We can now get to the point. We have a recursion, which can be made more

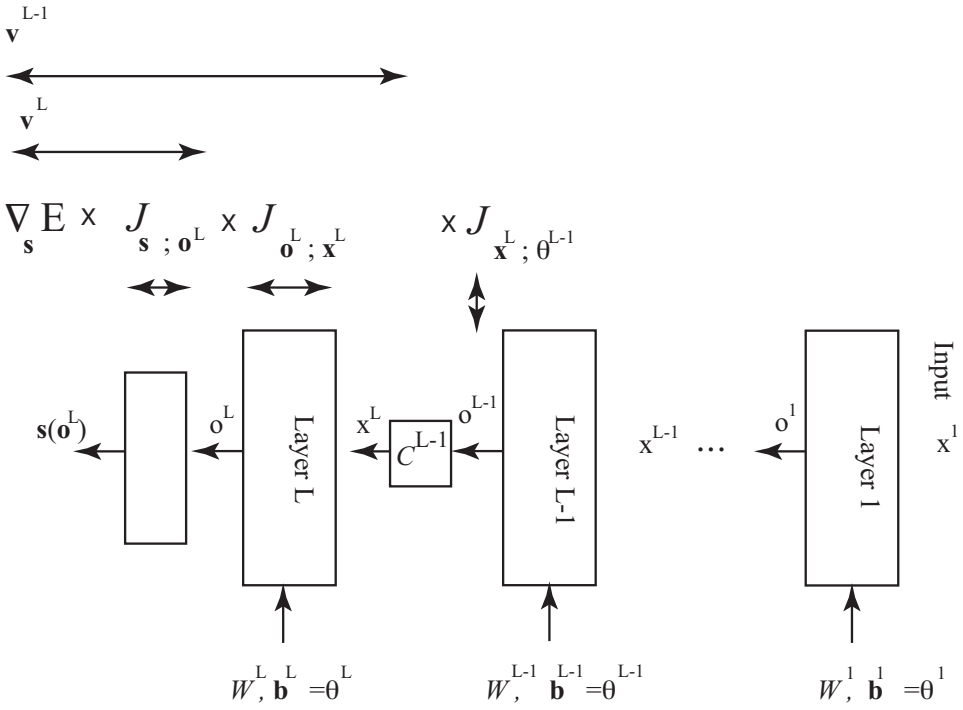


FIGURE 12.5: *Constructing the gradient with respect to θ^{L-1}.*

obvious with some notation. We have

$$\begin{aligned}
\mathbf{v}^L &= \left(\nabla^L_{\mathbf{o}} E_i\right) \\
\nabla_{\theta^L} E_i &= \mathbf{v}^L \mathcal{J}_{\mathbf{o}^L;\theta^L} \\
\mathbf{v}^{L-1} &= \mathbf{v}^L \mathcal{J}_{\mathbf{o}^L;\mathbf{x}^L} \\
\nabla_{\theta^{L-1}} E &= \mathbf{v}^{L-1} \mathcal{J}_{\mathbf{x}^L;\theta^{L-1}} \\
&\cdots \\
\mathbf{v}^{i-1} &= \mathbf{v}^i \mathcal{J}_{\mathbf{x}^{i+1};\mathbf{x}^i} \\
\nabla_{\theta^{i-1}} E &= \mathbf{v}^{i-1} \mathcal{J}_{\mathbf{x}^i;\theta^{i-1}} \\
&\cdots
\end{aligned}$$

I have not added notation to keep track of the point at which the partial derivative is evaluated (it should be obvious, and we have quite enough notation already). When you look at this recursion, you should see that, to evaluate $\mathbf{v}^{i-1}$, you will need to know $\mathbf{x}^k$ for $k \geq i-1$. This suggests the following strategy. We compute the $\mathbf{x}$'s (and, equivalently, $\mathbf{o}$'s) with a "forward pass", moving from the input layer to the output layer. Then, in a "backward pass" from the output to the input, we compute the gradient. Doing this is often referred to as **backpropagation**.

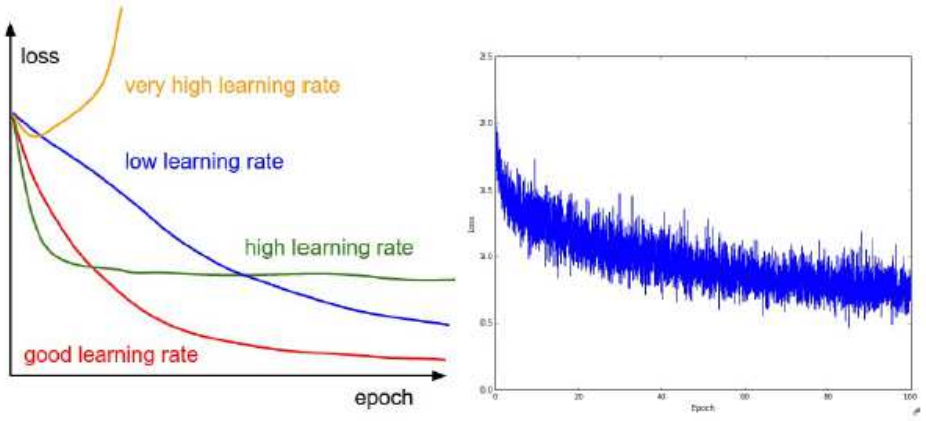


FIGURE 12.6: *On the* **left**, *a cartoon of the error rate encountered during training a multilayer network and the main phenomena you might observe. On the* **right**, *an actual example. Each of these figures is from the excellent course notes for the Stanford class cs231n Convolutional Neural Networks for Visual Recognition, written by Andrej Karpathy. You can find these notes at: http://cs231n.stanford.edu.*

12.2.3 Training Multiple Layers

A multilayer network represents an extremely complex, highly non-linear function, with an immense number of parameters. Training such networks is not easy. Neural networks are quite an old idea, but have only relatively recently had impact in practical applications. Hindsight suggests the problem is that networks are hard to train successfully. There is now a collection of quite successful tricks — I'll try to describe the most important — but the situation is still not completely clear.

The simplest training strategy is minibatch gradient descent. At round r, we have the set of weights $\theta^{(r)}$. We form the gradient for a minibatch $\nabla_\theta E$, and update the weights by taking a small step $\eta^{(r)}$ (usually referred to as the **learning rate**) backwards along the gradient, yielding

$$\theta^{(r+1)} = \theta^{(r)} - \eta^{(r)} \nabla_\theta E.$$

The most immediate difficulties are where to start, and what is $\eta^{(r)}$.

Initialization: As for a single layer of units, it is a bad idea to initialize each parameter to zero. It is usual to draw a sample of a zero mean normal random variable for each initial value. However, in a multilayer network, we may well have some units receiving input from more (or fewer) units than others (this is referred to as the **fan in** of the unit). Now assume that we have two units: one with many inputs, and one with few. If we initialize each units weights using the same zero mean normal random variable, the unit with more inputs will have a higher variance output (I'm ignoring the nonlinearity). This tends to lead to problems, because units at the next level will see unbalanced inputs. Experiment has shown that it is a good idea to allow the variance of the random variable you sample to

depend on the fan in of the unit whose parameter you are initializing. Write n for the fan in of the unit in question, and ϵ for a small non-negative number. Current best practice appears to be that one initializes each weight with an independent sample of a random variable with mean 0 and *variance*

$$\epsilon\frac{\sqrt{2}}{n}.$$

Choosing ϵ too small or too big can lead to trouble, but I'm not aware of any recipe for coming up with a good choice. Typically, biases are initialized either to 0, or to a small non-negative number; there is mild evidence that 0 is a better choice.

Learning rate: The remarks above about learning rate apply, but for more complicated networks it is usual to apply one of the methods of section 12.2.4, which adjust the gradient to get better optimization behavior.

Ensuring learning is proceeding: We need to keep track of what is going on inside the system as we train it. One way is to plot the loss as a function of the number of steps. These plots can be very informative (Figure 12.6). If the learning rate is small, the system will make very slow progress but may (eventually) end up in a good state. If the learning rate is large, the system will make fast progress initially, but will then stop improving, because the state will change too quickly to find a good solution. If the learning rate is very large, the system might even diverge. If the learning rate is just right, you should get fast descent to a good value, and then slow but fairly steady improvement. Of course, just as in the case of SVMs, the plot of loss against step isn't a smooth curve, but rather noisy. There is an amusing collection of examples of training problems at lossfunctions.tumblr.com It is quite usual to plot the error rate, or the accuracy, on a validation dataset while training. This will allow you to compare the training error with the validation error. If these are very different, you have a problem: the system is overfitting, or not generalizing well. You should increase the regularization constant.

Dead units: The remarks above apply.

Gradient problems: The remarks above apply.

Choosing the regularization constant: The remarks above apply, but for more complex networks, it is usual to use the more sophisticated regularization described in section **??** (at considerable training cost).

Does it work? Evaluating the classifier we have described is like evaluating any other classifier. You evaluate the error on a held-out data set that *wasn't* used to choose the regularization constant, or during training.

12.2.4 Gradient Scaling Tricks

Everyone is surprised the first time they learn that the best direction to travel in when you want to minimize a function is not, in fact, backwards down the gradient. The gradient *is* uphill, but repeated downhill steps are often not particularly efficient. An example can help, and we will look at this point several ways because different people have different ways of understanding this point.

We can look at the problem with algebra. Consider $f(x, y) = (1/2)(\epsilon x^2 + y^2)$, where ϵ is a small positive number. The gradient at (x, y) is $[\epsilon x, y]$. For simplicity, use a fixed learning rate η, so we have $\left[x^{(r)}, y^{(r)}\right] = \left[(1-\epsilon\eta)x^{(r-1)}, (1-\eta)y^{(r-1)}\right]$.

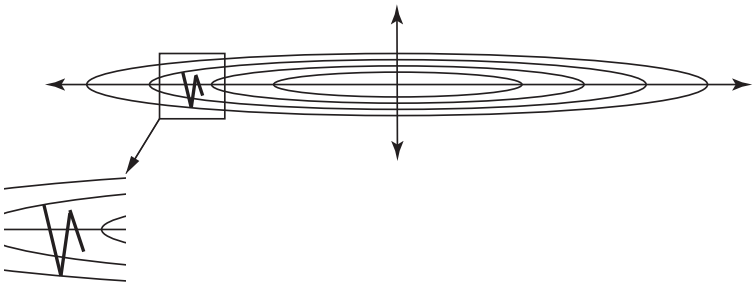


FIGURE 12.7: *A plot of the level curves (curves of constant value) of the function* $f(x,y) = (1/2)(\epsilon x^2 + y^2)$. *Notice that the value changes slowly with large changes in* x, *and quickly with small changes in* y. *The gradient points mostly toward the* x*-axis; this means that gradient descent is a slow zig-zag across the "valley" of the function, as illustrated. We might be able to fix this problem by changing coordinates,* if *we knew what change of coordinates to use.*

If you start at, say, $(x^{(0)}, y^{(0)})$ and repeatedly go downhill along the gradient, you will travel very slowly to your destination. You can show that $\left[x^{(r)}, y^{(r)}\right] = \left[(1-\epsilon\eta)^r x^{(0)}, (1-\eta)^r y^{(0)}\right]$. The problem is that the gradient in y is quite large (so y must change quickly) and the gradient in x is small (so x changes slowly). In turn, for steps in y to converge we must have $|1-\eta| < 1$; but for steps in x to converge, we require only the much weaker constraint $|1-\epsilon\eta| < 1$. Imagine we choose the largest η we dare for the y constraint. The y value will very quickly have small magnitude, though its sign will change with each step. But the x steps will move you closer to the right spot only extremely slowly.

Another way to see this problem is to reason geometrically. Figure 12.7 shows this effect for this function. The gradient is at right angles to the level curves of the function. But when the level curves form a narrow valley, the gradient points across the valley rather than down it. The effect isn't changed by rotating and translating the function (Figure 12.8).

You may have learned that Newton's method resolves this problem. This is

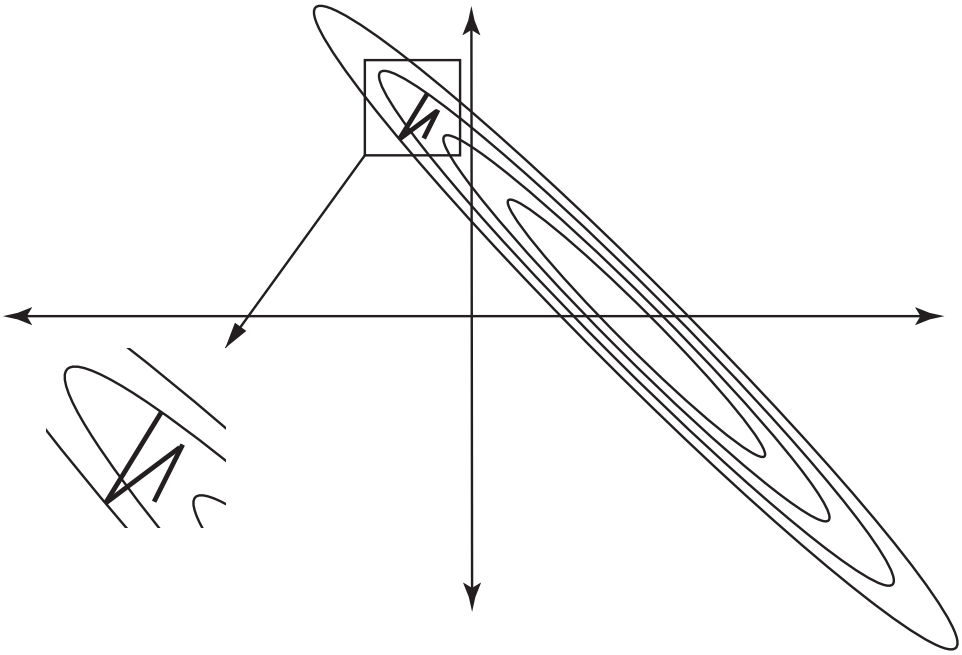


FIGURE 12.8: *Rotating and translating a function rotates and translates the gradient; this is a picture of the function of figure 12.7, but now rotated and translated. The problem of zig-zagging remains. This is important, because it means that we may have serious difficulty choosing a good change of coordinates.*

all very well, but to apply Newton's method we would need to know the matrix of second partial derivatives. A network can easily have thousands to millions of parameters, and we simply can't form, store, or work with matrices of these dimensions. Instead, we will need to think more qualitatively about what is causing trouble.

One useful insight into the problem is that fast changes in the gradient vector are worrying. For example, consider $f(x) = (1/2)(x^2 + y^2)$. Imagine you start far away from the origin. The gradient won't change much along reasonably sized steps. But now imagine yourself on one side of a valley like the function $f(x) = (1/2)(x^2 + \epsilon y^2)$; as you move along the gradient, the gradient in the x direction gets smaller very quickly, then points back in the direction you came from. You are not justified in taking a large step in this direction, because if you do you will end up at a point with a very different gradient. Similarly, the gradient in the y direction is small, and stays small for quite large changes in y value. You would like to take a small step in the x direction and a large step in the y direction.

You can see that this is the impact of the second derivative of the function (which is what Newton's method is all about). But we can't do Newton's method. We would like to travel further in directions where the gradient doesn't change much, and less far in directions where it changes a lot. There are several methods for doing so.

Momentum: We should like to discourage parameters from "zig-zagging" as in the example above. In these examples, the problem is caused by components of the gradient changing sign from step to step. It is natural to try and smooth the gradient. We could do so by forming a moving average of the gradient. Construct a vector $\mathbf{v}$, the same size as the gradient, and initialize this to zero. Choose a positive number $\mu < 1$. Then we iterate

$$\begin{aligned} \mathbf{v}^{(r+1)} &= \mu * \mathbf{v}^{(r)} + \eta \nabla_\theta E \\ \theta^{(r+1)} &= \theta^{(r)} - \mathbf{v}^{(r+1)} \end{aligned}$$

Notice that, in this case, the update is an average of all past gradients, each weighted by a power of μ. If μ is small, then only relatively recent gradients will participate in the average, and there will be less smoothing. Larger μ lead to more smoothing. A typical value is $\mu = 0.9$. It is reasonable to make the learning rate go down with epoch when you use momentum, but keep in mind that a very large μ will mean you need to take several steps before the effect of a change in learning rate shows.

Adagrad: We will keep track of the size of each component of the gradient. In particular, we have a running cache $\mathbf{c}$ which is initialized at zero. We choose a small number α (typically 1e-6), and a fixed η. Write $g_i^{(r)}$ for the i'th component of the gradient $\nabla_\theta E$ computed at the r'th iteration.Then we iterate

$$\begin{aligned} c_i^{(r+1)} &= c_i^{(r)} + (g_i^{(r)})^2 \\ \theta_i^{(r+1)} &= \theta_i^{(r)} - \eta \frac{g_i^{(r)}}{(c_i^{(r+1)})^{\frac{1}{2}} + \alpha} \end{aligned}$$

Notice that each component of the gradient has its own learning rate, set by the history of previous gradients.

RMSprop: This is a modification of Adagrad, to allow it to "forget" large gradients that occurred far in the past. Again, write $g_i^{(r)}$ for the i'th component of the gradient $\nabla_\theta E$ computed at the r'th iteration. We choose another number, Δ, (the **decay rate**; typical values might be 0.9, 0.99 or 0.999), and iterate

$$\begin{aligned} c_i^{(r+1)} &= \Delta c_i^{(r)} + (1 - \Delta)(g_i^{(r)})^2 \\ \theta_i^{(r+1)} &= \theta_i^{(r)} - \eta \frac{g_i^{(r)}}{(c_i^{(r+1)})^{\frac{1}{2}} + \alpha} \end{aligned}$$

Adam: This is a modification of momentum that rescales gradients, tries to forget large gradients, and adjusts early gradient estimates to correct for bias. Again, write $g_i^{(r)}$ for the i'th component of the gradient $\nabla_\theta E$ computed at the r'th iteration. We choose three numbers β_1, β_2 and ϵ (typical values are 0.9, 0.999 and

1e-8, respectively), and some steplength or learning rate η. We then iterate

$$\begin{aligned}
\mathbf{v}^{(r+1)} &= \beta_1 * \mathbf{v}^{(r)} + (1-\beta_1) * \nabla_\theta E \\
c_i^{(r+1)} &= \beta_2 * c_i^{(r)} + (1-\beta_2) * (g_i^r)^2 \\
\hat{\mathbf{v}} &= \frac{\mathbf{v}^{(r+1)}}{1-\beta_1^t} \\
\hat{c}_i &= \frac{\hat{c}_i^{(r+1)}}{1-\beta_2^t} \\
\theta_i^{(r+1)} &= \theta_i^{(r)} - \eta \frac{\hat{v}_i}{\sqrt{\hat{c}_i} + \epsilon}
\end{aligned}$$

As of writing, Adam seems to be the most widely used method, and is likely the method of choice.

12.2.5 Dropout

Regularizing by the square of the weights is all very well, but quite quickly we will have problems because there are so many weights. An alternative, and very useful, regularization strategy is to try and ensure that no unit relies too much on the output of any other unit. One can do this as follows. At each training step, randomly select some units, set their outputs to zero (and reweight the inputs of the units receiving input from them), and then take the step. Now units are trained to produce reasonable outputs even if some of their inputs are randomly set to zero — units can't rely too much on one input, because it might be turned off. Notice that this sounds sensible, but it isn't quite a proof that the approach is sound; that comes from experiment. The approach is known as **dropout**.

There are some important details we can't go into. Output units are not subject to dropout, but one can also turn off inputs randomly. At test time, there is no dropout. Every unit computes its usual output in the usual way. This creates an important training issue. Write p for the probability that a unit is dropped out, which will be the same for all units subject to dropout. You should think of the expected output of the i'th unit at *training* time as $(1-p)o_i$ (because with probability p, it is zero). But at test time, the next unit will see o_i; so at training time, you should reweight the inputs by $1/(1-p)$. In exercises, we will use packages that arrange all the details for us.

12.2.6 It's Still Difficult..

All the tricks above are helpful, but training a multilayer neural network is still difficult. Fully connected layers have many parameters. It's quite natural to take an input feature vector of moderate dimension, build one layer that produces a much higher dimensional vector, then stack a series of quite high dimensional layers on top of that. There is quite good evidence that having many layers can improve practical performance *if* one can train the resulting network. Such an architecture has been known for a long time, but hasn't been particularly successful until recently.

There are several structural obstacles. Without GPU's, evaluating such a network can be slow, making training slow. The number of parameters in just one

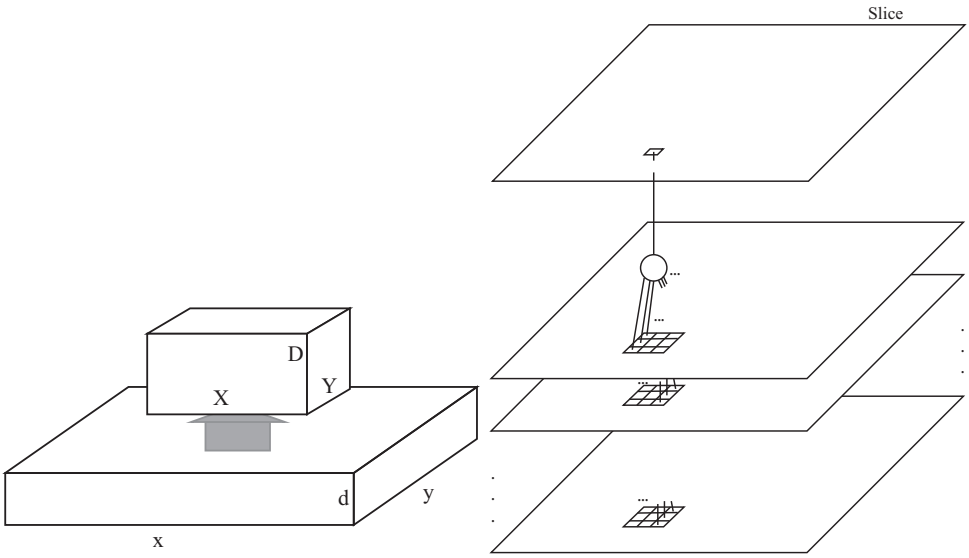


FIGURE 12.9: *Terminology for building a convolutional layer. On the* **left**, *a layer turns one block of data into another. The x and y coordinates index the position of a location in the block, and the d coordinate identifies the data item at that point. A natural example of a block is a color image, which usually has three layers (d=3); then x and y identify the pixel, and d chooses the R, G, or B slice. The dimensions x and y are the* spatial *dimensions. On the right* **right**, *a unit in a convolutional layer forms a weighted sum of set of locations, adds a bias, then applies a RELU. There is one unit for each (X, Y, D) location in the output block. For each (X, Y) in the output block, there is a corresponding window of size* (w_x, w_y) *in the (x, y) space. Each of the D units whose responses form the D values at the (X, Y) location in the output block forms a weighted sum of all the values covered by that window. These inputs come from each slice in the window below that unit (so the number of inputs is* $d \times w_x \times w_y$*). Each of the units that feed the a particular slice in the output block has the same set of weights, so you should think of a unit as a form of pattern detector; it will respond strongly if the block below it is "similar" to the weights.*

fully connected layer is high, meaning that multiple layers will need a lot of data to train, and will take many training batches. There is some reason to believe that multilayer neural networks were discounted in application areas for quite a long time because people underestimated just how much data and how much training was required to make them perform well.

One obstacle that remains technically important has to do with the gradient. Look at the recursion I described for backpropagation. The gradient update at the L'th (top) layer depends pretty directly on the parameters in that layer. But now consider a layer close to the input end of the network. The gradient update has been multiplied by several Jacobian matrices. The update may be very small (if these Jacobians shrink their input vectors) or unhelpful (if layers close to the output

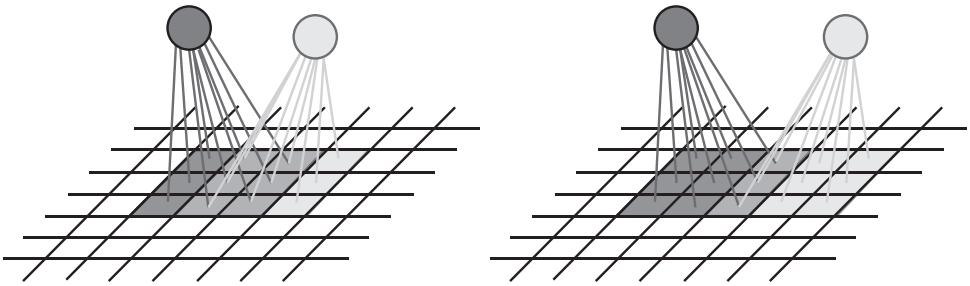


FIGURE 12.10: *Figure 12.9 shows units taking an input block and creating an output block. Each unit is fed by a window on the spatial dimensions of the input block. The window is advanced by the* stride *to feed the next unit. On the* **left***, two units fed by 3x3 windows with a stride of 1. I have shaded the pixels in the window to show which pixels go to which unit; notice the units share 6 pixels. In this case, the spatial dimensions of the output block will be either the same as those of the input block (if we find some values to feed pixels in windows that hang over the edge of the image), or only slightly smaller (if we ignore units whose windows hang over the edges of the image). On the* **right***, two units fed by 3x3 windows with a stride of 2. Notice the units share fewer pixels, and the output block will be smaller than the input block.*

have poor parameter estimates). For the gradient update to be really helpful, we'd like the layers higher up the network to be right; but we can't achieve this with lower layers that are confused, because they pass their outputs up. If a layer low in the network is in a nonsensical state, it may be very hard to get it out of that state. In turn, this means that adding layers to a network might improve performance, but also might make it worse because the training turns out poorly.

There are a variety of strategies for dealing with this problem. We might just train for a very long time, possibly using gradient rescaling tricks. We might reduce the number of parameters in the layers, by passing to convolutional layers (below) rather than fully connected layers. We might use various tricks to initialize each layer with a good estimate. This is a topic of widespread current interest, but one I can't deal with in any detail here. Finally, we might use architectural tricks (section 10.1) to allow inputs to bypass layers, so that poorly trained layers create fewer difficulties.

12.3 CONVOLUTIONAL NEURAL NETWORKS

One area where neural networks have had tremendous impact is in image understanding. Images have special properties that motivate special constructions of features. These constructions yield a layer architecture that has a significantly reduced number of parameters in the layer. This architecture has proven useful in other applications, but we will work with images.

12.3.1 Images and Convolutional Layers

Using the output of one layer to form features for another layer is attractive. The natural consequence of this idea is that the input to the network would be the image pixels. But this presents some important difficulties. There are an awful lot of pixels in most images, and that means it's likely that there will be an awful lot of parameters.

For a variety of reasons, it doesn't make much sense to have an input layer consisting of units each of which sees all the image pixels. There would be a tremendous number of weights to train. It would be hard to explain why units have different values for the weights. Instead, we can use some intuitions from computer vision.

First, we need to build systems that can handle different versions of what is essentially the same image. For example, imagine you turn the camera slightly to one side, or raise it or lower it when taking the image. If every input unit sees every pixel, this would constrain the form of the weights. Turning the camera up a bit shifts the image down a bit; the representation shouldn't be severely disrupted by this. Second, long experience in computer vision has produced a (very rough) recipe for building image features: you construct features that respond to patterns in small, localized neighborhoods; then other features look at patterns of *those* features; then others look at patterns of those, and so on (*big fleas have little fleas upon their backs to bite 'em; and little fleas have smaller ones, and so ad infinitum*).

We will assume that images are 3D. The first two dimensions will be the x and y dimensions in the image, the third (for the moment!) will identify the color layer of the image (for example, R, G and B). We will build layers that take 3D objects like images (which I will call **blocks**) and make new blocks (Figure 12.9; notice the input block has dimension $x \times y \times d$ and the output block has dimension $X \times Y \times D$). Each block is a stack of **slices**, which — like color layers in an image — have two spatial dimensions.

These layers will draw from a standard recipe for building an image feature that describes a small neighborhood. We construct a **convolution kernel**, which is a small block. This is typically odd sized, and typically from 3×3 to a few tens by a few tens in size along the spatial dimensions, and is always of size d in the other dimension.

Write $\mathcal{I}$ for a block (for example, an image), $\mathcal{K}$ for the kernel, b for a bias term (which might be zero; some, but not all, convolutional layers use a bias term), and $\mathcal{I}_{ijk}$ for the i, j'th pixel in the k'th slice of the block. Write F for the function implemented by a RELU, so that $F(x) = \max(0, x)$. Now we form a slice $\mathcal{O}$, whose u, v'th entry is

$$\mathcal{O}_{uv} = F(\mathcal{W}_{uv} + b) = F(\sum_{ijk} \mathcal{I}_{u+i,v+j,k}\mathcal{K}_{ijk} + b),$$

where I am assuming the sum goes over all values of i and j, and if the indices to either $\mathcal{I}$ or $\mathcal{K}$ go outside the domain, then the reported value is zero. There is room for some confusion here, because one can use a variety of different indexing schemes, and different authors use different ones (usually for compatibility with the history of convolution); this is of no significance. Figure 12.9 illustrates the process

that produces a slice from a block.

The operation that produces $\mathcal{W}$ from $\mathcal{I}$ and $\mathcal{K}$ is known as **convolution**, and it is usual to write $\mathcal{W} = \mathcal{K} * \mathcal{I}$. We will not go into all the properties of convolution, but you should notice one extremely important property. We obtain the value at a pixel by centering $\mathcal{K}$ on that pixel. We now have a patch sitting over all the layers of the image at some location; we multiply the pixels in that patch (by layer) by the corresponding image pixels (in layers), then accumulate the products — the result goes into $\mathcal{O}$. This is like a dot-product — we will get a large positive value in $\mathcal{O}$ at that pixel if the image window around that pixel looks like $\mathcal{K}$, and a small negative value if they're the same up to a sign change. You should think of a convolution kernel as being an example pattern.

Now when you convolve a kernel with an image, you get, at each location, an estimate of how much that image looks like that kernel at that point. The output is the response of a collection of simple pattern detectors, one at each pixel, *for the same pattern*. We may not need every such output; instead, we might look at every second (third, etc.) pixel in each direction. This choice is known as the **stride**. A stride of 1 corresponds to looking at every pixel; of 2, every second pixel; and so on (Figure 12.10)

A slice can be interpreted as a map, giving the response of a local feature detector at every (resp. every second; every third; etc.) pixel. At each pixel of interest (i.e. every pixel; every second, etc. depending on stride), we place a window (which should be odd-sized, to make indexing easier). Every pixel in that window is an input to a unit that corresponds to the window, which multiplies each pixel by a weight, sums all these terms, then applies a RELU. What makes a slice special is that *each unit uses the same set of weights*. The size of this object depends a little on the software package you are using. Assume the input image is of size $n_x \times n_y \times n_z$, and the kernel is of size $2k_x + 1 \times 2k_y + 1 \times n_z$. At least in principle, you cannot place a unit over a pixel that is too close to the edge, because then some of its inputs are outside the image. You could pad the image (either with constants, or by reflecting it, or by attaching copies of the columns/rows at the edge) and supply these inputs; in this case, the output could be $n_x \times n_y \times n_z$. Otherwise, you could place units only over pixels where all of the unit's inputs are inside the image. Then you would have an output of size $n_x - 2k_x \times n_y - 2n_y \times n_z$. The kernel is usually small, so the difference in sizes isn't that great. Most software packages are willing to set up either case.

A slice finds locations in the image where a particular pattern (identified by the weights) occurs. We could attach many slices to the image. They should all have the same stride, so they're all the same size. The output of this collection of slices would be one vector at each pixel location, where the components of the vector represent the similarity between the image patch centered at that location and a particular pattern. A collection of slices is usually referred to as a **convolutional layer**. You should think of a convolutional layer as being like a color image. There are now may different color layers (the slices), so that dimension has been expanded.

12.3.2 Convolutional Layers upon Convolutional Layers

Now the output of the initial convolutional layer is a set of slices, registered to the input image, forming a block of data. That looks like the input of that layer (a set of slices — color layers) forming a **block** of data. This suggests we could use the output of the first convolutional layer could be connected to a second convolutional layer, a second to a third, and so on. Doing so turns out to be an excellent idea.

Think about the output of the first convolutional layer. Each location receives inputs from pixels in a window about that location. Now if we put a second layer on top of the first, each location in the second receives inputs from first layer values in a window about that location. This means that locations in the second layer are affected by a larger window of pixels than those in the first layer. You should think of these as representing "patterns of patterns". If we place a third layer on top of the second layer, locations in that third layer will depend on an even larger window of pixels. A fourth layer will depend on a yet larger window, and so on.

12.3.3 Pooling

If you have several convolutional layers with stride 1, then each block of data has the same spatial dimensions. This tends to be a problem, because the pixels that feed a unit in the top layer will tend to have a large overlap with the pixels that feed the unit next to it. In turn, the values that the units take will be similar, and so there will be redundant information in the output block. It is usual to try and deal with this by making blocks get smaller. One natural strategy is to occasionally have a layer that has stride 2.

An alternative strategy is to use **max pooling**. A pooling unit reports the largest value of its inputs. In the most usual arrangement, a pooling layer will take an (x, y, d) block to a $(x/2, y/2, d)$ block. For the moment, ignore the entirely minor problems presented by a fractional dimension. The new block is obtained by pooling units that pool a 2x2 window at each slice of the input block to form each slice of the output block. These units are placed so they don't overlap, so the output block is half the size of the input block (for some reason, this configuration is hard to say but easy to see; Figure 12.11). If x or y or both are odd, there are two options; one could ignore the odd pixel on the boundary, or one could build a row (column; both) of imputed values, most likely by copying the row (column; both) on the edge. These two strategies yield, respectively, floor$(x/2)$ and ceil$(x/2)$ for the new dimension. Pooling seems to be falling out of favor, but not so much or so fast that you will not encounter it.

12.4 EXAMPLE: BUILDING AN IMAGE CLASSIFIER

There are two problems that lie at the core of image understanding. The first is **image classification**, where we decide what class an image of a fixed size belongs to. The taxonomy of classes is provided in advance, but it's usual to work with a collection of images of objects. These objects will be largely centered in the image, and largely isolated. Each image will have an associated object name. There are many collections with this structure. The best known, by far, is `ImageNet`, which can be found at http://www.image-net.org. There is a regular competition to

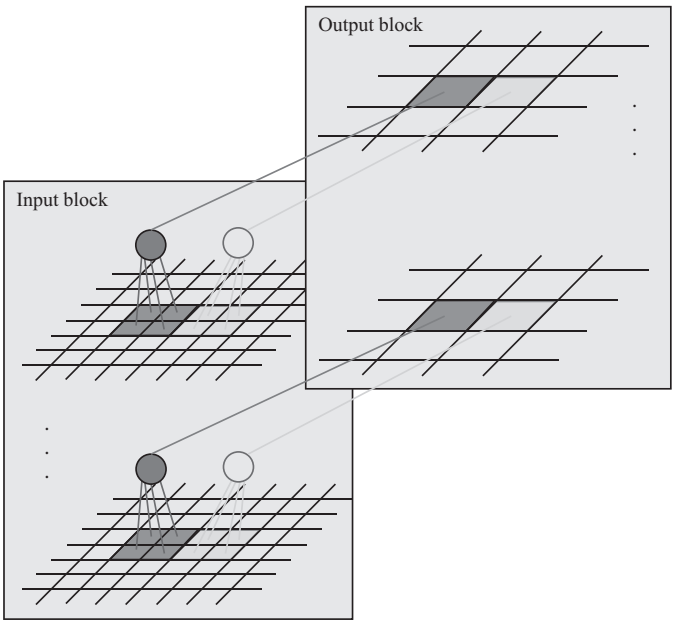


FIGURE 12.11: *In a pooling layer, pooling units compute the largest value of their inputs, then pass it on. The most common case is 2x2, illustrated here. We tile each slice with 2x2 windows that do not overlap. Pooling units compute the max, then pass that on to the corresponding location in the corresponding slice of the output block. As a result, the spatial dimensions of the output block will be about half those of the input block (details depend on how one handles windows that hang over the edge.*

classify ImageNet images. Be aware that, while this chapter tries to give a concise description of best practice, you might need to do more than read it to do well in the competition.

The second problem is **object detection**, where we try to find the locations of objects of a set of classes in the image. So we might try to mark all cars, all cats, all camels, and so on. As far as anyone knows, the right way to think about object detection is that we search a collection of windows in an image, apply an image classification method to each window, then resolve disputes between overlapping windows. How windows are to be chosen for this purpose is an active and quickly changing area of research. We will regard image classification as the key building block, and ignore the question of deciding which window to classify.

We have most of the pieces to build an image classifier. Architectural choices will make a difference to its performance. So will a series of tricks.

12.4.1 An Image Classification Architecture

We can now put together an image classifier. A convolutional layer receives image pixel values as input. The output is fed to a stack of convolutional layers, each

feeding the next. The output of the final layer is fed to one or more fully connected layers, with one output per class. The whole is trained by batch gradient descent, or a variant, as above.

There are a number of architectural choices to make, which are typically made by experiment. The main ones are the choice of the number of convolutional layers; the choice of the number of slices in each layer; and the choice of stride for each convolutional layer. There are some constraints on the choice of stride. The first convolutional layer will tend to have stride 1, so that we see all the resolution of the image. But the outputs of that layer are likely somewhat correlated, because they depend on largely the same set of pixels. Later layers might have larger stride for this reason. In turn, this means the spatial dimensions of the representation will get smaller.

Notice that different image classification networks differ by relatively straightforward changes in architectural parameters. Mostly, the same thing will happen to these networks (variants of batch gradient descent on a variety of costs; dropout; evaluation). In turn, this means that we should use some form of specification language to put together a description of the architecture of interest. Ideally, in such an environment, we describe the network architecture, choose an optimization algorithm, and choose some parameters (dropout probability, etc.). Then the environment assembles the net, trains it (ideally, producing log files we can look at) and runs an evaluation. Several such environments exist.

12.4.2 Useful Tricks - 1: Preprocessing Data

It usually isn't possible to simply feed any image into the network. We want each image fed into the network to be the same size. We can achieve this either by resizing the image, or by cropping the image. Resizing might mean we stretch or squash some images, which likely isn't a great idea. Cropping means that we need to make a choice about where the crop box lies in the image. Practical systems quite often apply the same network to different croppings of the same image. For our purposes, we will assume that all the images we deal with have the same size.

It is usually wise to preprocess images before using them. This is because two images with quite similar content might have rather different pixel values. For example, compare image $\mathcal{I}$ and $1.5\mathcal{I}$. One will be brighter than the other, but nothing substantial about the image class will have changed. There is little point in forcing the network to learn something that we know already. There are a variety of preprocessing options, and different options have proven to be best for different problems. I will sketch some of the more useful ones.

You could **whiten pixel values**. You would do this for each pixel in the image grid independently. For each pixel, compute the mean value at that pixel across the training dataset. Subtract this, and divide the result by the standard deviation of the value at that pixel across the training dataset. Each pixel location in the resulting stack of images has mean zero and standard deviation one. Reserve the offset image (the mean at each pixel location) and the scale image (ditto, standard deviation) so that you can normalize test images.

You could **contrast normalize the image** by computing the mean and standard deviation of pixel values in each training (resp. test) image, then subtracting

the mean from the image and dividing the result by the standard deviation.

You could **contrast normalize pixel values locally**. To do so, you compute a smoothed version of the image (convolve with a Gaussian, for insiders; everyone else should skip this paragraph, or perhaps search the internet). You can think of the result as a local estimate of the image mean. At each pixel, you subtract the smoothed value from the image value.

Useful Facts: 12.1 *Whitening a dataset*

For a dataset $\{\mathbf{x}\}$, compute:

- $\mathcal{U}$, the matrix of eigenvectors of $\mathsf{Covmat}(\{\mathbf{x}\})$;
- and $\mathsf{mean}(\{\mathbf{x}\})$.

Now compute $\{\mathbf{n}\}$ using the rule

$$\mathbf{n}_i = \mathcal{U}^T(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\})).$$

Then $\mathsf{mean}(\{\mathbf{n}\}) = \mathbf{0}$ and $\mathsf{Covmat}(\{\mathbf{n}\})$ is diagonal.
Now write Λ for the diagonal matrix of eigenvalues of $\mathsf{Covmat}(\{\mathbf{x}\})$ (so that $\mathsf{Covmat}(\{\mathbf{x}\})\mathcal{U} = \mathcal{U}\Lambda$). Assume that each of the diagonal entries of Λ is greater than zero (otherwise there is a redundant dimension in the data). Write λ_i for the i'th diagonal entry of Λ, and write $\Lambda^{-(1/2)}$ for the diagonal matrix whose i'th diagonal entry is $1/\sqrt{\lambda_i}$. Compute $\{\mathbf{z}\}$ using the rule

$$\mathbf{z}_i = \Lambda^{(-1/2)}\mathcal{U}(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\})).$$

We have that $\mathsf{mean}(\{\mathbf{z}\}) = 0$ and $\mathsf{Covmat}(\{\mathbf{z}\}) = \mathcal{I}$. The dataset $\{\mathbf{z}\}$ is often known as **whitened data**.

You could whiten the image as in section 12.1. It turns out this doesn't usually help all that much. Instead, you need to use **ZCA-whitening**. I will use the same notation as chapter 10.1, but I reproduce the useful facts box here as a reminder. Notice that, by using the rule

$$\mathbf{z}_i = \Lambda^{(-1/2)}\mathcal{U}(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\})),$$

we have rotated the data in the high dimensional space. In the case of images, this means that the image corresponding to $\mathbf{z}_i$ will likely not look like anything coherent. Furthermore, if there are very small eigenvalues, the scaling represented by $\Lambda^{-(1/2)}$ may present serious problems. But notice that the covariance matrix of a dataset is unaffected by rotation. We could choose a small non-negative constant ϵ, and use the rule

$$\mathbf{z}_i = \mathcal{U}^T(\Lambda + \epsilon\mathcal{I})^{(-1/2)}\mathcal{U}(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\}))$$

instead. The result looks significantly more like an image, and will have a covariance matrix that is the identity (or close, depending on the value of ϵ). This rule is ZCA whitening.

12.4.3 Useful Tricks - 2: Enhancing Training Data

Datasets of images are never big enough to show all effects accurately. This is because an image of a horse is still an image of a horse even if it has been through a small rotation, or has been resized to be a bit bigger or smaller, or has been cropped differently, and so on. There is no way to take account of these effects in the architecture of the network. Generally, a better approach is to expand the training dataset to include different rotations, scalings, and crops of images.

Doing so is relatively straightforward. You take each training image, and generate a collection of extra training images from it. You can obtain this collection by: resizing and then cropping the training image; using different crops of the same training image (assuming that training images are a little bigger than the size of image you will work with); rotating the training image by a small amount, resizing and cropping; and so on. You can't crop too much, because you need to ensure that the modified images are still of the relevant class, and an aggressive crop might cut out the horse, etc. When you rotate then crop, you need to be sure that no "unknown" pixels find their way into the final crop. All this means that only relatively small rescales, crops, rotations, etc. will work. Even so, this approach is an extremely effective way to enlarge the training set.

12.4.4 Useful Tricks - 3: Batch Normalization

There is good experimental evidence that large values of inputs to any layer within a neural network lead to problems. One source of the problem could be this. Imagine some input to some unit has a large absolute value. If the corresponding weight is relatively small, then one gradient step could cause the weight to change sign. In turn, the output of the unit will swing from one side of the RELU's non-linearity to the other. If this happens for too many units, there will be training problems because the gradient is then a poor prediction of what will actually happen to the output. So we should like to ensure that relatively few values at the input of any layer have large absolute values. We will build a new layer, sometimes called a **batch normalization layer**, which can be inserted between two existing layers.

Write $\mathbf{x}^b$ for the input of this layer, and $\mathbf{o}^b$ for its output. The output has the same dimension as the input, and I shall write this dimension d. The layer has two vectors of parameters, γ and β, each of dimension d. Write $\text{diag}(\mathbf{v})$ for the matrix whose diagonal is $\mathbf{v}$, and with all other entries zero. Assume we know the mean ($\mathbf{m}$) and standard deviation ($\mathbf{s}$) of each component of $\mathbf{x}^b$, where the expectation is taken over all relevant data. The layer forms

$$\begin{aligned} \mathbf{x}^n &= [\text{diag}(\mathbf{s}+\epsilon)]^{-1}\left(\mathbf{x}^b - \mathbf{m}\right) \\ \mathbf{o}^b &= [\text{diag}(\gamma)]\,\mathbf{x}^n + \beta. \end{aligned}$$

Notice that the output of the layer is a differentiable function of γ and β. Notice also that this layer *could* implement the identity transform, if $\gamma = \text{diag}(\mathbf{s}+\epsilon)$ and

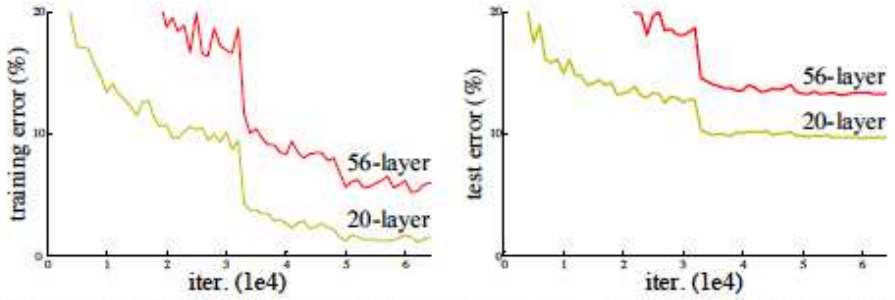


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

FIGURE 12.12: *This figure from* Deep Residual Learning for Image Recognition *Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun, ICCV 2016, illustrates the difficulties presented by training a deep network.*

$\beta = \mathbf{m}$. We adjust the parameters in training to achieve the best performance. It can be helpful to think about this layer as follows. The layer rescales its input to have zero mean and unit standard deviation, then allows training to readjust the mean and standard deviation as required. In essence, we expect that large values encountered between layers are likely an accident of the difficulty training a network, rather than required for good performance.

The difficulty here is we don't know either $\mathbf{m}$ or $\mathbf{s}$, because we don't know the parameters used for previous layers. Current practice is as follows. First, start with $\mathbf{m} = \mathbf{0}$ and $\mathbf{s} = \mathbf{1}$ for each layer. Now choose a minibatch, and train the network using that minibatch. Once you have taken enough gradient steps and are ready to work on another minibatch, reestimate $\mathbf{m}$ as the mean of values of the inputs to the layer, and $\mathbf{s}$ as the corresponding standard deviations. Now obtain another minibatch, and proceed. Remember, γ and β are parameters that are trained, just like the others (using gradient descent, momentum, adagrad, or whatever). Once the network has been trained, one then takes the mean (resp. standard deviation) of the layer inputs over the training data for $\mathbf{m}$ (resp. $\mathbf{s}$). Most neural network implementation environments will do all the work for you. It is quite usual to place a batch normalization layer between each layer within the network.

There is a general agreement that batch normalization improves training, but some disagreement about the details. Experiments comparing two networks, one with batch normalization the other without, suggest that the same number of steps tends to produce a lower error rate when batch normalized. Some authors suggest that convergence is faster (which isn't quite the same thing). Others suggest that larger learning rates can be used.

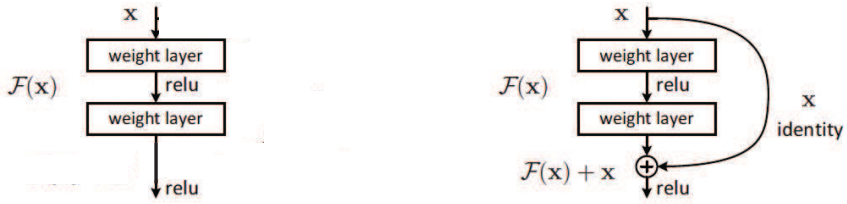


Figure 2. Residual learning: a building block.

FIGURE 12.13: *This figure, which is revised from* Deep Residual Learning for Image Recognition *Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun, ICCV 2016, conveys the intention of a residual network.*

12.4.5 Useful Tricks - 4: Residual Networks

A randomly initialized deep network can so severely mangle its inputs that only a wholly impractical amount of training will cause the latest layers to do anything useful. As a result, there have been practical limits on the number of layers that can be stacked (Figure 12.12). One recent strategy for avoiding this difficulty is to build a **residual layer**. Figure 12.13 sketches the idea in the form currently best understood. Remember, $F(x)$ is a RELU. Our usual layers produce $\mathbf{x}^{l+1} = \mathcal{F}(\mathbf{x}^l;\theta) = F(\mathcal{W}\mathbf{x}^l + \mathbf{b})$ as its output. This layer could be anything, but is most likely a fully connected or a convolutional layers. Then we can replace this layer with one that produces

$$\begin{aligned}\mathbf{x}^{l+1} &= F(\mathbf{x}^l + \mathcal{W}_1\mathbf{q} + \mathbf{b}_1)\\ \mathbf{q} &= F(\mathcal{W}_2\mathbf{x}^l + \mathbf{b}_2).\end{aligned}$$

It is usual, if imprecise, to think of this as producing an output that is $\mathbf{x} + \mathcal{F}(\mathbf{x};\theta)$ — the layer passes on its input with a residual added to it. The point of all this is that, at least in principle, this residual layer can represent its output as a small offset on its input. If it is presented with large inputs, it can produce large outputs by passing on the input. Its output is also significantly less mangled by stacking layers, because its output is largely given by its input plus a non-linear function.

Very recently, an improvement on this strategy has surfaced, in *Identity Mappings in Deep Residual Networks* by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (which you can find on ArXiV using a search engine). Rather than use the expression above (corresponding to Figure 12.13), we use a layer that produces

$$\begin{aligned}\mathbf{x}^{l+1} &= \mathbf{x}^l + \mathcal{W}_1\mathbf{q} + \mathbf{b}_1\\ \mathbf{q} &= F(\mathcal{W}_2F(\mathbf{x}^l) + \mathbf{b}_2).\end{aligned}$$

It is rather more informative to think of this as producing an output that is $\mathbf{x} + \mathcal{F}(\mathbf{x};\theta)$ — the layer passes on its input with a residual added to it. There is good

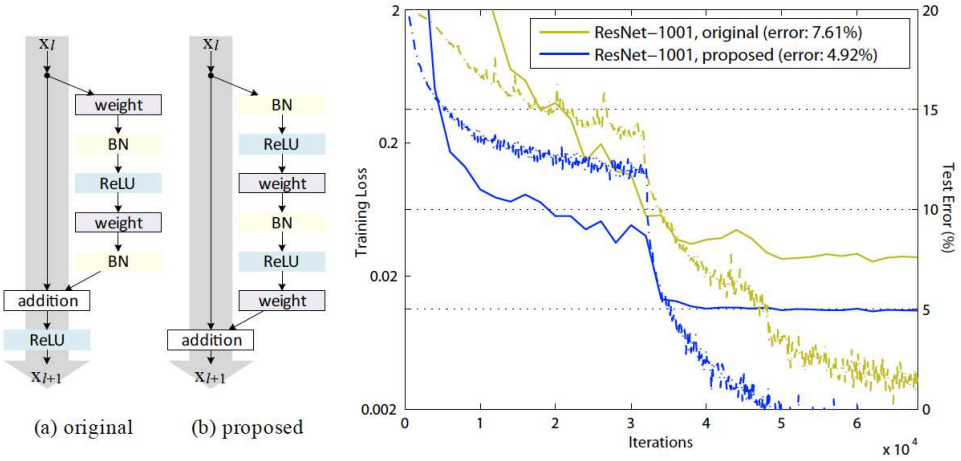


FIGURE 12.14: *This figure is revised from* Identity Mappings in Deep Residual Networks *by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (which you can find on ArXiV using a search engine). On the* **left**, *(a) shows the original residual network architecture, also described in Figure 12.13; (b) shows the current best architecture. On the* **right**, *train (dashed) and test (full) curves for the old and new architectures on CIFAR-10. Notice the significant improvement in performance.*

evidence that such layers can be stacked very deeply indeed (the paper I described uses 1001 layers to get under 5% error on CIFAR-10; beat that if you can!). One reason is that there are useful components to the gradient for each layer that do not get mangled by previous layers. You can see this by considering the Jacobian of such a layer with respect to its inputs. You will see that this Jacobian will have the form

$$\mathcal{J}_{x^{l+1};x^l} = (\mathcal{I} + \mathcal{M}_l)$$

where $\mathcal{I}$ is the identity matrix and $\mathcal{M}_l$ is a set of terms that depend on $\mathcal{W}$ and $\mathbf{b}$. Now remember that, when we construct the gradient at the k'th layer, we evaluate by multiplying a set of Jacobians corresponding to the layers above. This product in turn must look like

$$\mathcal{J}_{o;\theta^k} = \mathcal{J}_{o;\mathbf{x}^{k+1}}\mathcal{J}_{\mathbf{x}^{k+1};\theta^k} = (\mathcal{I} + \mathcal{M}_1 + \ldots)\mathcal{J}_{\mathbf{x}^{k+1};\theta^k}$$

which means that some components of the gradient at that layer do not get mangled by being passed through a sequence of poorly estimated Jacobians. One reason I am having trouble making a compelling argument explaining why this architecture is better is that the argument doesn't seem to be known in any tighter form (it certainly isn't to me). There is overwhelming evidence that the architecture is, in practice, better; it has produced networks that are (a) far deeper and (b) far more accurate than anything produced before. But why it works remains somewhat veiled.

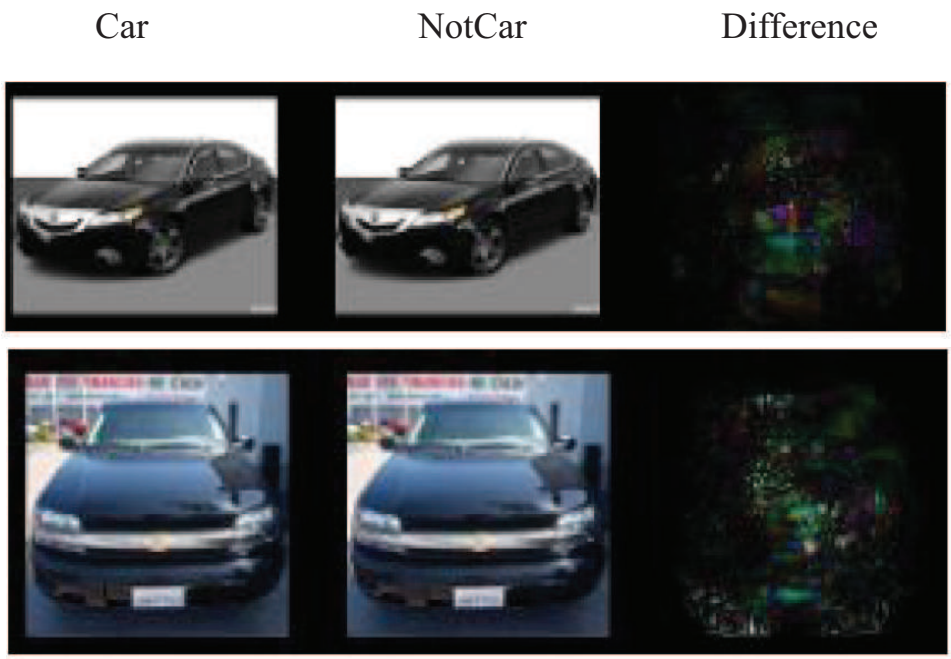


FIGURE 12.15: *This figure is from "Intriguing properties of neural networks", by Christian Szegedy* et al.*. The* **left** *column shows images classified as* `car`*; the* **middle** *column shows modified versions of those images classified as* `not car`*; and the* **right** *column shows the difference.*

12.5 ADVERSARIAL EXAMPLES

Adversarial examples are a curious experimental property of neural network image classifiers. Here is what happens. Assume you have an image $\mathbf{x}$ that is correctly classified with label l. The network will produce a probability distribution over labels $P(L|\mathbf{x})$. Choose some label k that is not correct. It is possible to use modern optimization methods to search for a modification to the image $\delta\mathbf{x}$ such that

$$\begin{aligned} \delta\mathbf{x} \quad & \text{is small} \\ & \textit{and} \\ P(k|\mathbf{x}+\delta x) \quad & \text{is large.} \end{aligned}$$

You might expect that $\delta\mathbf{x}$ is "large"; what is surprising is that mostly it is so tiny as to be imperceptible to a human observer. For example, the labels might be `car` and `not-car`. Figure 12.15 shows two images correctly labelled `car`, and the revisions required to make the image get the label `not-car`. The changes can't be detected by eye. Similarly, figure 12.16 shows what is required to turn a `panda` into a `nematode`. Again, the changes can't be detected by eye.

The property of being an adversarial example seems to be robust to image smoothing, simple image processing, and printing and photographing (see figure 12.17). The existence of adversarial examples raises the following, rather

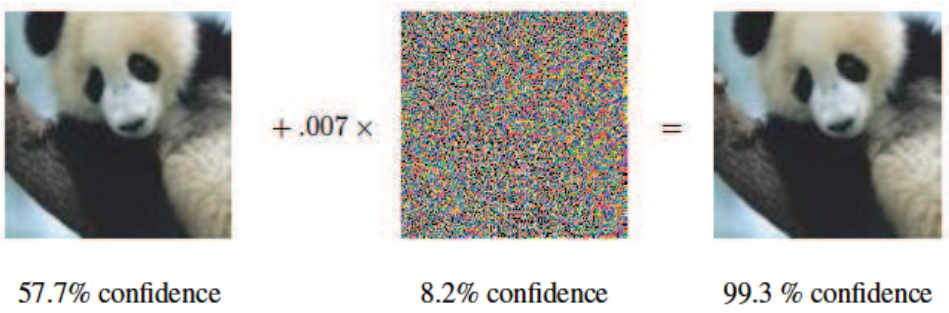


FIGURE 12.16: *This figure is from "EXPLAINING AND HARNESSING ADVERSARIAL EXAMPLES", by Ian Goodfellow* et al.. *The* **left** *column shows a* `panda`*; the* **middle** *column shows a modification (which has been exaggerated to make it non-grey; note the multiplier); and the* **right** *column shows a* `nematode`*.*

alarming, prospect: You could make a template that you could hold over a stop sign, and with one pass of a spraypaint can, turn that sign into something that is interpreted as a minimum speed limit sign by current computer vision systems. I haven't seen this demonstration done yet, but it appears to be entirely within the reach of modern technology, and it and activities like it offer significant prospects for mayhem.

What is startling about this behavior is that it is exhibited by networks that are very good at image classification, *assuming* that no-one has been fiddling with the images. So modern networks are very accurate on untampered pictures, but may behave very strangely in the presence of tampering. One can (rather vaguely) identify the source of the problem, which is that neural network image classifiers have far more degrees of freedom than can be pinned down by images. This observation doesn't really help, though, because it doesn't explain why they (mostly) work rather well, and it doesn't tell us what to do about adversarial examples. There have been a variety of efforts to produce networks that are robust to adversarial examples, but evidence right now is based only on experiment (some networks behave better than others) and we are missing clear theoretical guidance.

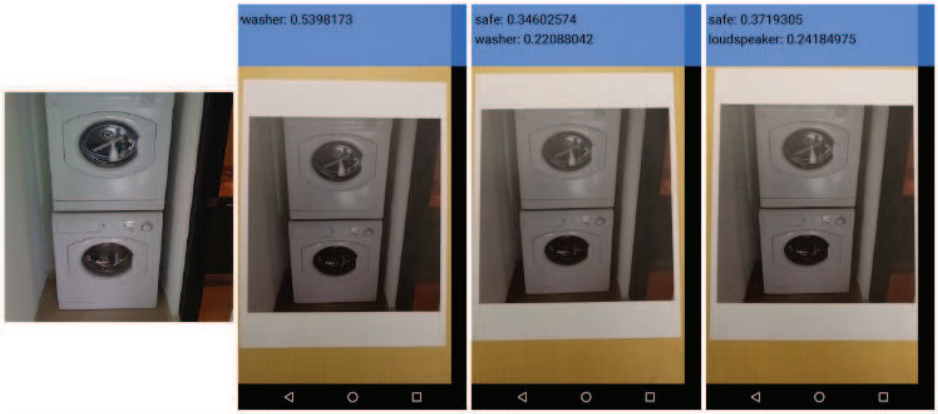


FIGURE 12.17: *This figure is from "ADVERSARIAL EXAMPLES IN THE PHYSICAL WORLD", by Alexey Kurakin* et al.*. The authors photographed the washing machine on the* **left**. *They then prepared clean and adversarial versions of that image and printed them. Finally, they photographed them with a cell-phone camera, producing the images with overprinting. The first is a photo of the printed washing machine, and is correctly classified as a* `washer`*; the second and third are photos of printed adversarial examples, and are (respectively) a* `safe` *(or perhaps a* `washer`*) and a* `safe` *(or perhaps a* `loudspeaker`*).*

CHAPTER 13

More Neural Networks

13.1 LEARNING TO MAP

Imagine we have a high dimensional dataset. As usual, there are N d-dimensional points $\mathbf{x}$, where the i'th point is $\mathbf{x}_i$. We would like to build a map of this dataset, to try and visualize its major features. We would like to know, for example, whether it contains many or few blobs; whether there are many scattered points; and so on. We might also want to plot this map using different plotting symbols for different kinds of data points. For example, if the data consists of images, we might be interested in whether images of cats form blobs that are distinct from images of dogs, and so on. I will write $\mathbf{y}_i$ for the point in the map corresponding the $\mathbf{x}_i$. The map is an M dimensional space (though M is almost always two or three in applications).

We have seen one tool for this exercise (section 10.1). This used eigenvectors to identify a linear projection of the data that made low dimensional distances similar to high dimensional distances. I argued that the choice of map should minimize

$$\sum_{i,j} \left(\| \mathbf{y}_i - \mathbf{y}_j \|^2 - \| \mathbf{x}_i - \mathbf{x}_j \|^2 \right)^2$$

then rearranged terms to produce a solution that minimized

$$\sum_{i,j} \left(\mathbf{y}_i^T \mathbf{y}_j - \mathbf{x}_i^T \mathbf{x}_j \right)^2 .$$

The solution produces a $\mathbf{y}_i$ that is a linear function of $\mathbf{x}_i$, just as a by-product of the mathematics. There are two problems with this approach (apart from the fact that I suppressed a bunch of detail). If the data lies on a curved structure in the high dimensional space, then a linear projection can distort the map very badly. Figure **??**) sketches one example.

You should notice that the original choice of cost function is not a particularly good idea, because our choice of map is almost entirely determined by points that are very far apart. This happens because squared differences between big numbers tend to be a lot bigger than squared differences between small numbers, and so distances between points that are far apart will be the most important terms in the cost function. In turn, this could mean our map does not really show the structure of the data – for example, a small number of scattered points in the original data could break up clusters in the map (the points in clusters are pushed apart to get a map that places the scattered points in about the right place with respect to each other).

13.1.1 Sammon Mapping

Sammon mapping is a method to fix these problems by modifying the cost function. We attempt to make the small distances more significant in the solution by minimizing

$$C(\mathbf{y}_1, \ldots, \mathbf{y}_N) = \left(\frac{1}{\sum_{i<j} \| \mathbf{x}_i - \mathbf{x}_j \|} \right) \sum_{i<j} \left[\frac{(\| \mathbf{y}_i - \mathbf{y}_j \| - \| \mathbf{x}_i - \mathbf{x}_j \|)^2}{\| \mathbf{x}_i - \mathbf{x}_j \|} \right].$$

The first term is a constant that makes the gradient cleaner, but has no other effect. What is important is we are biasing the cost function to make the error in small distances much more significant. Unlike straightforward multidimensional scaling, the range of the sum matters here – if i equals j in the sum, then there will be a divide by zero.

No closed form solution is known for this cost function. Instead, choosing the $\mathbf{y}$ for each $\mathbf{x}$ is by gradient descent on the cost function. You should notice there is no unique solution here, because rotating, translating or reflecting all the $\mathbf{y}_i$ will not change the value of the cost function. Furthermore, there is no reason to believe that gradient descent necessarily produces the best value of the cost function. Experience has shown that Sammon mapping works rather well, but has one annoying feature. If one pair of high dimensional points is very much closer together than any other, then getting the mapping right for that pair of points is extremely important to obtain a low value of the cost function. This should seem like a problem to you, because a distortion in a very tiny distance should not be much more important than a distortion in a small distance.

13.1.2 T-SNE

We will now build a model by reasoning about probability rather than about distance (although this story could likely be told as a metric story, too). We will build a model of the probability that two points in the high dimensional space are neighbors, and another model of the probability that two points in the low dimensional space are neighbors. We will then adjust the locations of the points in the low dimensional space so that the KL divergence between these two models is small.

We reason first about the probability that points in the high dimensional space are neighbors. Write the conditional probability that $\mathbf{x}_j$ is a neighbor of $\mathbf{x}_i$ as $p_{j|i}$. Write

$$w_{j|i} = \exp\left(\frac{\| \mathbf{x}_j - \mathbf{x}_i \|^2}{2\sigma_i^2} \right)$$

We use the model

$$p_{j|i} = \frac{w_{j|i}}{\sum_k w_{k|i}}.$$

Notice this depends on the scale at point i, written σ_i. For the moment, we assume this is known. Now we define p_{ij} the joint probability that $\mathbf{x}_i$ and $\mathbf{x}_j$ are neighbors by assuming $p_{ii} = 0$, and for all other pairs

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}.$$

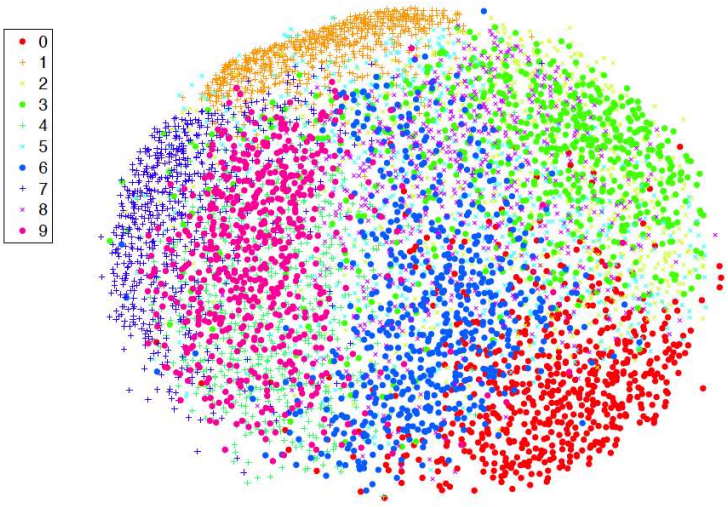


FIGURE 13.1: *A Sammon mapping of 6,000 samples of a 1,024 dimensional data set. The data was reduced to 30 dimensions using PCA, then subjected to a Sammon mapping. This data is a set of 6, 000 samples from the MNIST dataset, consisting of a collection of handwritten digits which are divided into 10 classes* $(0, \ldots 9)$*. The class labels were not used in training, but the plot shows class labels. This helps determine whether the visualization is any good – you could reasonably expect a visualization to put items in the same class close together and items in very different classes far apart. As the legend on the side shows, the classes are quite well separated. Figure from* Visualizing Data using t-SNE *Journal of Machine Learning Research 9 (2008) 2579-2605 Laurens van der Maaten and Geoffrey Hinton, to be replaced with a homemade figure in time.*

This is an $N \times N$ table of probabilities; you should check that this table represents a joint probability distribution (i.e. it's non-negative, and sums to one).

We use a slightly different probability model in the low dimensional space. We know that, in a high dimensional space, there is "more room" near a given point (think of this as a base point) than there is in a low dimensional space. This means that mapping a set of points from a high dimensional space to a low dimensional space is almost certain to move some points further away from the base point than we would like. In turn, this means there is a higher probability that a distant point in the low dimensional space is still a neighbor of the base point. Our probability model needs to have "long tails" – the probability that two points are neighbors should not fall off too quickly with distance. Write q_{ij} for the probability that $\mathbf{y}_i$ and $\mathbf{y}_j$ are neighbors. We assume that $q_{ii} = 0$ for all i. For other pairs, we use the model

$$q_{ij}(\mathbf{y}_1, \ldots, \mathbf{y}_N) = \frac{1/1+\|\mathbf{y}_i - \mathbf{y}_j\|^2}{\sum_{k,l,k \neq l} 1/1+\|\mathbf{y}_i - \mathbf{y}_k\|^2}$$

(where you might recognize the form of Student's t-distribution if you have seen

that before). You should think about the situation like this. We have a table representing the probabilities that two points in the high dimensional space are neighbors, from our model of p_{ij}. The values of the $\mathbf{y}$ can be used to fill in an $N \times N$ joint probability table representing the probabilities that two points are neighbors. We would like this tables to be like one another. A natural metric of similarity is the KL-divergence, of section 10.1. So we will choose $\mathbf{y}$ to minimize

$$C_{tsne}(\mathbf{y}_1, \ldots, \mathbf{y}_N) = \sum_{ij} p_{ij} \log \frac{p_{ij}}{q_{ij}(\mathbf{y}_1, \ldots, \mathbf{y}_N)}.$$

Remember that $p_{ii} = q_{ii} = 0$, so adopt the convention that $0 \log 0/0 = 0$ to avoid embarrassment (or, if you don't like that, omit the diagonal terms from the sum). Gradient descent with a fixed steplength and momentum was be sufficient to minimize this in the original papers, though likely the other tricks of section 10.1 might help.

There are two missing details. First, the gradient has a quite simple form (which I shall not derive). We have

$$\nabla_{\mathbf{y}_i} C_{tsne} = 4 \sum_j \left[(p_{ij} - q_{ij}) \frac{(\mathbf{y}_i - \mathbf{y}_j)}{1 + \| \mathbf{y}_i - \mathbf{y}_j \|^2} \right].$$

Second, we need to choose σ_i. There is one such parameter per data point, and we need them to compute the model of p_{ij}. This is usually done by search, but to understand the search, we need a new term. The **perplexity** of a probability distribution with entropy $H(P)$ is defined by

$$\text{Perp}(P) = 2^{H(P)}.$$

The search works as follows: the user chooses a value of perplexity; then, for each i, a binary search is used to choose σ_i such that $p_{j|i}$ has that perplexity. Experiments currently suggest that the results are quite robust to wide changes in the users choice.

In practical examples, it is quite usual to use PCA to get a somewhat reduced dimensional version of the $\mathbf{x}$. So, for example, one might reduce dimension from 1,024 to 30 with PCA, then apply t-SNE to the result.

13.2 ENCODERS, DECODERS AND AUTO-ENCODERS

An **encoder** is a network that can take a signal and produce a code. Typically, this code is a description of the signal. For us, signals have been images and I will continue to use images as examples, but you should be aware that all I will say can be applied to sound and other signals. The code might be "smaller" than the original signal – in the sense it contains fewer numbers – or it might even be "bigger" – it will have more numbers, a case referred to as an **overcomplete** representation. You should see our image classification networks as encoders. They take images and produce short representations. A **decoder** is a network that can take a code and produce a signal. We have not seen decoders to date.

An **auto-encoder** is a learned pair of coupled encoder and decoder; the encoder maps signals into codes, and the decoder reconstructs signals from those

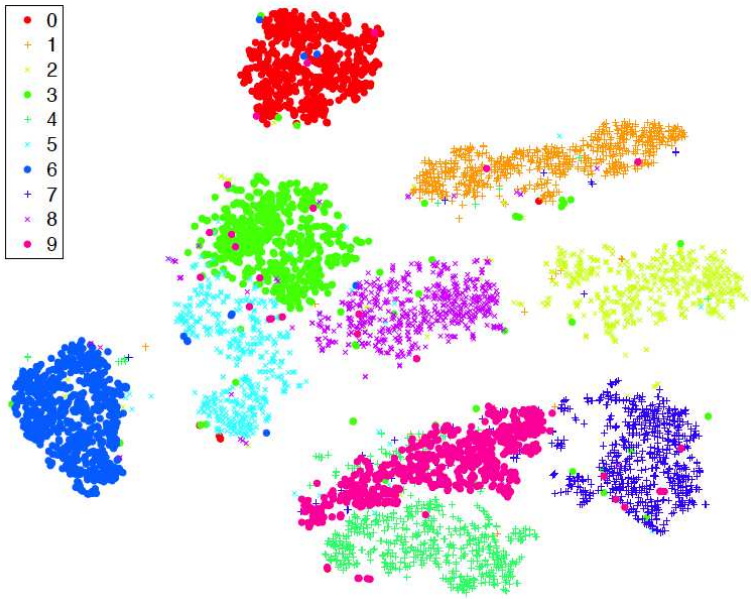


FIGURE 13.2: *A t-sne mapping of 6,000 samples of a 1,024 dimensional data set. The data was reduced to 30 dimensions using PCA, then subjected to a t-sne mapping. This data is a set of 6, 000 samples from the MNIST dataset, consisting of a collection of handwritten digits which are divided into 10 classes* $(0, \ldots 9)$*. The class labels were not used in training, but the plot shows class labels. This helps determine whether the visualization is any good – you could reasonably expect a visualization to put items in the same class close together and items in very different classes far apart. As the legend on the side shows, the classes are quite well separated. Figure from* Visualizing Data using t-SNE *Journal of Machine Learning Research 9 (2008) 2579-2605 Laurens van der Maaten and Geoffrey Hinton, to be replaced with a homemade figure in time.*

codes. Auto-encoders have great potential to be useful, which we will explore in the following sections. You should be aware that this potential has been around for some time, but has been largely unrealized in practice. One application is in unsupervised feature learning, where we try to construct a useful feature set from a set of unlabelled images. We could use the code produced by the auto-encoder as a source of features. Another possible use for an auto-encoder is to produce a clustering method – we use the auto-encoder codes to cluster the data. Yet another possible use for an auto-encoder is to generate images. Imagine we can train an auto-encoder so that (a) you can reconstruct the image from the codes and (b) the codes have a specific distribution. Then we could try to produce new images by feeding random samples from the code distribution into the decoder.

13.2.1 Auto-encoder Problems

Assume we wish to classify images, but have relatively few examples from each class. We can't use a deep network, and would likely use an SVM on some set of features, but we don't know what feature vectors to use. We could build an auto-encoder that produced an overcomplete representation, and use that overcomplete representation as a set of feature vectors. The decoder isn't of much interest, but we need to train with a decoder. The decoder ensures that the features actually describe the image (because you can reconstruct the image from the features). The big advantage of this approach is we could train the auto-encoder with a very large number of unlabelled images. We can then reasonably expect that, because the features describe the images in a quite general way, the SVM can find something discriminative in the set of features.

We will describe one procedure to produce an auto-encoder. The encoder is a layer that produces a code. For concreteness, we will discuss grey-level images, and assume the encoder is one convolutional layer. Write $\mathcal{I}_i$ for the i'th input image. All images will have dimension $m \times m \times 1$. We will assume that the encoder has r distinct units, and so produces a block of data that is $s \times s \times r$. Because there may be stride and convolution edge effects in the encoder, we may have that s is a lot smaller than m. Alternatively, we may have $s = m$. Write $\mathcal{E}(\mathcal{I}, \theta_e)$ for the encoder applied to image $\mathcal{I}$; here θ_e are the weights and biases of the units in the encoder. Write $Z_i = \mathcal{E}(\mathcal{I}_i, \theta_e)$ for the code produced by the encoder for the i'th image. The decoder must accept the output of the encoder and produce an $m \times m \times l$ image. Write $\mathcal{D}(Z, \theta_d)$ for the decoder applied to a code Z.

We have $Z_i = \mathcal{E}(\mathcal{I}_i, \theta_e)$, and would like to have $\mathcal{D}(Z_i, \theta_d)$ close to $\mathcal{I}_i$. We could enforce this by training the system, by stochastic gradient descent on θ_e, θ_d, to minimize $\|\mathcal{D}(Z_i, \theta_d) - \mathcal{I}_i\|^2$. One thing should worry you. If $s \times s \times r$ is larger than $m \times m$, then there is the possibility that the code is redundant in uninteresting ways. For example, if $s = m$, the encoder could consist of units that just pass on the input, and the decoder would pass on the input too – in this case, the code is the original image, and nothing of interest has happened.

13.2.2 The denoising auto-encoder

There is a clever trick to avoid this problem. We can require the codes to be robust, in the sense that if we feed a noisy image to the encoder, it will produce a code that recovers *the original image*. This means that we are requiring a code that not only describes the image, but is not disrupted by noise. Training an auto-encoder like this results in a **denoising auto-encoder**. Now the encoder and decoder can't just pass on the image, because the result would be the noisy image. Instead, the encoder has to try and produce a code that isn't affected (much) by noise, and the decoder has to take the possibility of noise into account while decoding.

Depending on the application, we could use one (or more) of a variety of different noise models. These impose slightly different requirements on the behavior of the encoder and decoder. There are three natural noise models: add independent samples of a normal random variable at each pixel (this is sometimes known as additive gaussian noise); take randomly selected pixels, and replace their values with 0 (masking noise); and take randomly selected pixels and replace their values

with a random choice of brightest or darkest value (salt and pepper noise).

In the context of images, it is natural to use the least-squares error as a loss for training the auto-encoder. I will write noise($\mathcal{I}_i$) to mean the result of applying noise to image I_i. We can write out the training loss for example i as

$$\|\mathcal{D}(Z_i, \theta_d) - \mathcal{I}_i\|^2 \text{ where } Z_i = \mathcal{E}(\text{noise}(\mathcal{I}_i), \theta_e)$$

You should notice that masking noise and salt and pepper noise are different to additive gaussian noise, because for masking noise and salt and pepper noise *only some pixels* are affected by noise. It is natural to weight the least-square error at these pixels *higher* in the reconstruction loss – when we do so, we are insisting that the encoder learn a representation that is really quite good at predicting missing pixels. Training is by stochastic gradient descent, using one of the gradient tricks of section 10.1. Note that each time we draw a training example, we construct a new instance of noise for that version of the training example, so the encoding and decoding layer may see the same example with different sets of pixels removed, etc.

13.2.3 Stacking Denoising Auto-encoders

An encoder that consists of a single convolutional layer likely will not produce a rich enough representation to do anything useful. After all, the output of each unit depends only on a small neighborhood of pixels. We would like to train a multi-layer encoder. Experimental evidence over many years suggests that just building a multi-layer encoder network, hooking it to a multi-layer decoder network, and proceeding to train with stochastic gradient descent just doesn't work well. It is tough to be crisp about the reasons, but the most likely problem seems to be that interactions between the layers make the problem wildly ambiguous. For example, each layer could act to undo much of what the previous layer has done.

Here is a strategy that works for several different types of auto-encoder (though I will describe it only in the context of a denoising auto-encoder). First, we build a single layer encoder $\mathcal{E}$ and decoder $\mathcal{D}$ using the denoising auto-encoder strategy to get parameters θ_{e1} and θ_{d1}. The number of units, stride, support of units, etc. are chosen by experiment. We train this auto-encoder to get an acceptable reconstruction loss in the face of noise, as above.

Now I can think of each block of data $Z_{i1} = \mathcal{E}(I_i, \theta_{e1})$ as being "like" an image; it's just $s \times s \times r$ rather than $m \times m \times 1$. Notice that $Z_{i1} = \mathcal{E}(I_i, \theta_{e1})$ is the output of the encoder on a real image (rather than a real image with noise). I could build *another* denoising auto-encoder that handles Z_1's. In particular, I will build single layer encoder $\mathcal{E}$ and decoder $\mathcal{D}$ using the denoising auto-encoder strategy to get parameters θ_{e2} and θ_{d2}. This encoder/decoder pair must auto-encode the objects produced by the first pair. So I fix θ_{e1}, θ_{d1}, and the loss for image i as a function of θ_{e2}, θ_{d2} becomes

$$\begin{aligned} \|\mathcal{D}(Z_{i2}, \theta_{d2}) - Z_{i1}\|^2 \quad & \text{where} \quad Z_{i2} = \mathcal{E}(\text{noise}(Z_{i1}), \theta_{e2}) \\ & \text{and} \quad Z_{i1} = \mathcal{E}(\mathcal{I}_1, \theta_{e1}) \end{aligned}$$

Again, training is by stochastic gradient descent using one of the tricks of section 10.1.

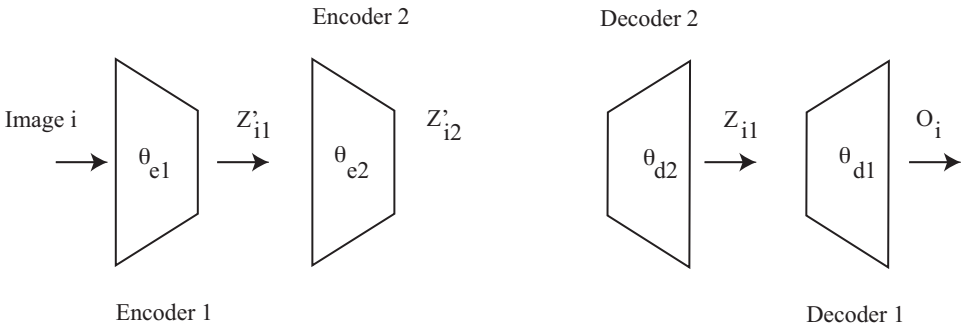


FIGURE 13.3: *Two layers of denoising auto-encoder, ready for fine tuning. This figure should help with the notation in the text.*

We can clearly apply this approach recursively, to stack train multiple layers. But more work is required to produce the best auto-encoder. In the two layer example, notice that the error does not take into account the effect of the first decoder on errors made by the second. We can fix this once all the layers have been trained *if* we need to use the result as an auto-encoder. This is sometimes referred to as *fine tuning*. We now train all the θ's. So, in the two layer case, the image passes into the first encoder, the result passes into the second encoder, then into the second decoder, then into the first decoder, and what emerges should be similar to the image. This gives a loss for image i in the two layer case as

$$\begin{aligned} \left\|\mathcal{D}(Z_{i1},\theta_{d1}) - I_i\right\|^2 \quad \text{where} \quad & Z_{i1} = \mathcal{D}((Z'_{i2}),\theta_{d2}) \\ \text{and} \quad & Z'_{i2} = \mathcal{E}(Z'_{i1},\theta_{e2}) \\ \text{and} \quad & Z'_{i1} = \mathcal{E}(\mathcal{I}_i,\theta_{e1}) \end{aligned}$$

(Figure 13.3 might be helpful here).

13.2.4 Classification using an Auto-encoder

It isn't usually the case that we want to use an auto-encoder as a compression device. Instead, it's a way to learn features that we hope will be useful for some other purpose. One important case occurs when we have little labelled image data. There aren't enough labels to learn a full convolutional neural network, but we could hope that using an auto-encoder would produce usable features. The process involves: fit an auto-encoder to a large set of likely relevant image data; now discard the decoders, and regard the encoder stack as something that produces features; pass the code produced by the last layer of the stack into a fully connected layer; and fine-tune the whole system using labelled training data. There is good evidence that denoising auto-encoders work rather well as a way of producing features, at least for MNIST data.

13.3 MAKING IMAGES FROM SCRATCH WITH VARIATIONAL AUTO-ENCODERS

*** This isn't right - need to explain why I would try to generate from scratch? *** we talk about himages here, but pretty much everything applies to other signals

too

13.3.1 Auto-Encoding and Latent Variable Models

There is a crucial, basic difficulty building a model to generate images. There is a lot of structure in an image. For most pixels, the colors nearby are about the same as the colors at that pixel. At some pixels, there are sharp changes in color. But these **edge points** are very highly organized spatially, too – they (largely) demarcate shapes. There is coherence at quite long spatial scales in images, too. For example, in an image of a donut sitting on a table, the color of the table inside the hole is about the same as the color outside. All this means that the overwhelming majority of arrays of numbers are not images. If you're suspicious, and not easily bored, draw samples from a multivariate normal distribution with unit covariance and see how long it will take before one of them even roughly looks like an image (hint: it won't happen in your lifetime, but looking at a few million samples is a fairly harmless way to spend time).

The structure in an image suggests a strategy. We could try to decode "short" codes to produce images. Write X for a random variable representing an image, and z for a code representing a compressed version of the image. Assume we can find a "good" model for $P(X|z, \theta)$. This might be built using a decoder network whose parameters are θ. Assume also that we can build codes and a decoder such that anything that comes out of the decoder looks like an image, *and* the probability distribution of codes corresponding to images is "easy". Then we could model $P(X)$ as

$$\int P(X|z, \theta)P(z)dz.$$

Such a model is known as a **latent variable model**. The codes z are **latent variables** – hidden values which, if known, would "explain" the image. In the first instance, assume we have a model of this form. Then generating an image would be simple in principle. We draw a sample from $P(z)$, then pass this through the network and regard the result as a sample from $P(X)$. This means that, for the model to be useful, we need to be able to actually draw these samples, and this constrains an appropriate choice of models. It is very natural to choose that $P(z)$ be a distribution that is easy to draw samples from. We will assume that $P(z)$ is a standard multivariate normal distribution (i.e. it has mean $\mathbf{0}$, and its covariance matrix is the identity). This is *by choice* – it's my model, and I made that choice.

However, we need to think very carefully about how to train such a model. One strategy might be to pass in samples from a normal distribution, then adjust the network parameters (by stochastic gradient descent, as always) to ensure what comes out is always an image. This isn't going to work, because it remains a remarkably difficult research problem to tell whether some array is an image or not. An alternative strategy is to build an encoder to make codes out of example images. We then train so that (a) the encoder produces codes that have a standard normal distribution and (b) the decoder takes the code computed from the i'th image and turns it into the i'th image. This isn't going to work either, because we're not taking account of the gaps between codes. We need to be sure that, if we present the decoder with *any* sample from a standard normal distribution (not

just the ones we've seen), it will give us an image.

The correct strategy is as follows. We train an encoder and a decoder. Write X_i for the i'th image, $\mathrm{E}(X_i) = z_i$ for the code produced by the decoder applied to X_i, $\mathrm{D}(z)$ for the image produced by the decoder on code z. For some image X_i, we produce $\mathrm{E}(X_i) = z_i$. We then obtain z *close to* z_i. Finally, we produce $\mathrm{D}(z)$. We train the encoder by requiring that the z "look like" IID samples from a standard normal distribution. We train the decoder by requiring that $\mathrm{D}(z)$ is close to X_i. Actually doing this will require some wading through probability, but the idea is quite clean.

13.3.2 Building a Model

Now, at least in principle, we could try to choose θ to maximize

$$\sum_i \log P(X_i|\theta).$$

But we have no way to evaluate the probability model, so this is hopeless. Recall the variational methods of chapters 10.1 and 10.1. Now choose some variational distribution $Q(z|X)$. This will have parameters, too, but I will suppress these and other parameters in the notation until we need to deal with them. Notice that

$$\begin{aligned}
\mathbb{D}(Q(z|X) \parallel P(z|X)) &= \mathbb{E}_Q[\log Q(z|X) - \log P(z|X)] \\
&= \mathbb{E}_Q[\log Q(z|X)] - \mathbb{E}_Q[\log P(X|z) + \log P(z) - \log P(X)] \\
&= \mathbb{E}_Q[\log Q(z|X)] - \mathbb{E}_Q[\log P(X|z) + \log P(z)] + \log P(X))
\end{aligned}$$

where the last line works because $\log P(X)$ doesn't depend on z. Recall the definition of the variational free energy from chapter 10.1. Write

$$\mathsf{F}_Q = \mathbb{E}_Q[\log Q] - \mathbb{E}_Q[\log P(X|z) + \log P(z)]$$

and so we have

$$\log P(X) - \mathbb{D}(Q(z|X) \parallel P(z|X)) = -\mathsf{F}_Q.$$

We would like to maximize $\log P(X)$ by choice of parameters, but we can't because we can't compute it. But we do know that $\mathbb{D}(Q(z|X) \parallel P(z|X)) \geq 0$. This means that $-\mathsf{F}_Q$ is a *lower* bound on $\log P(X)$. If we maximize this lower bound (equivalently, minimize the variational free energy), then we can reasonably hope that we have a large value of $\log P(X)$. The big advantage of this observation is that we *can* work with $-\mathsf{F}_Q$.

13.3.3 Turning the VFE into a Loss

The best case occurs when $Q(z|X) = P(z|X)$ (because then $\mathbb{D}(Q(z|X) \parallel P(z|X)) = 0$, and the lower bound is tight). We don't expect this to occur in practice, but it suggests a way of thinking about the problem. We can build our model of $Q(z|X)$ around an encoder that predicts a code from an image. Similarly, our model of $P(X|z)$ would be built around a decoder that predicts an image from a code.

We can simplify matters by rewriting the expression for the variational free energy. We have

$$\begin{aligned} -\mathsf{F}_Q &= -\mathbb{E}_Q[\log Q] + \mathbb{E}_Q[\log P(X|z) + \log P(z)] \\ &= \mathbb{E}_Q[\log P(X|z)] - \mathbb{D}(Q(z|X) \,\|\, P(z)). \end{aligned}$$

We want to build a model of $Q(z|X)$, which is a probability distribution, using a neural network. This model accepts an image, X, and needs to produce a *random* code z which depends on X. We will do this by using the network to predict the mean and covariance of a normal distribution, then drawing the code z from a normal distribution with that mean and covariance. I will write $\mu(X)$ for the mean and $\Sigma(X)$ for the covariance, where the (X) is there to remind you that these are functions of the input, and they are modelled by the neural network. We choose the covariance to be diagonal, because the code might be quite large and we do not wish to try and learn large covariance matrices.

Now consider the term $\mathbb{D}(Q(z|X) \,\|\, P(z))$. We get to choose the prior on the code, and we choose $P(z)$ to be a standard normal distribution (i.e. mean $\mathbf{0}$, covariance matrix the identity; I'll duck the question of the dimension of z for the moment). We can write

$$Q(z|X) = \mathcal{N}(\mu(X); \Sigma(X)).$$

We need to compute the KL-divergence between this distribution and a standard normal distribution. This can be done in closed form. For reference (if you don't feel like doing the integrals yourself, and can't look it up elsewhere), the KL-divergence between two multivariate normal distributions for k dimensional vectors is

$$\mathbb{D}(\mathcal{N}(\mu_0; \Sigma_0) \,\|\, \mathcal{N}(\mu_1; \Sigma_1)) = \frac{1}{2} \begin{pmatrix} \mathsf{Tr}\left(\Sigma_1^{-1}\Sigma_0\right) + (\mu_1 - \mu_0)^T \sigma_1^{-1} (\mu_1 - \mu_0) \\ -k + \log\left(\frac{\mathsf{Det}(\Sigma_1)}{\mathsf{Det}(\Sigma_0)}\right) \end{pmatrix}.$$

In turn, this means that

$$\mathbb{D}(\mathcal{N}(\mu(X); \Sigma(X)) \,\|\, \mathcal{N}(\mathbf{0}; \mathcal{I})) = \frac{1}{2} \begin{pmatrix} \mathsf{Tr}\left(\Sigma(X)\right) + \mu(X)^T \mu(X) \\ -k - \log\left(\mathsf{Det}\left(\Sigma\right)\right) \end{pmatrix}.$$

At this point, we are close to having an expression for a loss that we can actually minimize. We must deal with the term $\mathbb{E}_Q[\log P(X|z)]$. Recall that we modelled $Q(z|X)$ by drawing z from a normal distribution with mean $\mu(X)$ and covariance $\Sigma(X)$. We can obtain such a z by drawing from a standard normal distribution, then multiplying by $\Sigma(X)^{1/2}$ and adding back the mean $\mu(X)$. In equations, we have

$$\begin{aligned} \mathbf{u} &\sim \mathcal{N}(\mathbf{0}; \mathcal{I}) \\ z &= \mu(X) + \Sigma(X)^{1/2}\mathbf{u} \\ \log P(X|z) &= \log P(X|\mu(X) + \Sigma(X)^{1/2}\mathbf{u}). \end{aligned}$$

Our data X consists of a collection of images which we believe are IID samples from $P(X)$. I will write X_i for the i'th image. Originally, we wanted to choose

parameters to maximize

$$\begin{aligned}\log P(X) &= \sum_i \log P(X_i) \\ &= \mathbb{D}(Q(z|X) \parallel P(z|X)) - \mathsf{E}_{Q(z|X)} \\ &= \sum_i \left[\mathbb{D}(Q(z|X_i) \parallel P(z|X_i)) - \mathsf{E}_{Q(z|X_i)}\right].\end{aligned}$$

It's usual to train networks to minimize losses. We can write the loss as

$$\begin{aligned}\mathsf{E}_Q &= -\mathbb{E}_Q[\log Q] + \mathbb{E}_Q[\log P(X|z) + \log P(z)] \\ &= \mathbb{D}(Q(z|X) \parallel P(z)) - \mathbb{E}_Q[\log P(X|z)] \\ &= \sum_i \left[\mathbb{D}(Q(z|X_i) \parallel P(z)) - \mathbb{E}_{Q(z|X_i)}[\log P(X_i|z)]\right].\end{aligned}$$

I am now going to insert parameters. I will write parameters θ, with a subscript that tells you what the parameters are for. Recall we modelled Q with a network that took an image X_i and produced a mean $\mu(X_i;\theta_\mu)$ and a covariance $\Sigma(X_i;\theta_\Sigma)$. This network is an encoder - it makes codes (the means) from images. We will need a decoder to model $P(X|z)$. We will write $\mathrm{D}(z;\theta_D)$ for a network that produces an image from a code. We assume that images are given by $P(X|z) = \mathcal{N}(\mathrm{D}(z;\theta_D);\mathcal{I})$, so that

$$\log P(X_i|z) = \frac{-\left(\| X_i - \mathrm{D}(z;\theta_D)\|^2\right)}{2}.$$

So the loss becomes

$$\begin{aligned}\mathsf{E}_Q &= \sum_i \left[\mathbb{D}(Q(z|X_i) \parallel P(z)) - \mathbb{E}_{Q(z|X_i)}[\log P(X_i|z)]\right] \\ &= \sum_i \left[\begin{array}{c} \frac{1}{2}\left(\begin{array}{c} \mathsf{Tr}\left(\Sigma(X_i;\theta_\Sigma)\right) + \mu(X_i;\theta_\mu)^T\mu(X_i;\theta_\mu) \\ -k - \log\left(\mathsf{Det}\left(\Sigma(X_i;\theta_\Sigma)\right)\right) \end{array}\right) \\ -\mathbb{E}_{Q(z|X_i)}\left[\frac{-\left(\|X_i - \mathrm{D}(z;\theta_D)\|^2\right)}{2}\right] \end{array}\right].\end{aligned}$$

The expectation term is a nuisance. We will approximate the expectation by drawing one sample from $Q(z|X)$ and averaging over that one sample. Assume $\mathbf{u}_i$ is an IID sample of $\mathcal{N}(\mathbf{0};\mathcal{I})$. Then we write

$$\begin{aligned}\mathsf{E}_Q &= \sum_i \left[\mathbb{D}(Q(z|X_i) \parallel P(z)) - \mathbb{E}_{Q(z|X_i)}[\log P(X_i|z)]\right] \\ &\approx \sum_i \left[\begin{array}{c} \frac{1}{2}\left(\begin{array}{c} \mathsf{Tr}\left(\Sigma(X_i;\theta_\Sigma)\right) + \mu(X_i;\theta_\mu)^T\mu(X_i;\theta_\mu) - k \\ -\log\left(\mathsf{Det}\left(\Sigma(X_i;\theta_\Sigma)\right)\right) \end{array}\right) \\ -\frac{-\left(\|X_i - \mathrm{D}(\mu(X_i;\theta_\mu) + \Sigma(X_i;\theta_\Sigma)^{1/2}\mathbf{u}_i;\theta_D)\|^2\right)}{2} \end{array}\right].\end{aligned}$$

This is a loss, and it can be differentiated in θ_μ, θ_Σ, and θ_D. To train a variational auto-encoder, we use stochastic gradient descent with a variety of tricks on this loss.

13.3.4 Some Caveats

As of writing, variational auto-encoders are the cutting edge of generative models. They seem to be better at generating images than any other technology. However, they are interesting because a really strong generative model for images would be extremely useful, not because they're particularly good at generating images. There are a variety of important problems. Solutions to any, or all, of these problems would be very exciting, because it is extremely useful to be able to generate images.

Training: Variational auto-encoders are notoriously hard to train. There's a strong tendency to get no descent in the initial stages of training. The usual way to manage this is to weight the loss terms. You can break the loss into two terms. One measures the similarity of the code distribution to the normal distribution, the other measures the accuracy of reconstruction. Current practice weights the reconstruction loss very high in the early stages of training, then reduces that weight as training proceeds. This seems to help, for reasons I can't explain and have never seen explained.

Small images: Variational auto-encoders produce small images. Images bigger than 64×64 are tough to produce.

Mysterious code properties: There seems to be some limit to the complexity of the family of images that a variational auto-encoder can produce. This means that MNIST (for example) pretty much always works quite convincingly, but auto-encoding all the images in (say) ImageNet doesn't produce particularly good results. There is likely some relationship between the size of the code and the complexity of the family of images, but the effectiveness of training has something to do with it as well.

Blurry reconstructions: Variational auto-encoders produce blurry images. This is somewhat predictable from the loss and the training process. I know two arguments, neither completely rigorous. First, the image loss is L_2 error, which always produces blurry images because it regards a sharp edge in the wrong place as interchangeable with a slower edge in the right place. Second, the code distributions predicted by the encoder for two similar images must overlap; this means that the decoder is being trained to produce two distinct images for the same z, which must mean it averages and so loses detail.

Gaps in the code space: Codes are typically 32 dimensional. Expecting to produce a good estimate of an expectation with a single sample in a 32 dimensional space is a bit ambitious. This means that, in turn, there are many points in the code space that have never been explored by the encoder, or used in training the decoder. As a result, it is likely that a small search around a code can produce another code that generates a truly awful image. Of course, this result will only appear during an important live demo...

CHAPTER 14

Structured Models, Inference and Learning

14.1 DYNAMIC PROGRAMMING, REVISITED

One great feature of a hidden Markov model is that inference is straightforward. I showed how dynamic programming could be used both by reasoning about trellises and by reasoning about recursion. I demonstrated this in the context of a probabilistic model that factored in a particular way. We had

$$\begin{aligned} P(Y_1, Y_2, \ldots, Y_N, X_1, X_2, \ldots, X_N) &= P(X_1)P(Y_1|X_1) \\ &\quad P(X_2|X_1)P(Y_2|X_2) \\ &\quad \ldots \\ &\quad P(X_N|X_{N-1})P(Y_N|X_N). \end{aligned}$$

so that

$$\begin{aligned} \log P(Y_1, Y_2, \ldots, Y_N, X_1, X_2, \ldots, X_N) &= \log P(X_1) + \log P(Y_1|X_1) + \\ &\quad \log P(X_2|X_1) + \log P(Y_2|X_2) + \\ &\quad \ldots \\ &\quad \log P(X_N|X_{N-1}) + \log P(Y_N|X_N). \end{aligned}$$

For the trellis reasoning, I drew a trellis representing the sequence, then transferred the log probabilities as costs to the nodes and edges. We used dynamic programming to find the directed path through the trellis with the largest sum of costs. Nothing in this reasoning *required* the costs to actually be log probabilities. Remarkably, all that matters is the structure of the cost function.

14.1.1 Chain Graphs and Dynamic Programming

We now consider a situation where we will choose values for a set discrete variables, traditionally written X_i for $i = 1, \ldots, n$, to maximize the value of an objective function $f(X_1, \ldots, X_n)$. This objective function is a sum of **unary terms** (i.e., functions that take one argument), which we write $u_j(X_i)$, and **binary terms** (i.e., functions that take two arguments), which we write $b_k(X_i, X_j)$. This is a relatively general model of parts and relations. There is one score (which would be the u_k) associated with each part (which would be the X_i) and another associated with some of the relations (i.e., the b_k). For example, in the case of the HMM, the variables would be the hidden states, the unary terms would be the logs of emission probabilities, and binary terms would be the logs of transition probabilities. It is natural to think of the unary terms as nodes in a graph, and the binary terms as

edges. However, it is not required to think of the unary or binary terms as log probabilities; instead, you could think of them as negative energies (because we are maximizing), or you could minimize, and think of them as energies or costs. For this kind of model, maximization is straightforward if the graph we have described is a forest.

HMM's are a special case, because the graph in that case is a chain. We will redescribe inference for an HMM in this more general setting, because it will then follow easily that the method applies to a forest. We can write the objective function as

$$f_{\text{chain}}(X_1, \ldots, X_n) = \sum_{i=1}^{i=n} u_i(X_i) + \sum_{i=1}^{i=n-1} b_i(X_i, X_i + 1)$$

and we wish to maximize this function (you should check that the terms match terms in the expression for the joint for an HMM; a strategically placed logarithm will help). Now we define a new function, the **cost-to-go function**, with a recursive definition. Write

$$f^{(n-1)}_{\text{cost-to-go}}(X_{n-1}) = \max_{X_n} b_{n-1}(X_{n-1}, X_n) + u_n(X_n),$$

and notice that we have

$$\max_{X_1, \ldots, X_n} f_{\text{chain}}(X_1, \ldots, X_n)$$

is equal to

$$\max_{X_1, \ldots, X_{n-1}} \left(f_{\text{chain}}(X_1, \ldots, X_{n-1}) + f^{(n-1)}_{\text{cost-to-go}}(X_{n-1}) \right),$$

which means that we can eliminate the nth variable from the optimization by replacing the term $b_{n-1}(X_{n-1}, X_n) + u_n(X_n)$ with a function of X_{n-1}. This function is obtained by maximizing this term with respect to X_n. Equivalently, assume we must choose a value for X_{n-1}. The cost-to-go function tells us the value of $b_{n-1}(X_{n-1}, X_n) + u_n(X_n)$ obtained by making the best choice of X_n conditioned on our choice of X_{n-1}. Because any other choice would not lead to a maximum, if we know the cost-to-go function at X_{n-1}, we can now compute the best choice of X_{n-1} conditioned on our choice of X_{n-2}. This yields that

$$\max_{X_{n-1}, X_n} \left[b_{n-2}(X_{n-2}, X_{n-1}) + u_{n-1}(X_n - 1) + b_{n-1}(X_{n-1}, X_n) + u_n(X_n) \right]$$

is equal to

$$\max_{X_{n-1}} \left[b_{n-2}(X_{n-2}, X_{n-1}) + u_{n-1}(X_n - 1) + \left(\max_{X_n} b_{n-1}(X_{n-1}, X_n) + u_n(X_n) \right) \right].$$

But all this can go on recursively, yielding

$$f^{(k)}_{\text{cost-to-go}}(X_k) = \max_{X_{k+1}} b_k(X_k, X_{k+1}) + u_k(X_k) + f^{(k+1)}_{\text{cost-to-go}}(X_{k+1}).$$

We can expand this to describe our use of the trellis in Section 9.4.2. Notice that

$$\max_{X_1,\ldots,X_n} f_{\text{chain}}(X_1,\ldots,X_n)$$

is equal to

$$\max_{X_1,\ldots,X_{n-1}} \left(f_{\text{chain}}(X_1,\ldots,X_{n-1}) + f^{(n-1)}_{\text{cost-to-go}}(X_{n-1})\right)$$

which is equal to

$$\max_{X_1,\ldots,X_{n-2}} \left(f_{\text{chain}}(X_1,\ldots,X_{n-2}) + f^{(n-2)}_{\text{cost-to-go}}(X_{n-2})\right),$$

and we can apply the recursive definition of the cost-to-go function to get

$$\max_{X_1,\ldots,X_n} f_{\text{chain}}(X_1,\ldots,X_n) = \max_{X_1} \left(f_{\text{chain}}(X_1) + f^{1}_{\text{cost-to-go}}(X_1)\right),$$

which yields an extremely powerful maximization strategy. We start at X_n, and construct $f^{(n-1)}_{\text{cost-to-go}}(X_{n-1})$. We can represent this function as a table, giving the value of the cost-to-go function for each possible value of X_{n-1}. We build a second table giving the optimum X_n for each possible value of X_{n-1}. From this, we can build $f^{(n-2)}_{\text{cost-to-go}}(X_{n-2})$, again as a table, and also the best X_{n-1} as a function of X_{n-2}, again as a table, and so on. Now we arrive at X_1. We obtain the solution for X_1 by choosing the X_1 that yields the best value of $\left(f_{\text{chain}}(X_1) + f^{1}_{\text{cost-to-go}}(X_1)\right)$. But from this solution, we can obtain the solution for X_2 by looking in the table that gives the best X_2 as a function of X_1; and so on. It should be clear that this process yields a solution in polynomial time; in the exercises, you will show that, if each X_i can take one of k values, then the time is $O(nK^2)$.

This strategy will work for a model with the structure of a forest. The proof is an easy induction. If the forest has no edges (i.e., consists entirely of nodes), then it is obvious that a simple strategy applies (choose the best value for each X_i independently). This is clearly polynomial. Now assume that the algorithm yields a result in polynomial time for a forest with e edges, and show that it works for a forest with $e+1$ edges. There are two cases. The new edge could link two existing trees, in which case we could re-order the trees so the nodes that are linked are roots, construct a cost-to-go function for each root, and then choose the best pair of states for these roots from the cost-to-go functions. Otherwise, one tree had a new edge added, joining the tree to an isolated node. In this case, we reorder the tree so that this new node is the root and build a cost-to-go function from the leaves to the root. The fact that the algorithm works is a combinatorial insight, but many kinds of model have a tree structure. Models of this form are particularly important in cases of tracking and of parsing.

14.2 CONDITIONAL RANDOM FIELD MODELS FOR SEQUENCES

HMM models have been widely used, but have one odd feature that is inconsistent with practical experience. Recall X_i are the hidden variables, and Y_i are the

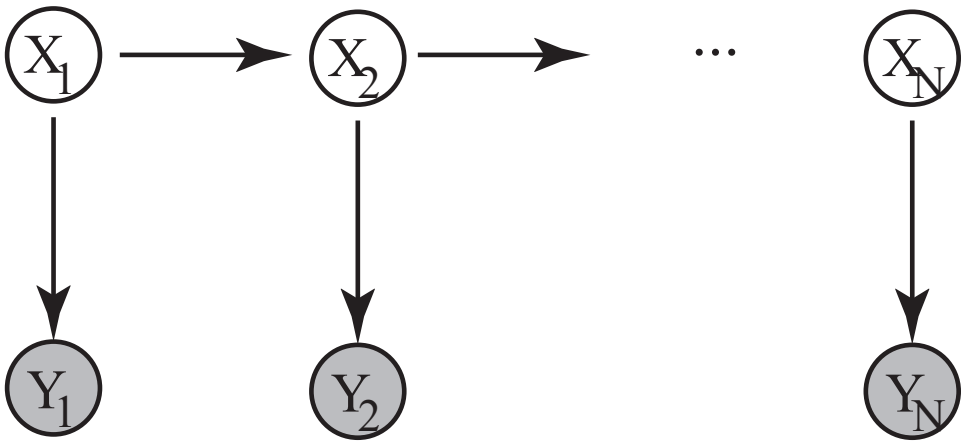


FIGURE 14.1:

observations. HMM's model

$$P(Y_1, \ldots, Y_n | X_1, \ldots, X_n) \propto P(Y_1, \ldots, Y_n, X_1, \ldots, X_n),$$

which is the probability of observations given the hidden variables. This is modelled using the factorization

$$\begin{aligned} P(Y_1, Y_2, \ldots, Y_N, X_1, X_2, \ldots, X_N) &= P(X_1)P(Y_1|X_1) \\ &\quad P(X_2|X_1)P(Y_2|X_2) \\ &\quad \ldots \\ &\quad P(X_N|X_{N-1})P(Y_N|X_N). \end{aligned}$$

In much of what we will do, this seems unnatural. For example, in the case of reading written text, we would be modelling the probability of the observed ink given the original text. But we would not attempt to find a single character by modelling the probability of the observed ink given the character (I will call this a **generative** strategy). Instead, we would search using a classifier, which is a model of the probability of the character conditioned on the observed ink (I will call this a **discriminative** strategy). The two strategies are quite different in practice. A generative strategy would need to explain all possible variants of the ink that a character could produce, but a discriminative strategy just needs to judge whether the ink observed is the character or not.

As another example, in the case of parsing people, we would be modelling the probability of the observed pixels given the configuration of the body. Again, this is an odd thing to do, because we may need quite sophisticated models to predict any set of pixels that could be produced by a body part. The alternative is to use a body part detector – a model of the probability the part is present, given the observed pixels. This reasoning applies to each of our examples.

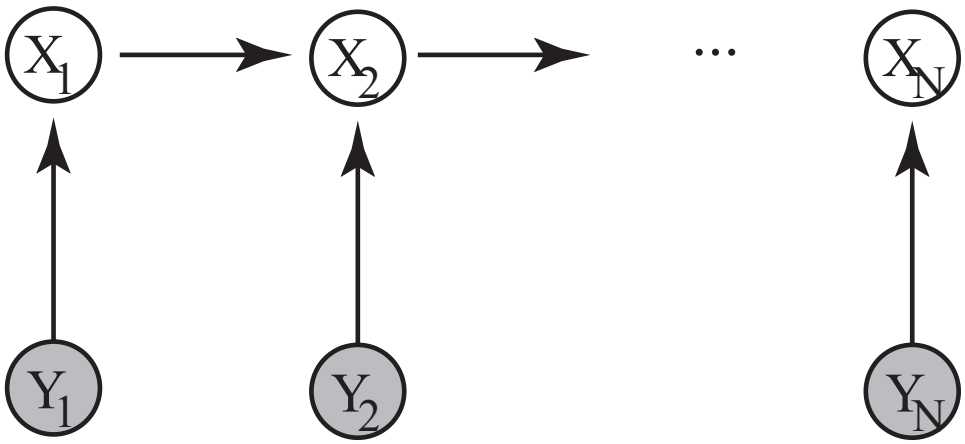


FIGURE 14.2:

14.2.1 Drawing a Model

We now adopt a convention that allows us to draw a model as a graph. Assume we have a probability distribution over a collection of variables. In the HMM example, these would be the hidden and the observed states in the HMM example. We will draw each variable as a circle. We will fill the circle if the variable's value is observed. We will add edges to this drawing according to the following rules. Assume we know a factorization of the probability distribution we are modelling into a collection of terms. There will be an edge between each pair of variables that appears together in a at least once in any factor. If two variables, say X_i and Y_i, only appear together in a factor in the form $P(Y_i|X_i)$, then the edge will be directed from X_i to Y_i (the arrow points toward the variable that is "generated").
TODO: FIX
These rules allow us to draw an HMM as Figure 14.1. This drawing corresponds to the factorization above.

14.2.2 MEMM's and Label Bias

An alternative that might seem plausible is to try and build a model with the structure suggested by Figure 14.2. This would correspond to

$$\begin{aligned} P(X_1, X_2, \ldots, X_N|Y_1, Y_2, \ldots, Y_N) &= P(X_1|Y_1) \times \\ &\quad P(X_2|Y_2, X_1) \times \\ &\quad P(X_3|X_2, Y_2) \times \\ &\quad \ldots \times \\ &\quad P(X_N|X_{N-1}, Y_N). \end{aligned}$$

Such models are known as **maximum entropy markov models** or **MEMM**'s. It turns out these models present practical difficulties, which are worth understanding. Assume we have fitted a model, and wish to recover the best sequence of X_i

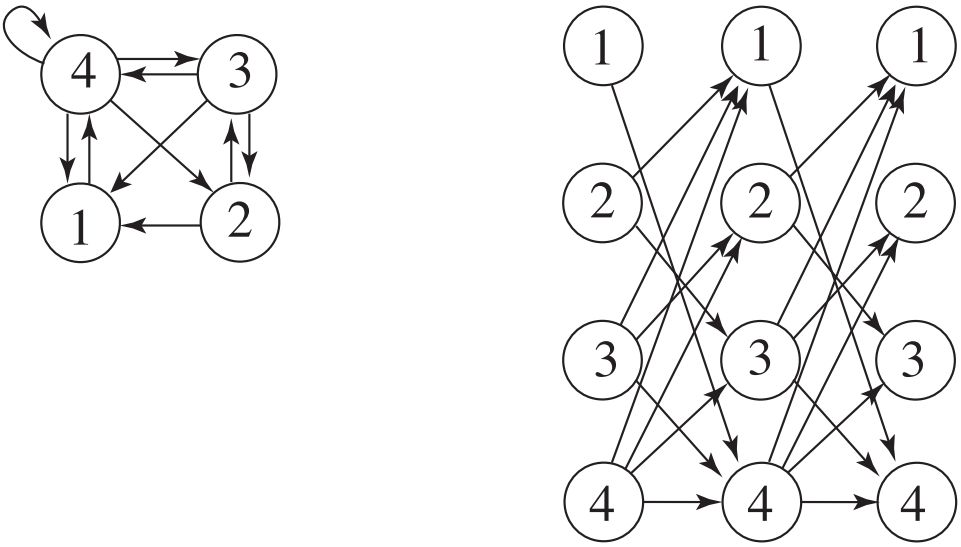


FIGURE 14.3:

corresponding to a given sequence of observations Y_i. We could minimize

$$\begin{aligned} -\log P(X_1, X_2, \ldots, X_N | Y_1, Y_2, \ldots, Y_N) \quad = \quad & -\log P(X_1|Y_1) \\ & -\log P(X_2|Y_2, X_1) \\ & -\log P(X_3|X_2, Y_2) \\ & \ldots \\ & -\log P(X_N|X_{N-1}, Y_N). \end{aligned}$$

by choice of $X_1, \ldots, X_N$. We can represent this cost function on a trellis, as for the HMM, but now notice that the costs on the trellis behave differently. For an HMM, each state (circle) in a trellis had a cost, corresponding to $-\log P(Y_i|X_i)$, and each edge had a cost $(-\log P(X_{i+1}|X_i))$, and the cost of a particular sequence was the sum of the costs along the implied path. But for an MEMM, the representation is slightly different. There is no term associated with each state in the trellis; instead, we associate the edge going from the state $X_i = U$ to the state $X_{i+1} = V$ with the cost $-\log P(X_{i+1} = V|X_i = U, Y_i)$. Again, the cost of a sequence of states is represented by the sum of costs along the corresponding path. This may look to you like a subtle change, but it has nasty effects.

Look at the example of Figure 14.3. Notice that when the model is in state 1, it can only transition to state 2. In turn, this means that $-\log P(X_{i+1} = 2|X_i = 1, Y_i) = 0$ *whatever* the measurement Y_i is. Furthermore, either $P(X_{i+1} = 3|X_i = 2, Y_i) \geq 0.5$ or $P(X_{i+1} = 1|X_i = 2, Y_i) \geq 0.5$ (because there are only two options leaving state 2). Here the measurement can determine which of the two options has higher probability. That figure shows a trellis corresponding to three measurements. In this trellis, the path 2 1 4 will be the lowest cost path *unless* the first measurement overwhelmingly disfavors the transition $2 \to 1$. This is because

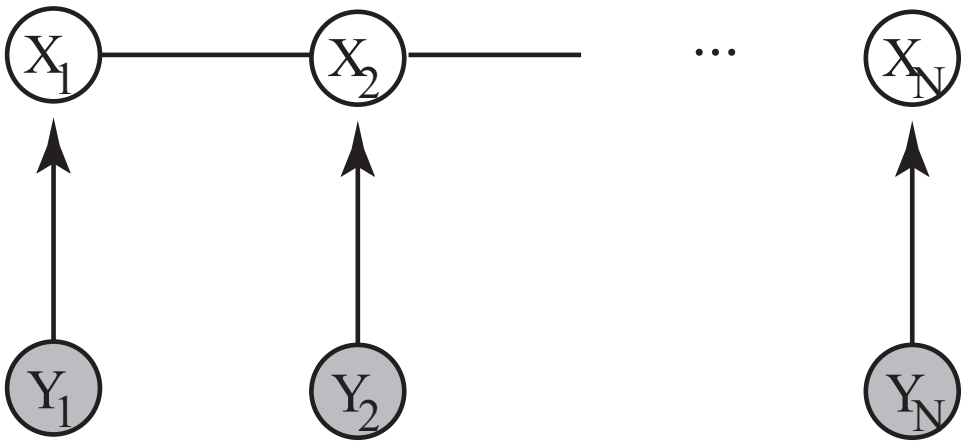


FIGURE 14.4:

most other paths must share weights between many outgoing edges; but $1 \rightarrow 4$ is very cheap, and $2 \rightarrow 1$ will be cheap unless there is an unusual measurement. Paths which pass through many states with few outgoing edges are strongly favored. This is known as the **label bias problem**. There are some fixes that can be applied, but it is better to reengineer the model.

14.2.3 Conditional Random Field Models

We want a model of sequences that is discriminative, but doesn't have the label bias problem. We'd also like that model to be as tractable as an HMM for sequences. Notice that what made an HMM easy to deal with was that we could solve for the best sequence of states using dynamic programming. We can use a trellis to represent the inference problem for any model where transitions between the hidden states are Markov. We can certainly use dynamic programming to solve for the best sequence of states if the cost model consists of per-edge and per-state costs (as it does in the case of the HMM). But to make the model discriminative, we need to change the probabilistic interpretation of these costs.

We start with the costs, written on the trellis. Write $E_i(a, b)$ for the cost of the edge from $X_i = a$ to $X_{i+1} = b$, and write $V_i(a)$ for the cost of assigning $X_i = a$. We will assume that $E_i(a, b) \geq 0$ and $V_i(a) \geq 0$. Note that for a useful model, we would like $V_i(a)$ to depend on Y_i. With appropriate rescaling, etc., we can interpret $V_i(a)$ as $-\log P(X_i = a|Y_i)$ (exercise: show that this doesn't affect the best path). We now apply a probabilistic interpretation to this model as

$$P(X_1 = x_1, X_2 = x_2, \ldots, X_N = x_n|Y_1, Y_2, \ldots, Y_N) = \frac{1}{K} \exp - \left[\begin{array}{l} V_1(x_1) + E_1(x_1, x_2) + \\ V_2(x_2) + E_2(x_2, x_3) + \\ \ldots \\ V_N(x_n) \end{array} \right],$$

where K is some normalizing constant, chosen to ensure the probability distribution sums to 1. There is a crucial difference with the MEMM; there are now node as

well as edge costs *but* we can't interpret the edge costs as transition probabilities. A model with this structure is known as a **conditional random field**. We can expand our graphical vocabulary to include undirected edges, corresponding to the $E_i(a, b)$ terms. The result is the model depicted in Figure 14.4; this figure is interpreted as yielding the probabilistic model above.

You should also be aware of the minus sign. This means that the best sequence has the smallest value of

$$\begin{bmatrix} V_1(x_1) + E_1(x_1, x_2) + \\ V_2(x_2) + E_2(x_2, x_3) + \\ \dots \\ V_N(x_n) \end{bmatrix},$$

and we can think of this expression as a cost. Inference is straightforward by dynamic programming, as above, if we have known E_i and V_i terms.

What is much more interesting is *learning* a CRF. We don't know K, and K depends on the E_i and V_i terms in a complicated way. There must be some set of parameters, θ that we are trying to adjust. However, we need some principle to drive the choice of θ. We can't simply collect some observations Y_i, then choose θ to maximize

$$P(Y_1, \dots, Y_N, |\theta)$$

because we don't know $P(Y_1, \dots, Y_N, |\theta)$. In fact, the reason we went to all this trouble was because it was easier to construct $P(X|Y)$ than $P(Y|X)$. We could try to maximize the posterior as a function of these parameters, but that means we need to know K. Notice that

$$K = \sum_{x_1, x_2, \dots, x_n} \exp - \begin{bmatrix} V_1(x_1) + E_1(x_1, x_2) + \\ V_2(x_2) + E_2(x_2, x_3) + \\ \dots \\ V_N(x_n) \end{bmatrix}$$

We need to be able to compute the sum. This isn't easy unless the model allows dynamic programming (replace the cost to go function with an appropriate sum of products), and even then is slow. But we can avoid doing this, by restating what we want to achieve.

14.3 DISCRIMINATIVE LEARNING OF CRFS

A really powerful strategy for learning a CRF follows from thinking about the model as representing an energy or cost. We write the cost of a sequence $x_1, x_2, \dots x_N$ as

$$\begin{aligned} C(x_1, x_2, \dots, x_N) &= V_1(x_1) + E_1(x_1, x_2) + V_2(x_2) + E_2(x_2, x_3) + \dots V_N(x_N) \\ &= V_1(x_1) + \sum_{i=1}^{n-1} [E_i(x_i, x_{i+1}) + V_{i+1}(x_{i+1})] \\ &= \sum_{i=1}^{N} V_i(x_i) + \sum_{i=1}^{N-1} E_i(x_i, x_{i+1}). \end{aligned}$$

Notice that the slightly funny form of the sum in the second line occurs because when there are N vertical arrows in Figure **??**, there are $N-1$ horizontal edges. The notation in the third line separates edge costs from node costs. Each notation appears on occasion. Notice that this cost is

$$-\log P(X_1 = x_1, X_2 = x_2, \ldots, X_N = x_n | Y_1, Y_2, \ldots, Y_N) - \log K.$$

What we will do is obtain both the observations and state for a set of example sequences. Different sequences in this set might have different lengths; this doesn't matter. Now write $Y_i^{(k)}$ for the i'th element of the k'th sequence of observations, etc. For any set of parameters, we can recover a solution from the observations using dynamic programming (write Inference$(Y_1, \ldots, Y_N^{(k)}, \theta)$ for this). Now we will choose a set of parameters $\hat{\theta}$ so that

$$\text{Inference}(Y_1^{(k)}, \ldots, Y_N^{(k)}, \hat{\theta}) \text{ is close to } X_1^{(k)} \ldots X_N^{(k)}.$$

In words, the principle is this. Choose parameters so that, if you infer a sequence of hidden states from a set of training observations, you will get the hidden states that are (about) the same as those observed.

14.3.1 Representing the Model

We need a parametric representation of V_i and E_i; we'll then search for the parameters that yield the right model. For the V, we will assume that V_i and V_j differ only if $Y_i \neq Y_j$. We then construct a set of functions $\phi_j^{(v)}(X, Y)$, and a vector of parameters $\theta_j^{(v)}$. Finally, we assert that $V_i(x) = \sum_j \theta_j^{(v)} \phi_j^{(v)}(x, Y_i)$. Similarly, for E_i we will construct a set of functions $\phi_j^{(v)}(U, V)$, and have $E_i(U, V) = \sum_j \theta_j^{(v)} \phi_j^{(v)}(U, V)$.

I give some sample constructions below, but you may find them somewhat involved at first glance. What is important is that (a) we have some parameters θ so that, for different choices of parameter, we get different cost functions; and (b) for any choice of parameters (say $\hat{\theta}$) and any sequence of Y_i we can label each vertex and each edge on the trellis with a number representing the cost of that vertex or edge. Assume we have these properties. Then for any particular $\hat{\theta}$ we can construct the best sequence of X_i using dynamic programming (as above). Furthermore, we can try to adjust the choice of $\hat{\theta}$ so that for the i'th training sequence, $\mathbf{y}^{(i)}$, inference yields $\mathbf{x}^{(i)}$ or something close by.

Building V: The choice of $\phi_j^{(v)}(X, Y)$ will naturally depend on the application. We can get some guidance by noticing we would like V to behave like a classifier — we'd like smaller values for $V(x, Y)$ when the x is a "good" interpretation of the observed state, and larger values when it isn't. For example, assume that the observed states are inked numerals (like MNIST), and that they appear in a window of fixed size. Multinomial logistic regression works quite well on MNIST using just the pixel values as features. This means that I can compute 10 linear functions of the pixel values (one for each numeral) such that the linear function corresponding to the right numeral is smaller than any other of the functions, at

least most of the time. Now for each possible true numeral x and each pixel location p we can build a feature function $\phi(X, Y) = \mathbb{I}_{[X=x]}\mathbb{I}_{[Y(p)=0]}$. This is 1 if $X = x$ (i.e. for a particular numeral, x) *and* the ink at pixel location p is dark, and otherwise zero. We index these feature functions in any way that seems convenient to get $\phi_j^{(v)}(X, Y)$, and stack them into a vector. Now $\sum_j \theta_j^{(v)}\phi_j^{(v)}(X, Y)$ is (a) linear in the ink for any fixed value of X and (b) different for different values of X and the same ink. With a good choice of $\theta^{(v)}$, we could produce a function that was small for the right numeral and large for the wrong numeral.

Another strategy would be to build the $\phi(X, Y)$ out of classifiers. For each class x, we will build several classifiers each of which can tell that class from all the others. We obtain different classifiers by using different training sets; or different features; or different classifier architectures; or all of these. Write $g_{i,x}(Y)$ for the i'th classifier for class x. We ensure that $g_{i,x}(Y)$ is small if Y is of class x, and large otherwise. Then for each classifier and each class we can build a feature function by $\phi(X, Y) = g_{i,X}(Y)$. We index these feature functions in any way that seems convenient. Now, with appropriate choice of $\theta_j^{(v)}$, we have that $\sum_j \theta_j^{(v)}\phi_j^{(v)}(X, Y)$ is a weighted sum of classifiers that produces a small value if X has the same class as Y and a large value otherwise.

Building E: We must now build $E_i(a, b)$. There are several possibilities. I will use U and V as dummy variables; each could take the value of any state. I will assume the states are labelled with counting numbers, without any loss of generality, and will write a, b for particular values of the state. One simple construction is to build a set of feature functions $\phi_{ab}^{(e)}(U, V) = \mathbb{I}_{[U=a]}\mathbb{I}_{[V=b]}$. There is one feature function for each possible pair of states, so if there are S possible states, there will be S^2 of these feature functions. Each one takes the value 1 when U and V take the corresponding state values, otherwise is 0. Now we construct a vector of S^2 parameters $\theta_{ab}^{(e)}$, and construct $E(U, V) = \sum_{a,b} \theta_{ab}^{(e)}\phi_{ab}^{(e)}(U, V)$. This construction will allow us to represent any possible cost for any transitions, as long as this doesn't depend on the observations.

We can build a set of feature functions that depend on the observed states, which I will write $\phi_{ab}^{(e)}(U, V; Y_i, Y_{i+1})$. The construction would depend on the application. For example, in the case of reading numerals, we could build a collection of classifiers that look at the ink corresponding to a pair of numerals together. Write $g_{j,ab}(Y_i, Y_{i+1})$ for these classifiers. Here j is an index that identifies the particular classifier. We require that $g_{j,ab}(Y_i, Y_{i+1})$ give a small response for examples where Y_i is the ink for numeral a and Y_{i+1} is the ink for numeral b; otherwise, the response should be large. Then we construct a vector of parameters $\theta_j^{(e)}$, and construct $E(U, V) = \sum_j \theta_j^{(e)} g_{j,UV}(Y_i, Y_{i+1})$. This yields a function that is small when the ink is consistent with the two states in the argument, and large otherwise.

General notation: We now have a model of the cost. I will write sequences like vectors, so $\mathbf{x}$ is a sequence, and x_i is the i'th element of that sequence. Write $C(\mathbf{x}; \mathbf{y}, \theta)$ for the cost of a sequence $\mathbf{x}$ of hidden variables, conditioned on observed values $\mathbf{y}$ and parameters θ. I'm suppressing the number of items in this sequence

for conciseness, but will use N if I need to represent it. We have

$$C(\mathbf{x};\mathbf{y},\theta) = \sum_{i=1}^{N}\left[\sum_{j}\theta_j^{(v)}\phi_j^{(v)}(x_i,y_i)\right] + \sum_{i=1}^{N-1}\left[\left(\sum_{l}\theta_l^{(e)}\phi_l^{(e)}(x_i,x_{i+1})\right)\right].$$

Notice that this cost function is linear in θ. We will use this to build a search for the best setting of θ.

14.3.2 Setting Up the Learning Problem

I will write $\mathbf{x}^{(i)}$ for the i'th training sequence of hidden states, and $\mathbf{y}^{(i)}$ for the i'th training sequence of observations. I will write $x_j^{(i)}$ for the hidden state at step j in the i'th training sequence, etc. The general principle we will adopt is that we should train a model by choosing θ such that, if we apply inference to $\mathbf{y}^{(i)}$, we will recover $\mathbf{x}^{(i)}$ (or something very similar).

For *any* sequence $\mathbf{x}$, we would like to have $C(\mathbf{x}^{(i)};\mathbf{y}^{(i)},\theta) \leq C(\mathbf{x};\mathbf{y}^{(i)},\theta)$. This inequality is much more general than it seems, because it covers *any* available sequence. Assume we engage in inference on the model represented by θ, using $\mathbf{y}^{(i)}$ as observed variables. Write $\mathbf{x}^{+,i}$ for the sequence recovered by inference, so that

$$\mathbf{x}^{+,i} = \underset{\mathbf{x}}{\operatorname{argmin}}\ C(\mathbf{x};\mathbf{y}^{(i)},\theta)$$

(i.e. $\mathbf{x}^{+,i}$ is the sequence recovered from the model by inference if the parameters take the value θ). In turn, the inequality means that

$$C(\mathbf{x}^{(i)};\mathbf{y}^{(i)},\theta) \leq C(\mathbf{x}^{+,i};\mathbf{y}^{(i)},\theta).$$

It turns out that this is not good enough; we would also like the cost of solutions that are further from the true solution to be higher. So we want to ensure that the cost of a solution grows at least as fast as its distance from the true solution. Write $d(\mathbf{u},\mathbf{v})$ for some appropriate distance between two sequences $\mathbf{u}$ and $\mathbf{v}$. We want to have

$$C(\mathbf{x}^{(i)};\mathbf{y}^{(i)},\theta) + d(\mathbf{x},\mathbf{x}^{(i)}) \leq C(\mathbf{x};\mathbf{y}^{(i)},\theta).$$

Again, we want this inequality to be true for *any* sequence $\mathbf{x}$. This means that

$$C(\mathbf{x}^{(i)};\mathbf{y}^{(i)},\theta) \leq C(\mathbf{x};\mathbf{y}^{(i)},\theta) - d(\mathbf{x},\mathbf{x}^{(i)})$$

for *any* $\mathbf{x}$. Now write

$$\mathbf{x}^{(*,i)} = \underset{\mathbf{x}}{\operatorname{argmin}}\ C(\mathbf{x};\mathbf{y}^{(i)},\theta) - d(\mathbf{x},\mathbf{x}^{(i)}).$$

The inequality becomes

$$C(\mathbf{x}^{(i)};\mathbf{y}^{(i)},\theta) \leq C(\mathbf{x}^{(*,i)};\mathbf{y}^{(i)},\theta) - d(\mathbf{x}^{(*,i)},\mathbf{x}^{(i)}).$$

This constraint is likely to be violated in practice. Assume that

$$\xi_i = \max(C(\mathbf{x}^{(i)};\mathbf{y}^{(i)},\theta) - C(\mathbf{x}^{(*,i)};\mathbf{y}^{(i)},\theta) + d(\mathbf{x}^{(*,i)},\mathbf{x}^{(i)}),0)$$

so that xi_i measures the extent to which the constraint is violated. We would like to choose θ so that we have the smallest possible set of constraint violations. It is natural to want to minimize the sum of ξ_i over all training data. But we also want to ensure that θ is not "too large", for the same reasons we regularized a support vector machine. Choose a regularization constant λ. Then we want to choose θ to minimize the regularized cost

$$\sum_{i \in \text{examples}} \xi_i + \lambda \theta^T \theta$$

where ξ_i is defined as above. This problem is considerably harder than it might look, because each ξ_i is a (rather strange) function of θ.

14.3.3 Evaluating the Gradient

We will solve the learning problem by stochastic gradient descent, as usual. First, we obtain an initial value of θ. Then we repeatedly choosing a minibatch of examples at random, evaluate the gradient for that minibatch, update the estimate of θ, and go again. There is the usual nuisance of choosing a steplength, etc. which is handled in the usual way. The important question is evaluating the gradient.

Imagine we have chosen the u'th example. We must evaluate $\nabla_\theta \xi_u$. Recall

$$\xi_u = \max(C(\mathbf{x}^{(u)}; \mathbf{y}^{(u)}, \theta) - C(\mathbf{x}^{(*,u)}; \mathbf{y}^{(u)}, \theta) + d(\mathbf{x}^{(*,u)}, \mathbf{x}^{(u)}), 0)$$

and assume that we know $\mathbf{x}^{(*,u)}$. We will ignore the concern that ξ_u may not be differentiable in θ as a result of the max. If $\xi_u = 0$, we will say the gradient is zero. For the other case, recall that

$$C(\mathbf{x}; \mathbf{y}, \theta) = \sum_{i=1}^{N} \left[\sum_j \theta_j^{(v)} \phi_j^{(v)}(x_i, y_i) \right] + \sum_{i=1}^{N-1} \left[\left(\sum_l \theta_l^{(e)} \phi_l^{(e)}(x_i, x_{i+1}) \right) \right]$$

and that this cost function is *linear* in θ. The distance term $d(\mathbf{x}^{(*,u)}, \mathbf{x}^{(u)})$ doesn't depend on θ, so doesn't contribute to the gradient. So if we *know* $\mathbf{x}^{*,i}$, the gradient is straightforward because C is linear in θ.

To be more explicit, we have

$$\frac{\partial C}{\partial \theta_j^{(v)}} = \sum_{i=1}^{N} \left[\phi_j^{(v)}(x_i^{(u)}, y_i^{(u)}) - \phi_j^{(v)}(x_i^{(*,u)}, y_i^{(u)}) \right]$$

and

$$\frac{\partial C}{\partial \theta_l^{(e)}} = \sum_{i=1}^{N-1} \left[\phi_l^{(e)}(x_i^{(u)}, x_{i+1}^{(u)}) - \phi_l^{(e)}(x_i^{(*,u)}, x_{i+1}^{(*,u)}) \right].$$

The problem is that we don't know $\mathbf{x}^{(*,u)}$ because it could change each time we change θ. Recall

$$\mathbf{x}^{(*,u)} = \operatorname*{argmin}_{\mathbf{x}} C(\mathbf{x}; \mathbf{y}^{(u)}, \theta) - d(\mathbf{x}, \mathbf{x}^{(u)}).$$

So, to compute the gradient, we must first run an inference on the example to obtain $\mathbf{x}^{(*,u)}$. But this inference could be hard, depending on the form of

$$C(\mathbf{x}; \mathbf{y}^{(u)}, \theta) - d(\mathbf{x}, \mathbf{x}^{(u)})$$

(which is often known as the **loss augmented constraint violation**). We would like to choose $d(\mathbf{x}, \mathbf{x}^{(u)})$ so that we get a distance that doesn't make the inference harder. One good, widely used example is the **Hamming distance**.

The Hamming distance between two sequences is the number of locations in which they disagree. Write $\text{diff}(m, n) = 1 - \mathbb{I}_{[m=n]}(m, n)$ for a function that returns zero if its arguments are the same, and one otherwise. Then we can express the Hamming distance as

$$d_h(\mathbf{x}, \mathbf{x}^{(u)}) = \sum_k \text{diff}(x_k, x_k^{(u)}).$$

We could scale the Hamming distance, to express how quickly we expect the cost to grow. So we will choose a non-negative number ϵ, and write

$$d(\mathbf{x}, \mathbf{x}^{(u)}) = \epsilon d_h(\mathbf{x}, \mathbf{x}^{(u)}).$$

The expression for Hamming distance is useful, because it allows us to represent the distance term on a trellis. In particular, think about the trellis corresponding to the u'th example. Then to represent the cost

$$C(\mathbf{x}; \mathbf{y}^{(u)}, \theta) - d(\mathbf{x}, \mathbf{x}^{(u)})$$

we adjust the node costs on each column. For the k'th column, we subtract ϵ from each of the node costs *except* the one corresponding to the k'th term in $\mathbf{x}^{(u)}$. Then the sum of edge and node terms along any path will correspond to $C(\mathbf{x}; \mathbf{y}^{(u)}, \theta) - d(\mathbf{x}, \mathbf{x}^{(u)})$. In turn, this means we can construct $\mathbf{x}^{(*,u)}$ by dynamic programming to this offset trellis.

Now we can compute the gradient for any example, so learning is (conceptually) straightforward. In practice, computing the gradient at any example involves finding the best sequence predicted by the loss augmented constraint violation, then using this to compute the gradient. Every gradient evaluation involves a round of inference, making the method slow.

CHAPTER 15

Boosting

The following idea may have occurred to you after reading the chapter on classification. Imagine you have trained a classifier - you could try and train a second classifier to fix errors made by the first. There doesn't seem to be any reason to stop there, and you might try and train a third classifier to fix errors made by the first and the second, and so on. It turns out this idea is fruitful, and (once all the details have been filled in) is usually known as **boosting**.

The details take some work, as you would expect. It isn't enough to just fix errors – you need some procedure to decide what the overall prediction of the system of classifiers is, and you need some way to be confident that the overall prediction will be better than the prediction produced by the initial classifier.

The idea may also have occurred to you after reading the chapter on regression. Imagine you have a regression that makes errors - you could try and produce a second regression that fixes those errors. You may have dismissed this idea, though, because if one uses only linear regressions trained using least squares, it's hard to see how to build a second regression that fixes the first regression's errors.

It turns out that one can boost a regressor as well as a classifier. The key idea is now a **predictor** – a function that accepts features and produces predictions. A classifier is a predictor that produces labels. A regressor is a predictor that produces numbers (or, sometimes, more complicated objects like vectors or trees, though we haven't talked about that much). This predictor is trained using a loss.

Now assume we would like to produce an optimal predictor. We could see this optimal predictor as a sum of less ambitious predictors, obtained using a form of gradient descent. Doing this cleanly will take some work, but the framework makes it possible to boost a very wide range of classifiers and regressors. Boosting is particularly attractive when one has a classifier (resp. regressor) that is simple and easy to train; one can often produce a boosted classifier (resp. regressor) that can be evaluated very fast.

15.1 GREEDY AND STAGEWISE METHODS

15.1.1 Greedy Stagewise Regression

We wish to build a regression of y against some high dimensional vector $\mathbf{x}$, and we know that a linear regression won't work. We might consider using bump functions, as in chapter **??**. But this turns out to work poorly in high dimension, because relying on a "nearby" data point to help becomes increasingly unwise as the dimension gets larger. The curse of dimension means there isn't a nearby point – all our data points will tend to be far from all the others.

A regression tree is a natural solution. We have not used regression trees before, but they are straightforward regression models. One builds a tree by splitting on coordinates, so each leaf represents a cell in space where the coordinates satisfy

some inequalities. At each leaf, there is a single value representing the value the predictor takes in that cell.

Here is a strategy for using regression trees (and other regression predictors) that is often successful. Assume we have a regression procedure that doesn't produce a global minimum in a single step (I will explain shortly why this matters). Then we could regress y against $\mathbf{x}$; construct the residuals; and regress the residuals against $\mathbf{x}$. Of course, this could be repeated, perhaps indefinitely. We will use set of predictors $f(\mathbf{x};\theta_k)$ (where θ_k are parameters internal to the predictor). Our model is

$$F(\mathbf{x}) = \sum_k a_k f(\mathbf{x};\theta_k)$$

where there might be quite a lot of functions indexed by k. Now we must fit this regression model to the data. We could fit the model by minimizing

$$\sum_i (y_i - F(\mathbf{x}_i))^2$$

as a function of the a's and the θ's. This is unattractive, because we may need to solve a very large minimization problem.

Here is an alternative strategy for fitting a model. Start with a $F_0 = 0$. Write $r_i^{(n)}$ for the residual at the n'th round and the i'th example (so that $r_i^{(0)} = y_i$). Now iterate the following steps: Choose a_n and θ_n to minimize

$$\sum_i \left(r_i^{(n)} - a_n f(\mathbf{x}_i;\theta_n) \right)^2 = \sum_i \left(\left[y_i - \sum_{j=1}^{n-1} a_j f(\mathbf{x};\theta_j) \right] - a_n f(\mathbf{x}_i;\theta_n) \right)^2 .$$

This is sometimes referred to as **greedy stagewise regression**. Notice that there is no particular reason to stop, unless (a) the residual is zero at all data points or (b) for some reason, it is clear that no future progress would be possible.

Worked example 15.1 *Greedy stagewise regression for prawns*

Construct a stagewise regression of score 1 against latitude and longitude, using the prawn trawls dataset from http://www.statsci.org/data/oz/reef.html. Use a regression tree.

Solution: There are good packages for building such trees (I used R's `rpart`). Stagewise regression is straightforward. I started with a current prediction of zero. Then I iterated: form the current residual (score 1 - current prediction); regress that against latitude and longitude; then update the current residual. Figure **??** shows the result. For this example, I used a function of two dimensions so I could plot the regression function in a straightforward way. It's easy to visualize a regression tree in 2D. The root node of the tree splits the plane in half, usually with an axis aligned line. Then each node splits its parent into two pieces, so each leaf is a rectangular cell on the plane (which might stretch to infinity). The value is constant in each leaf. You can't make a smooth predictor out of such trees, but the regressions are quite good (Figure **??**).

This process won't work for a linear regression, except in special cases. It's worth understanding why. Consider the first step. We will choose β to minimize $(\mathbf{y} - \mathcal{X}\beta)^T(\mathbf{y} - \mathcal{X}\beta)$. But there's a closed form solution for this β (which is $\hat{\beta} = (\mathcal{X}^T\mathcal{X})^{-1}\mathcal{X}^T\mathbf{y}$; remind yourself if you've forgotten by referring to chapter 10.1), and this is a global minimizer. So to minimize

$$(\left[\mathbf{y} - \mathcal{X}\hat{\beta}\right] - \mathcal{X}\gamma)^T(\left[\mathbf{y} - \mathcal{X}\hat{\beta}\right] - \mathcal{X}\gamma)$$

by choice of γ, we'd have to have $\mathcal{X}\gamma = 0$. In linear regression, we could benefit from a stagewise method if the number of features is very large. We randomly choose a subset of features, perform linear regression on those, then proceed with another random subset of the features. This is just a method of getting otherwise unruly linear algebra under control.

Worked example 15.2 *Predicting the quality of education of a university*

You can find a dataset of measures of universities at https://www.kaggle.com/mylesoneill/world-university-rankings/data. These measures are used to predict rankings. From these measures, but not using the rank or the name of the university, predict the quality of education using a stagewise regression. Use a regression tree.

Solution: Ranking universities is a fertile source of light entertainment for assorted politicians, bureaucrats, and journalists. I have no idea what any of the numbers in this dataset mean (and I suspect I may not be the only one). Anyhow, one could get some sense of how reasonable they are by trying to predict the quality of education score from the others. I used R's `rpart`, and the strategy in the preceeding example. More interesting is interpreting the results (Figure 15.2).

Worked example 15.3 *How long has a prescriber been in the profession?*

You can find a dataset of prescriber behavior at https://www.kaggle.com/roamresearch/prescriptionbasedprediction. This collects a variety of data on prescriber behavior. Predict the years in practice from all other variables except "specialty", using a stagewise regression with a regression tree as a predictor

Solution: This is pretty hard work. You have to read a JSON file (I read the first 10,000 lines for the figures), clean up the NA entries, which appear to mean the prescriber didn't prescribe that drug, and (at least if you're using `rpart`) clean up the variable names, too (otherwise they contain forbidden characters that annoy the regression tree function). There are a very large number of variables. I constructed predictors by selecting a small subset of independent variables at random, then building a regression tree for the residuals against just this set. You'll notice the number of years in the profession isn't evenly distributed, either. However, a stagewise regression with a regression tree produces quite good results (Figure 15.5; Figure **??**).

15.2 BOOSTING A CLASSIFIER

15.2.1 Recipe: Stagewise Reduction of Loss

Most of our recipes for a two class classifier can be coerced into this form. We have some function $F(\mathbf{x};\theta)$ (where $\mathbf{x}$ are the features, and θ are the parameters). We classify example $\mathbf{x}$ by evaluating this function, and testing it against a threshold. We arrange things so that the predicted label is $\mathsf{sign}\left(F(\mathbf{x};\theta)\right)$ (add a constant to F, and possibly multiply by -1). The classifier is learned by choosing a value $\hat{\theta}$ for θ that gets good behavior on a training set.

We evaluated the behavior on a training set with a loss function. For example, for an SVM we minimized the hinge loss (typically, regularized with a penalty on the size of the coefficients). Write $\mathcal{L}$ for the loss function, which depends both on predictions and the true labels. Then (for the moment ignoring the question of regularization), we obtained a classifier by minimizing

$$\mathbb{E}_{\mathcal{L}(F(\mathbf{x};\theta),y)}[P(y,x)] \approx \frac{1}{N}\sum_i \mathcal{L}(F(\mathbf{x}_i;\theta), y_i).$$

Now for many predictors F we are unlikely to be able to minimize this successfully in θ. It is natural to want to fit a second predictor to the errors made by the first. Here is a strategy.

Assume we wish to produce $F_M(\mathbf{x};\theta) = \sum_{j=1}^{M} a_j f_j(\mathbf{x};\theta_j)$. This is a sum of predictors, as in the stagewise regression case. Assume we have some F_{r-1}, and want to compute a new predictor that improves it. We need to minimize

$$\frac{1}{N}\sum_i \mathcal{L}(F_{r-1}(\mathbf{x}_i) + a_r f_r(\mathbf{x}_i;\theta_r), y_i)$$

but a Taylor series gives us

$$\begin{aligned}\frac{1}{N}\sum_i \mathcal{L}(F_{r-1}(\mathbf{x}_i) + a_r f_r(\mathbf{x}_i;\theta_r), y_i) &\approx \frac{1}{N}\sum_i \mathcal{L}(F_{r-1}(\mathbf{x}_i), y_i) + a_r \frac{\partial \mathcal{L}}{\partial F} f_r(\mathbf{x}_i;\theta_r) \\ &= \frac{1}{N}\sum_i \mathcal{L}(F_{r-1}(\mathbf{x}_i), y_i) + a_r \frac{1}{N}\sum_i \left[\frac{\partial \mathcal{L}}{\partial F} f_r(\mathbf{x}_i;\theta_r)\right]\end{aligned}$$

where the partial derivative is evaluated at the current predictor F_{r-1} at each data point. The proper strategy is to minimize

$$\sum_i \frac{\partial \mathcal{L}}{\partial F} f_r(\mathbf{x}_i;\theta_r)$$

by choice of f_r, then choose a value of a_r. We assume that f_r is bounded. Notice that

$$\sum_i (\frac{\partial \mathcal{L}}{\partial F} - f_r(\mathbf{x}_i;\theta_r))^2 = \sum_i \left[(\frac{\partial \mathcal{L}}{\partial F}^2) + (f_r(\mathbf{x}_i;\theta_r))^2 - 2(\frac{\partial \mathcal{L}}{\partial F} f_r(\mathbf{x}_i;\theta_r))\right]$$

] so the way to minimize $\sum_i \frac{\partial \mathcal{L}}{\partial F} f_r(\mathbf{x}_i;\theta_r)$ is to obtain an f_r that matches the values of $\frac{\partial \mathcal{L}}{\partial F}$ as closely as possible at each data point.

Now assume we have a solution $\hat{\theta}_r = \theta_r$. Then we can obtain a_r by minimizing

$$g(a_r) = \frac{1}{N}\sum_i \mathcal{L}(F_{r-1}(\mathbf{x}_i) + a_r f_r(\mathbf{x}_i;\hat{\theta}_r), y_i)$$

which is a one-dimensional problem. It is natural to use a line search method from an optimization package (or just minimize this function with an optimization package). This recipe, which is extremely general, is known as **gradient boost**.

15.2.2 Worked Example: Gradient Boost and Exponential Loss

One can apply gradient boost to any loss that appears convenient. However, there is a strong tradition of using the **exponential loss**. Write y_i for the true label for the i'th example (which is either 1 or -1). Then the exponential loss is

$$\mathcal{L}_e(F(\mathbf{x}_i), y_i) = e^{[-y_i F(\mathbf{x}_i)]}.$$

Notice if $F(\mathbf{x}_i)$ has the right sign, the loss is small; if it has the wrong sign, the loss is large.

To apply gradient boost, assume we know F_{r-1}, and seek a_r and f_r. There are two cases, but they work the same way. In the first case, f_r can report only 1 or -1. For example, f_r might be a classification tree which does not report vote counts. In the second case, f_r can report a continuous number, where the magnitude of the number gets larger as the predictor becomes more confident. For example, f_r might be logistic regression, or a tree that does report vote counts. In either case, we must choose the parameters of f_r to minimize

$$\sum_i (\frac{\partial \mathcal{L}_\rceil(\mathcal{F}(\mathbf{x}_\rangle), \dagger_\rangle)}{\partial F} - f_r(\mathbf{x}_i; \theta_r))^2 \quad = \quad \sum_i (\left[-y_i e^{[-y_i F(\mathbf{x}_i)]}\right] - f_r(\mathbf{x}_i; \theta_r))^2$$

Here the partial derivative is evaluated How we do this depends somewhat on the predictor, and on the form of the software that estimates a predictor. If f_r can report only 1 or -1, it is natural to try and maximize

$$\sum_i (\left[-y_i e^{[-y_i F(\mathbf{x}_i)]}\right] f_r(\mathbf{x}_i; \theta_r))$$

*** Exponential loss

Worked example 15.4 *Is a prescriber an opiate prescriber?*

You can find a dataset of prescriber behavior focusing on opiate prescriptions at https://www.kaggle.com/apryor6/us-opiate-prescriptions. One column of this data is a 0-1 answer, giving whether the individual prescribed opiate drugs more than 10 times in the year. Predict this column from the other entries, using a boosted decision tree and the exponential loss function. You should drop 'NPI', 'Gender', 'State', 'Credentials','Specialty')

Solution: Confusingly, the column is named "Opioid.Prescriber" but all the pages, etc. use the term "opiate"; the internet suggests that "opiates" come from opium, and "opioids" are semi-synthetic or synthetic materials that bind to the same receptors. Quite a lot of money rides on soothing the anxieties of internet readers about these substances, so I'm inclined to assume that easily available information is unreliable; for us, they will mean the same thing.

15.3 EXAMPLE: USING THE LASSO ON BOOSTED PREDICTORS

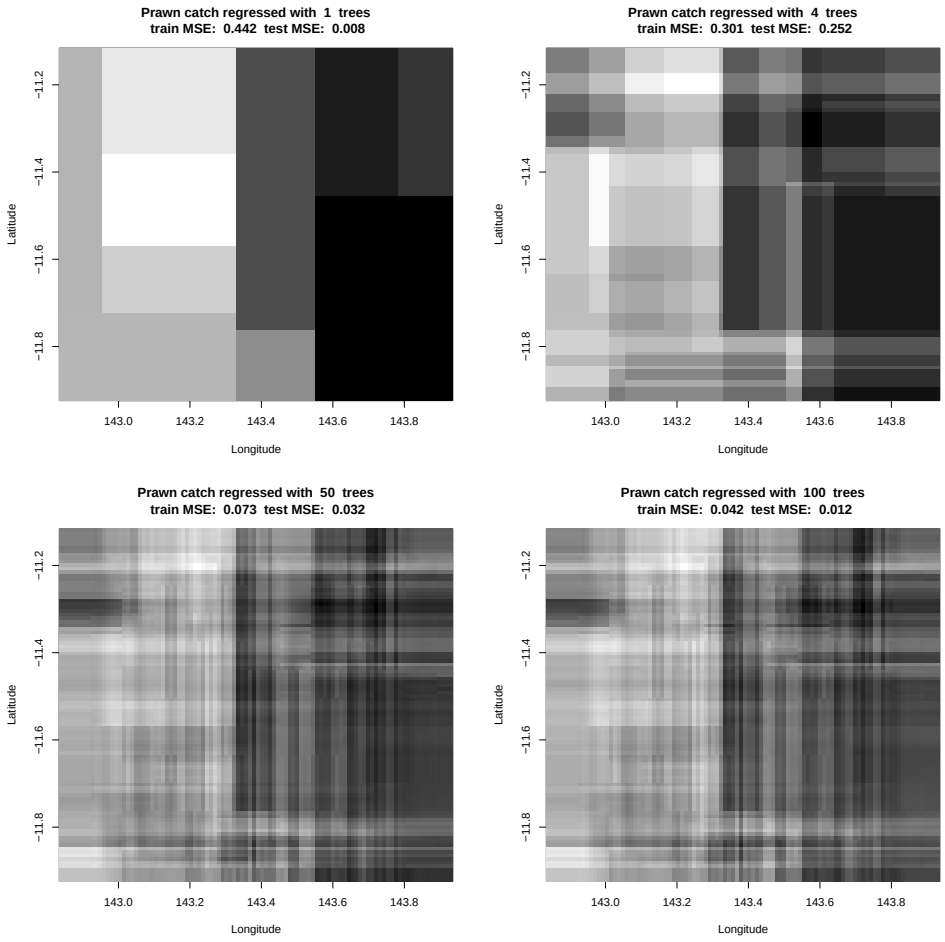


FIGURE 15.1: *Score 1 of the prawn trawls data from* http://www.statsci.org/data/oz/reef.html*, regressed against latitude and longitude (I did not use depth, also in that dataset). Four iterations of greedy stagewise regression using regression trees. Notice that both train and test error go down, and the model gets more complex as we add trees.*

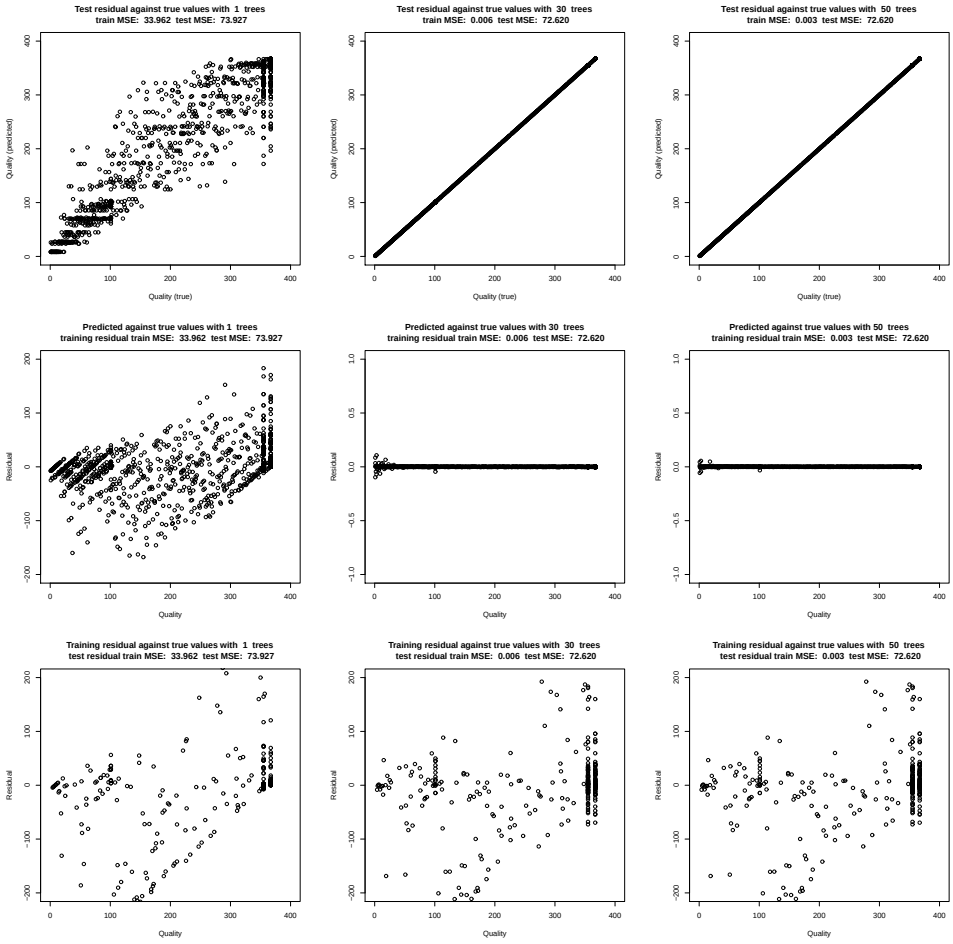


FIGURE 15.2: *Models from a stagewise regression of rank of quality of education against all other variables except overall rank and name for the data of* https://www.kaggle.com/mylesoneill/world-university-rankings/data. **Top row:** *Model predictions against true values;* **middle row:** *residual against true values, training examples;* **bottom row:** *residuals against true values, test examples. Notice how the residual from the model with one predictor is really quite structured; but once there are more predictors, the residual is more noise-like. Both test and training residuals go down with more predictors, but there is a big gap. Notice how universities with strong quality of education (i.e. low rank) are fairly easy to predict, but for universities with weaker quality (i.e. further down the pool) the predictions are poor. It would be interesting to see whether the regression was improved by incorporating the total rank.*

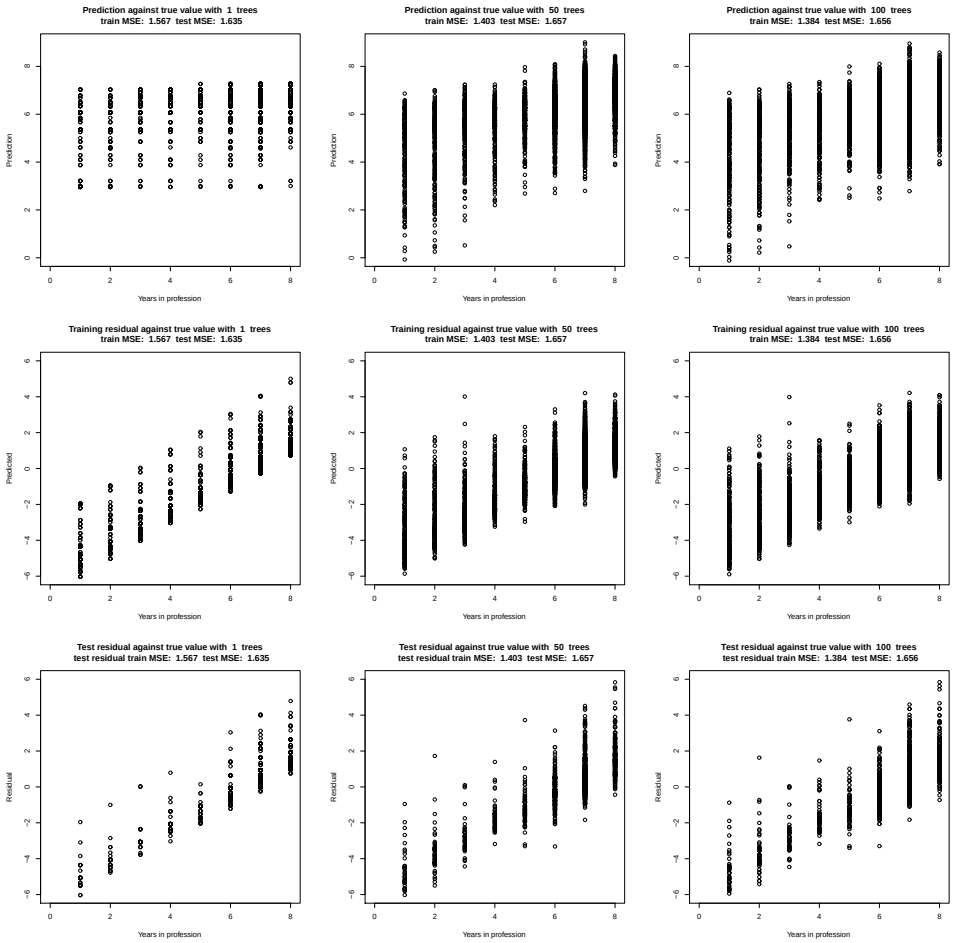


FIGURE 15.3: *Models from a stagewise regression of years in the profession against number of prescriptions for a large variety of medications for the data of* https://www.kaggle.com/roamresearch/prescriptionbasedprediction. **Top row:** *Model predictions against true values;* **middle row:** *residual against true values, training examples;* **bottom row:** *residuals against true values, test examples.*

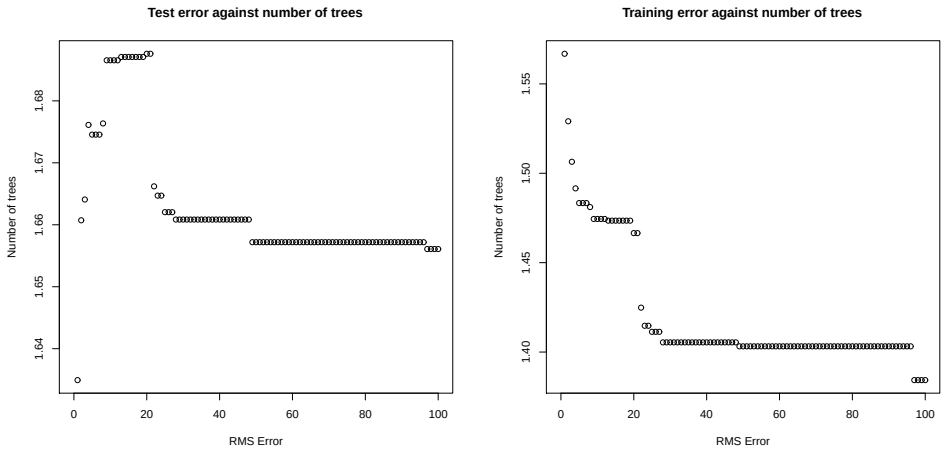


FIGURE 15.4: *Models from a stagewise regression of years in the profession against number of prescriptions for a large variety of medications for the data of* https://www.kaggle.com/roamresearch/prescriptionbasedprediction. **Left:** *training residual against number of trees;* **right:** *test error against number of trees. Notice that both go down, but there is a test-train gap.*

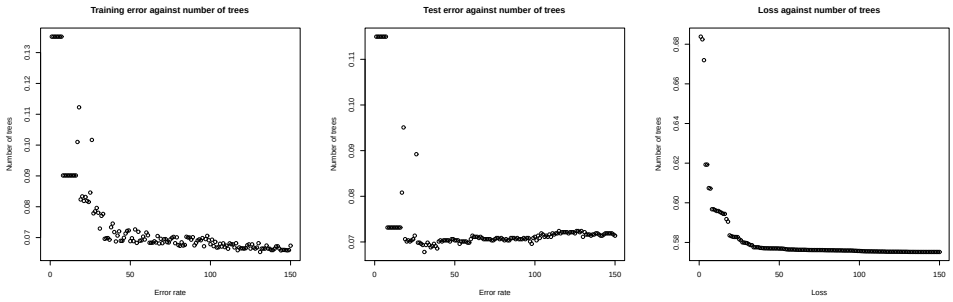


FIGURE 15.5: *Models for a boosted decision tree classifier predicting whether a given prescriber will write more than 10 opioid prescriptions in a year, using the data of* https://www.kaggle.com/apryor6/us-opiate-prescriptions. **Left:** *training error against number of trees;* **center:** *test error against number of trees;* **right:** *exponential loss against number of trees. Notice that both test and train error go down, but there is a test-train gap. Note also that lower exponential loss doesn't guarantee lower training error.*