Contents

CHAPTER 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:

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

- $\emptyset$ 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 426).
- $P(\{\mathcal{E}\})$ is the probability of event $\mathcal{E}$ (page 426).
- $P(\{\mathcal{E}\}|\{\mathcal{F}\})$ is the probability of event $\mathcal{E}$, conditioned on event $\mathcal{F}$ (page 426).
- $p(x)$ is the probability that random variable X will take the value x; also written $P(\{X = x\})$ (page 426).
- $p(x, y)$ 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 426).
- $\underset{x}{\text{argmax}}\ f(x)$ 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.

1.0.1 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.

1.1 ACKNOWLEDGEMENTS

Typos spotted by: Han Chen (numerous!), Henry Lin (numerous!), Eric Huber, Brian Lunt, Yusuf Sobh, Scott Walters, — Your Name Here — Jian Peng and Paris Smaragdis taught courses from versions of these notes, and improved them by detailed comments, suggestions and typo lists. 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.

CHAPTER 2

Some Preliminaries

2.1 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).

Vectors are always column vectors. I will use two notations for components of vectors. If I need to refer to the j'th component of a vector $\mathbf{x}$, I will write x_j. This is the standard notation, but it presents problems when I need to refer to the j'th component of the i'th vector $\mathbf{x}_i$. In this case, I 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).

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:

- mean $(\{x\})$ is the mean of the dataset $\{x\}$
- std $(\{x\})$ is the standard deviation of the dataset $\{x\}$
- var $(\{x\})$ is the variance of the dataset $\{x\}$
- median $(\{x\})$ is the standard deviation of the dataset $\{x\}$
- percentile$(\{x\}, k)$ is the $k\%$ percentile of the dataset $\{x\}$
- 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 **??**,

- $\mathsf{corr}\left(\{(x,y)\}\right)$ 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 426).
- $P(\{\mathcal{E}\})$ is the probability of event $\mathcal{E}$ (page 426).
- $P(\{\mathcal{E}\}|\{\mathcal{F}\})$ is the probability of event $\mathcal{E}$, conditioned on event $\mathcal{F}$ (page 426).
- $p(x)$ is the probability that random variable X will take the value x; also written $P(\{X = x\})$ (page 426).
- $p(x,y)$ 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 426).
- $\underset{x}{\text{argmax}}\ f(x)$ 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.

2.1.1 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.

2.2 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.

2.3 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.

TODO: TA list from giant recent offering

2.4 THINGS YOU SHOULD KNOW BEFORE WE BEGIN

TODO: list of terms from 361 book? possibly with section references?

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.

2.4.1 Programming Languages

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.

2.4.2 Probability, Random Variables, and Expectations

TODO: list of topics, and pointers to 361 book

2.4.3 Empirical Distributions, Expectations and Statistics

TODO: smooth in
You should have noticed we now have two notions each for mean, variance, covariance, and standard deviation. One, which we expounded in section **??**, describes datasets. We will call these **descriptive statistics**. The other, described above, is a property of probability distributions. We will call these **expectations**. In each case, the reason we have one name for two notions is that the notions are not really all that different.

Imagine we have a dataset $\{x\}$ of N items, where the i'th item is x_i. We can build a probability distribution out of this dataset, by placing a probability on each data item. We will give each data item the same probability (which must be $1/N$, so all probabilities add to 1). Write $\mathbb{E}[x]$ for the mean of this distribution. We have

$$\mathbb{E}[x] = \sum_i x_i p(x_i) = \frac{1}{N} \sum_i x_i = \mathsf{mean}\left(\{x\}\right).$$

The variances, standard deviations and covariance have the same property: For this particular distribution (sometimes called the **empirical distribution**), the expectations have the same value as the descriptive statistics (exercises).

In section **??**, we will see a form of converse to this fact. Imagine we have a dataset that consists of independent, identically distributed samples from a probability distribution (i.e. we know that each data item was obtained independently from the distribution). For example, we might have a count of heads in each of a number of coin flip experiments. Then the descriptive statistics will turn out to be accurate estimates of the expectations.

PART ONE

CLASSIFICATION

CHAPTER 3

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: 3.1 *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.

3.1 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.

3.1.1 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.

3.1.2 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 3.1: *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 4.1.*

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 3.1 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 3.1. 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.

3.1.3 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.*

3.2 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.

3.2.1 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 5). 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 3.1 *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.

3.3 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 3.1 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.

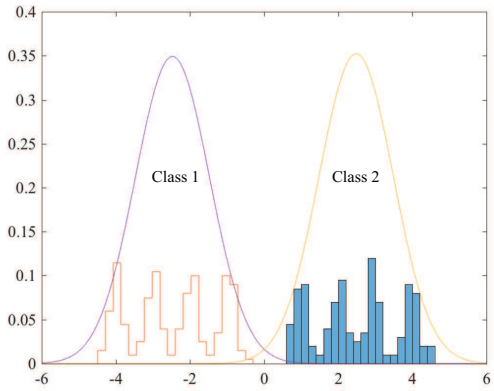


FIGURE 3.1: *The figure shows class conditional histograms of a feature x for two different classes. The histograms have been normalized so that the counts sum to one, so you can think of them as probability distributions. It should be fairly obvious that a normal model (superimposed) doesn't describe these histograms well. However, the normal model will result in a good naive bayes classifier.*

Worked example 3.2 *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.

3.3.1 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.

3.3.2 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.*

3.4 YOU SHOULD

3.4.1 remember these definitions:

3.4.2 remember these terms:

3.4.3 remember these facts:

3.4.4 remember these procedures:

3.4.5 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

3.1. 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.

3.2. 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?

3.3. 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.

3.4. 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 4

SVM's and Random Forests

4.1 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: 4.1 *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: 4.2 *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.

4.1.1 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 $\mathbf{x}_i$.

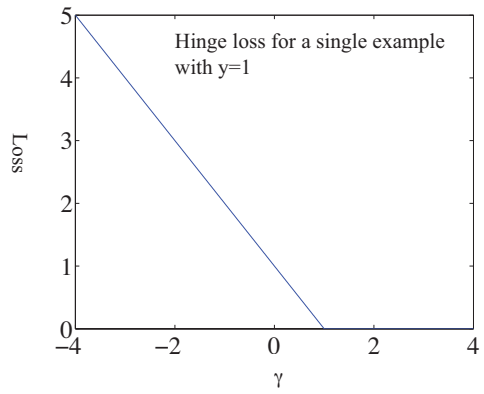


FIGURE 4.1: *The hinge loss, plotted for the case $y_i = 1$. The horizontal variable is the $\gamma_i = \mathbf{a}^T\mathbf{x}_i + b$ of the text. Notice that giving a strong negative response to this positive example causes a loss that grows linearly as the magnitude of the response grows. Notice also 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**.

4.1.2 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.

4.1.3 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

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

and that

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

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 4.2 and 4.3 show examples of these curves, which are sometimes known as **learning curves**.

4.1.4 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 4.1.3). 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 3.3.1). 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.

4.1.5 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: 4.1 *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 4.2 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: 4.2 *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: 4.3 *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 4.2 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

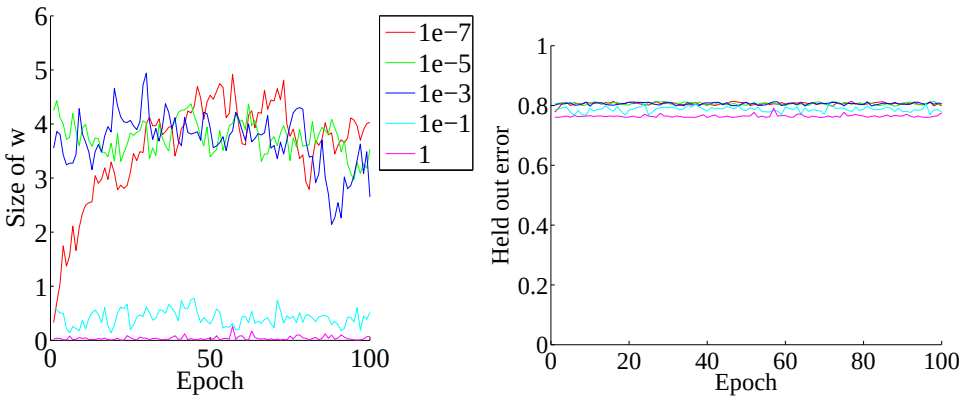


FIGURE 4.2: *On the* **left**, *the magnitude of the weight vector* $\mathbf{a}$ *at the end of each season for the first training regime described in the text. On the* **right**, *the accuracy on held out data at the end of each season. Notice how different choices of regularization parameter lead to 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 4.3 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.

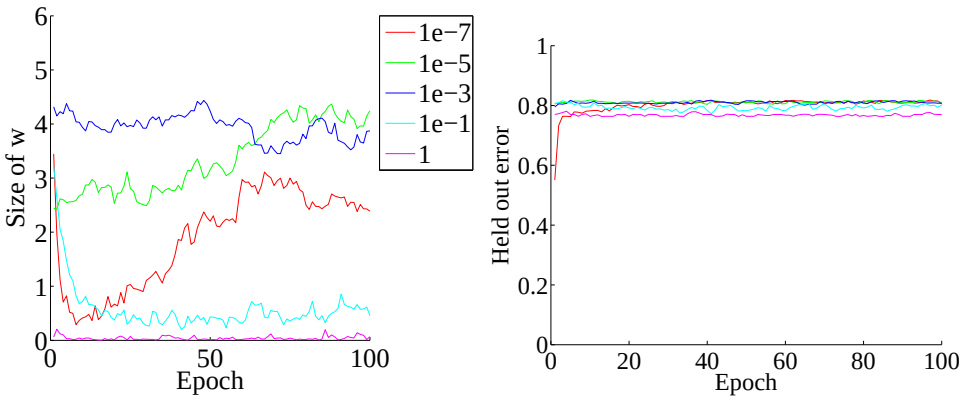


FIGURE 4.3: *On the* **left**, *the magnitude of the weight vector* **a** *at the end of each season for the second training regime described in the text. On the* **right**, *the accuracy on held out data at the end of each season. Notice how different choices of regularization parameter lead to different magnitudes of* **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.*

4.1.6 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.*

4.2 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 4.5 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 4.5, 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.

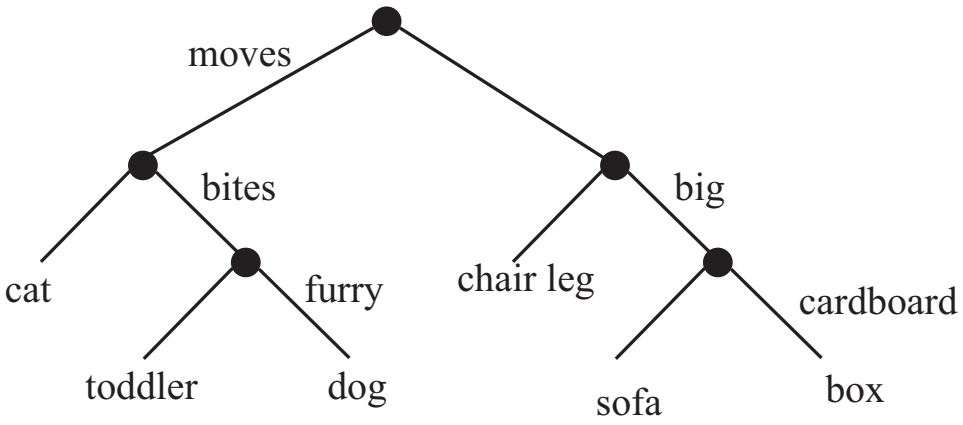


FIGURE 4.4: *This — the household robot's guide to obstacles — is a typical decision tree. I have labelled only one of the outgoing branches, because the other is the negation. So if the obstacle moves, bites, but isn't furry, then it's a toddler. In general, an item is passed down the tree until it hits a leaf. It is then labelled with the leaf's label.*

4.2.1 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

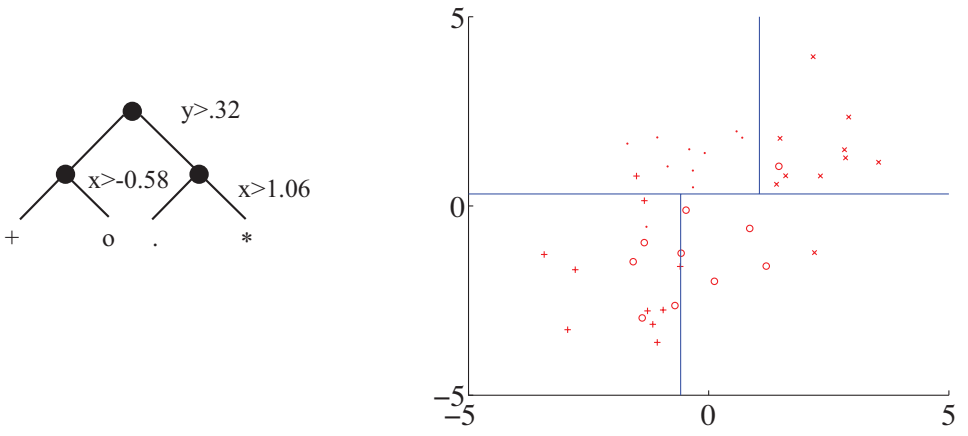


FIGURE 4.5: *A straightforward decision tree, illustrated in two ways. On the* **left**, *I have given the rules at each split; on the* **right**, *I have shown the data points in two dimensions, and the structure that the tree produces in the feature space.*

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 4.6 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 4.7 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.

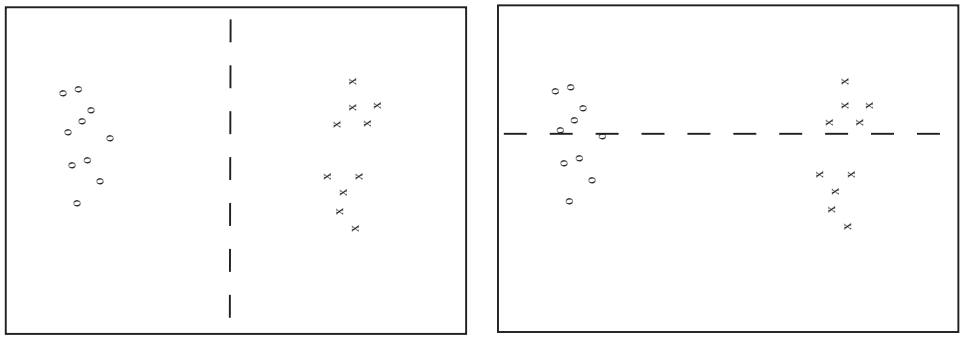


FIGURE 4.6: *Two possible splits of a pool of training data. Positive data is represented with an 'x', negative data with a 'o'. Notice that if we split this pool with the informative line, all the points on the left are 'o's, and all the points on the right are 'x's. This is an excellent choice of split — 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.

4.2.2 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

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

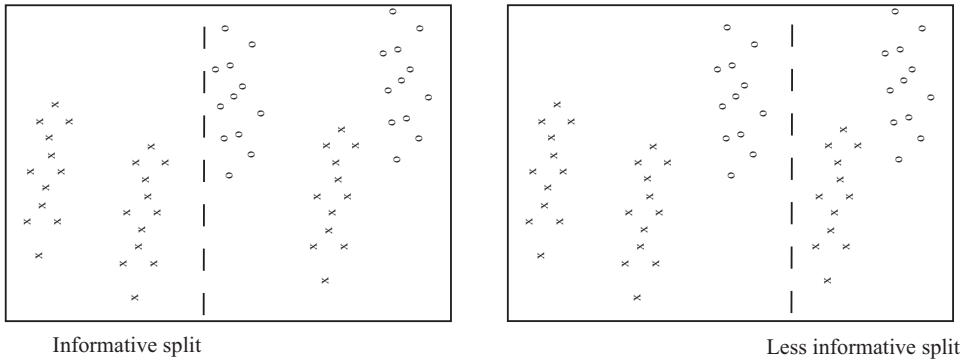


FIGURE 4.7: *Two possible splits of a pool of training data. Positive data is represented with an 'x', negative data with a 'o'. Notice that if we split this pool with the informative line, all the points on the left are 'x's, and two-thirds of the points on the right are 'o's. This means that knowing 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.

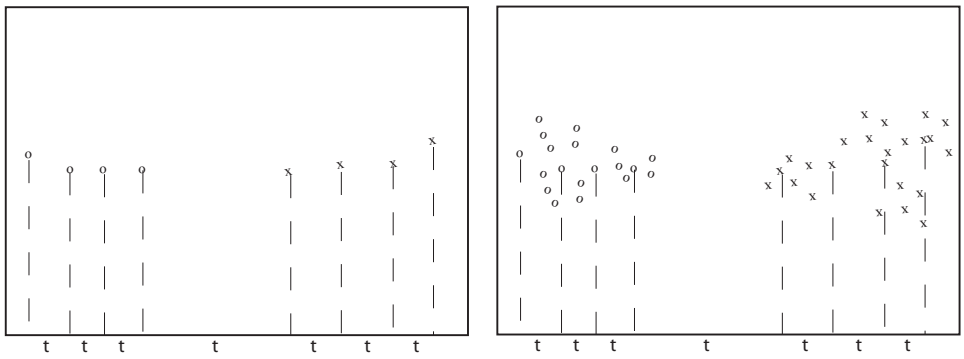


FIGURE 4.8: *We search for a good splitting threshold by looking at values of the chosen component that yield different splits. On the* **left***, I show a small dataset and its projection onto the chosen splitting component (the horizontal axis). For the 8 data points here, there are only 7 threshold 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 4.8). 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: 4.4 *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 4.5, otherwise use the procedure of box 4.6.

Overall approach to build a decision tree

Procedure: 4.5 *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: 4.6 *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

4.2.3 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.

4.2.4 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: 4.7 *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: 4.8 *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.

4.2.5 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: 4.9 *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 4.1 *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 3.1. 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.*

4.3 YOU SHOULD

4.3.1 remember these definitions:

4.3.2 remember these terms:

4.3.3 remember these facts:

4.3.4 use these procedures:

4.3.5 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

4.1. 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

4.2. 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

4.3. 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?

4.4. 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. For this exercise, we will use raw pixels as features.

4.5. Investigate classifying MNIST using naive bayes. Use the procedures of Section 3.3.1 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 3.3.1 if you're vague on this point).

4.6. 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?

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

4.8. 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 cases: untouched raw pixels and stretched bounding box raw pixels. Which works best? Why?

4.9. 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?

PART TWO

HIGH DIMENSIONAL DATA

CHAPTER 5

High-dimensional Data

We have a dataset that is a collection of d dimensional vectors. A dataset like this is hard to plot, though section 5.1 suggests some tricks that are helpful. Most readers will already know the mean as a summary (it's an easy generalization of the 1D mean). The covariance matrix may be less familiar. This is a collection of all covariances between pairs of components. We use covariances, rather than correlations, because covariances can be represented in a matrix easily. Natural transformations of the dataset lead to easy transformations of mean and the covariance matrix, which we exploit in the next few chapters. In turn, this means we can construct a transformation that produces a new dataset, whose covariance matrix has desirable properties, from any dataset.

High dimensional data has some nasty properties (it's usual to lump these under the name "the curse of dimension"). The data isn't where you think it is, and this can be a serious nuisance, making it difficult to fit complex probability models. The cure is to use extremely simple representations of the data. The most powerful of these is to think of a dataset as a collection of blobs of data. Each blob of data consists of points that are "reasonably close" to each other and "rather far" from other blobs. A blob can be modelled with a multivariate normal distribution. Our knowledge of what transformations do to a dataset's mean and covariance reveals the main points about the multivariate normal distribution.

5.1 SUMMARIES AND SIMPLE PLOTS

In this part, 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.

For 1D data, mean and variance are a very helpful description of data that had a unimodal histogram. If there is more than one mode, one needs to be somewhat careful to interpret the mean and variance, because the mean doesn't summarize the modes particularly well, and the variance depends on how the modes are placed. 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 9). 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 in this part. 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

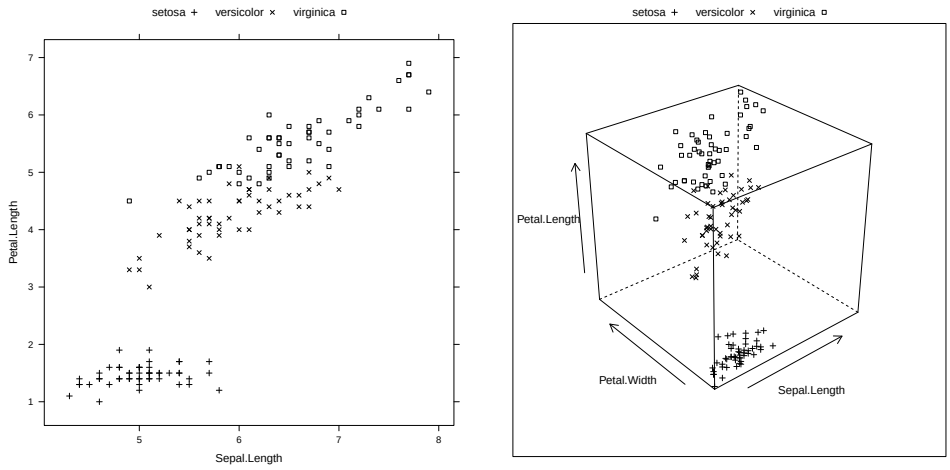


FIGURE 5.1: **Left:** *a 2D scatterplot for the Iris data. I have chosen two variables from the four, and have plotted each species with a different marker.* **Right:** *a 3D scatterplot for the same data. You can see from the plots that the species cluster quite tightly, and are different from one another. 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.*

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.

5.1.1 The Mean

For one-dimensional data, we wrote

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

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

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

and call this the mean of the data. Notice that each component of $\mathsf{mean}\left(\{\mathbf{x}\}\right)$ is the mean of that component of the data. There is not an easy analogue of the median,

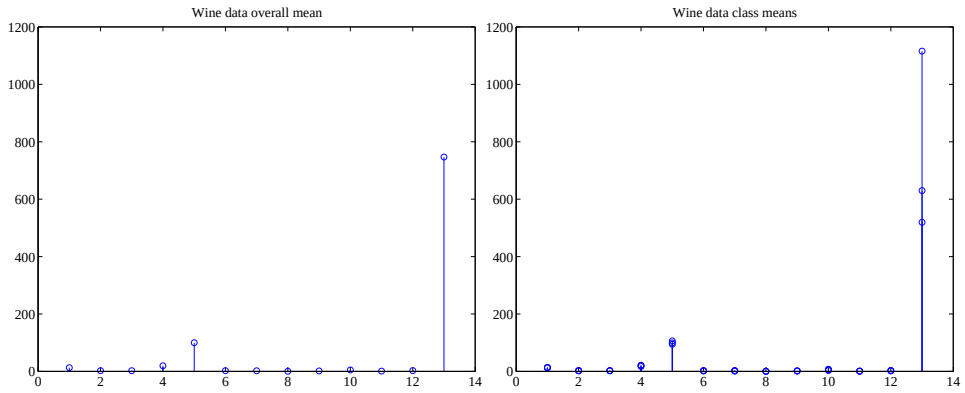


FIGURE 5.2: *On the* **left**, *a stem plot of the mean of all data items in the wine dataset, from* http://archive.ics.uci.edu/ml/datasets/Wine. *On the* **right**, *I have overlaid stem plots of each class mean from the wine dataset, from* http://archive.ics.uci.edu/ml/datasets/Wine, *so that you can see the differences between class means.*

however (how do you order high dimensional data?) and this is a nuisance. Notice that, just as for the one-dimensional mean, we have

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

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

5.1.2 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 5.1 shows such a scatterplot, for data that was originally four dimensional. This is the famous Iris dataset (it has to do with the botanical classification of irises), which was collected by Edgar Anderson in 1936, and made popular amongst statisticians by Ronald Fisher in that year. I found a copy at the UC Irvine repository of datasets that are important in machine learning (at http://archive.ics.uci.edu/ml/index.html). I will show several plots of this dataset.

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 5.2). 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 5.2).

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

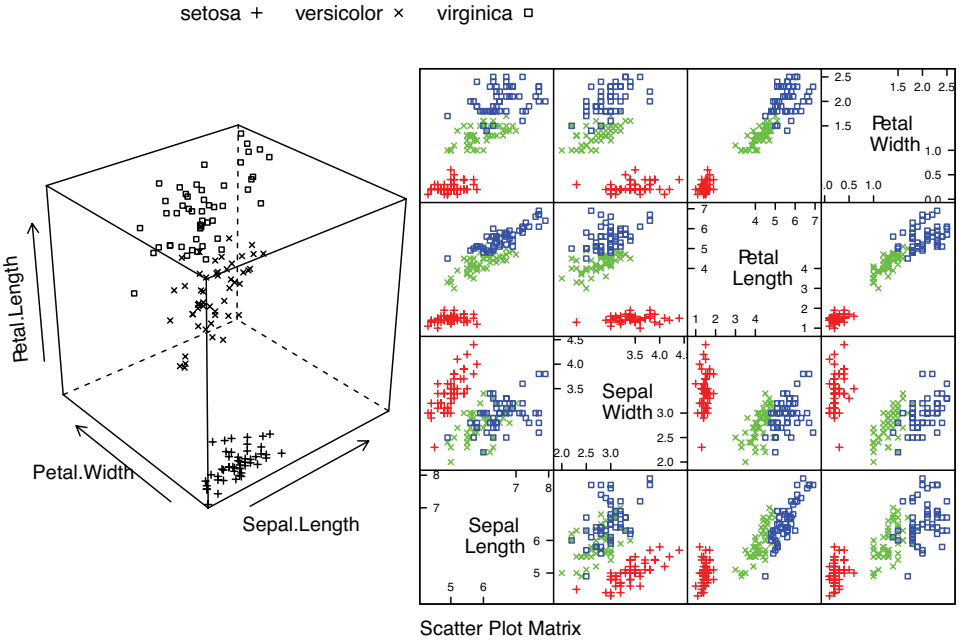


FIGURE 5.3: **Left:** *the 3D scatterplot of the iris data of Figure 5.1, for comparison.* **Right:** *a scatterplot matrix for the Iris data. There are four variables, measured for each of three species of iris. I have plotted each species with a different marker. You can see from the plot that the 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 5.3, which shows both a 3D scatter plot and a scatterplot matrix for the same dataset.

Figure 5.4 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 5.4, 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.

5.1.3 Covariance

Variance, standard deviation and correlation can each be seen as an instance of a more general operation on data. Extract two components from each vector of a dataset of vectors, yielding two 1D datasets of N items; write $\{x\}$ for one and $\{y\}$

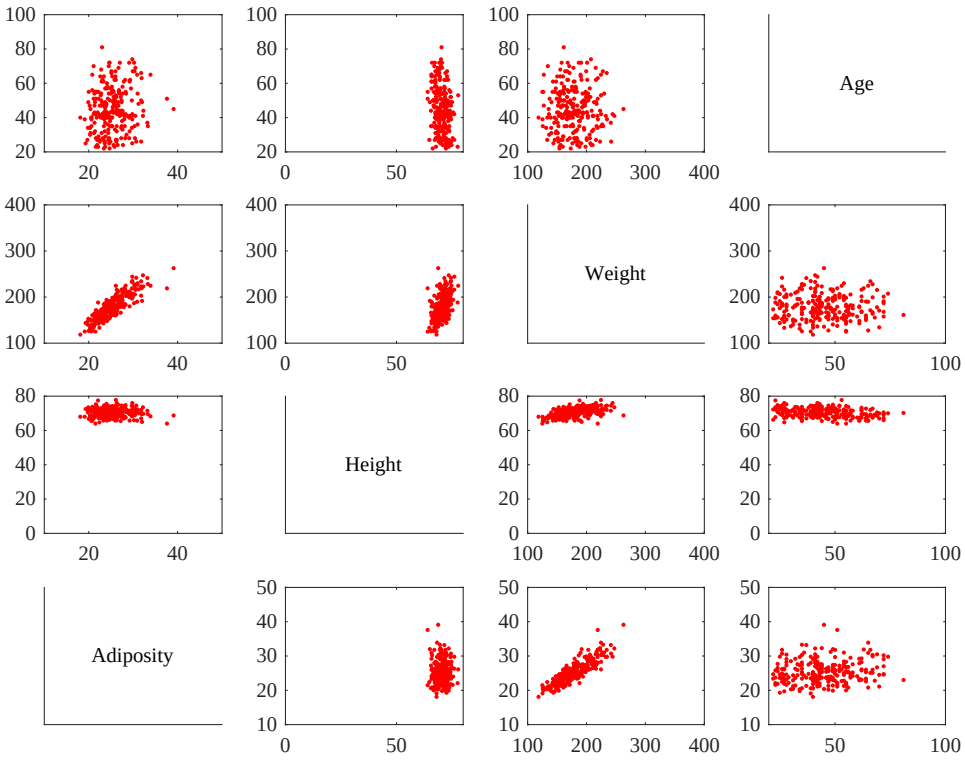


FIGURE 5.4: *This is a scatterplot matrix for four of the variables in the height weight dataset of http://www2.stetson.edu/~jrasp/data.htm. Each plot is a scatterplot of a pair of variables. The name of the variable for the horizontal axis is obtained by running your eye down the column; for the vertical 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).*

for the other. The i'th element of $\{x\}$ corresponds to the i'th element of $\{y\}$ (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: 5.1 *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. 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.

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.

5.1.4 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: 5.2 *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: 5.1 *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:

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

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.

5.2 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 5.5 shows a two-dimensional data set. Notice that there is obviously some correlation between the x and y coordinates (it's a diagonal

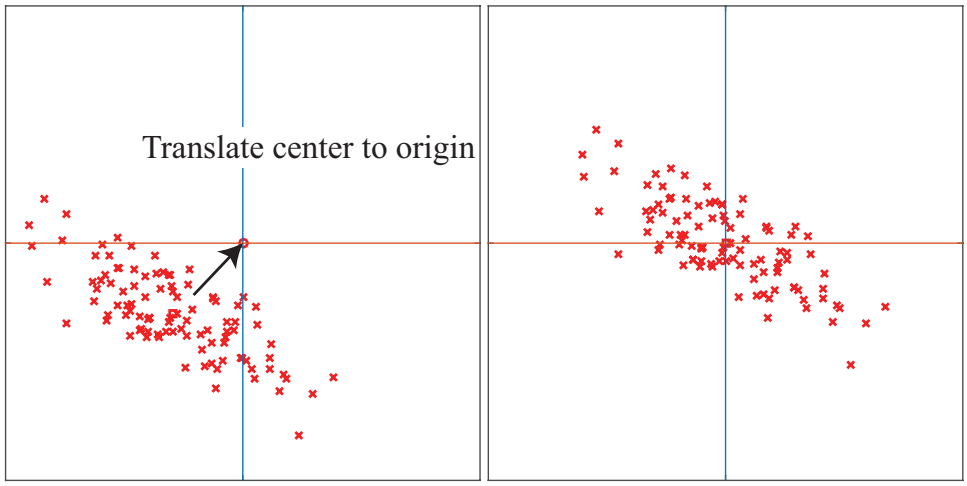


FIGURE 5.5: *On the* **left**, *a "blob" in two dimensions. This is a set of data points that lie somewhat clustered around a single center, given by the mean. I have plotted the mean of these data points with a hollow square (it's easier to see when there is a lot of data). To translate the blob to 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 5.5). 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.

5.2.1 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.

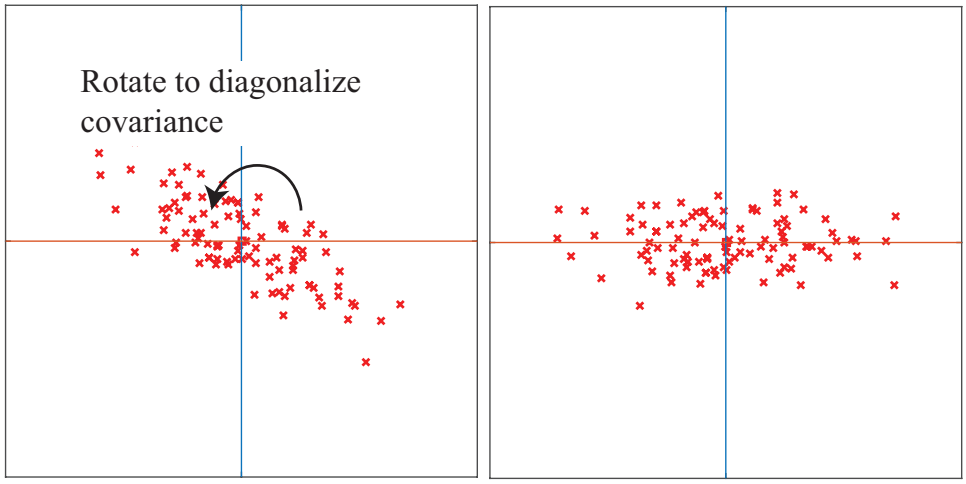


FIGURE 5.6: *On the* **left**, *the translated blob of figure 5.5. This blob lies somewhat diagonally, because the vertical and horizontal components are correlated. On the* **right**, *that blob of data rotated so that there is no correlation between these components. We can now describe the blob by 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

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

so you get the new mean by multiplying the original mean by $\mathcal{A}$ and adding $\mathbf{b}$; equivalently, by transforming the old mean the same way you transformed the points.

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

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

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.

Remember this: *Transform a dataset* $\{\mathbf{x}\}$ *into a new dataset* $\{\mathbf{m}\}$, *where* $\mathbf{m}_i = \mathcal{A}\mathbf{x}_i + \mathbf{b}$. *Then*

$$\begin{aligned} \mathsf{mean}(\{\mathbf{m}\}) &= \mathcal{A}\mathsf{mean}(\{\mathbf{x}\}) + \mathbf{b} \\ \mathsf{Covmat}(\{\mathbf{m}\}) &= \mathcal{A}\mathsf{Covmat}(\{\mathbf{x}\})\mathcal{A}^T. \end{aligned}$$

5.2.2 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: 5.1 *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: 5.2 *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.

5.2.3 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 5.5 and 5.6.

Useful Fact: 5.3 *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.

5.3 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.

5.3.1 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 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 $P(\mathbf{x}) = P(x_1)P(x_2)\dots P(x_d)$. 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.

5.3.2 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.

Instead, high dimensional data is typically represented in terms 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. This representation is explored in part **??**

It can be hard to get accurate estimates of the mean of a high dimensional normal distribution (and so of any other). This is mostly a minor nuisance, but it's worth understanding what is happening. The data is a set of N IID samples of a normal distribution with mean μ and covariance Σ in d dimensional space. These points will tend to lie far away from one another. But they may not be evenly spread out, so there may be slightly more points on one side of the true mean than on the other, and so the estimate of the mean is likely noisy. It's tough to be crisp about what it means to be on one side of the true mean, so I'll do this in algebra, too. The estimate of the mean is

$$X^N = \frac{\sum_i \mathbf{x}_i}{N}$$

which is a random variable, because different draws of data will give different values of X^N. In the exercises, you will show that $\mathbb{E}\left[X^N\right]$ is μ (so the estimate is reasonable). One reasonable measure of the total error in estimating the mean is $(X^N - \mu)^T(X^N - \mu)$. In the exercises, you will show that the expected value of this error is

$$\frac{\text{Trace}(\Sigma)}{N}$$

which may grow with d unless Σ has some strong properties. Likely, your estimate of the mean for a high dimensional distribution is poor.

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. Another is that your estimate of the mean for a high dimensional distribution is likely poor.*

5.4 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: 5.4 *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).

5.4.1 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 $\mathsf{mean}\left(\{\mathbf{x}_i\}\right)$, and the covariance is $\mathsf{Covmat}\left(\{\mathbf{x}_i\}\right)$. 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.

5.4.2 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**.

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* $\mathsf{Covmat}\,(\{\mathbf{x}\})$.

5.4.3 Descriptive Statistics and Expectations

It is quite usual to use each of the terms mean, variance, covariance, and standard deviation in two slightly different ways. One sense of each term, as in the discription of covariance above, describes a property of a dataset. Terms used in this sense are known as **descriptive statistics**. The other sense is a property of probability distributions; so mean, for example, means $\mathbb{E}[X]$; variance means $\mathbb{E}\left[(X - \mathbb{E}[X])^2\right]$; and so on. Terms used in this sense are known as **expectations**. The reason we use one name for two notions is that the notions are not really all that different.

Here is a useful construction to illustrate the point. Imagine we have a dataset $\{\mathbf{x}\}$ of N items, where the i'th item is $\mathbf{x}_i$. Build a random variable X using this dataset by placing the same probability on each data item. This means that each data item has probability $1/N$. Write $\mathbb{E}[X]$ for the mean of this distribution. We have

$$\mathbb{E}[X] = \sum_i x_i P(x_i) = \frac{1}{N}\sum_i x_i = \mathsf{mean}\,(\{x\})$$

and, by the same reasoning,

$$\mathsf{var}[X] = \mathsf{var}\,(\{x\}).$$

This construction works for standard deviation and covariance, too. For this particular distribution (sometimes called the **empirical distribution**), the expectations have the same value as the descriptive statistics.

There is a form of converse to this fact, which you should have seen already, and which we shall see on and off later. Imagine we have a dataset that consists of independent, identically distributed samples from a probability distribution (i.e. we know that each data item was obtained independently from the distribution). For example, we might have a count of heads in each of a number of coin flip experiments. The **weak law of large numbers** says the descriptive statistics will turn out to be accurate estimates of the expectations.

In particular, assume we have a random variable X with distribution $P(X)$ which has finite variance. We want to estimate $\mathbb{E}[X]$. Now if we have a set of IID samples of X, which we write x_i, write

$$X_N = \frac{\sum_{i=1}^{N} x_i}{N}.$$

This is a random variable (different sets of samples yield different values of X_N), and the weak law of large numbers gives that, for any positive number ϵ

$$\lim_{N \to \infty} P(\{|X_N - \mathbb{E}[X]| > \epsilon\}) = 0.$$

You can interpret this as saying that, that for a set of IID random samples x_i, the probability that

$$\frac{\sum_{i=1}^{N} X_i}{N}$$

is very close to $\mathbb{E}[X]$ for large N

Useful Facts: 5.5 *Weak law of large numbers*

Given a random variable X with distribution $P(X)$ which has finite variance, and a set of N IID samples $\mathbf{x}_i$ from $P(X)$, write

$$X_N = \frac{\sum_{i=1}^{N} x_i}{N}.$$

Then for any positive number ϵ

$$\lim_{N \to \infty} P(\{|X_N - \mathbb{E}[X]| > \epsilon\}) = 0.$$

Remember this: *Mean, variance, covariance and standard deviation can refer either to properties of a dataset, or to expectations. The sense usually tells you which. There is a strong relationship between these senses. Given a dataset, you can construct an empirical distribution, whose mean, variance and covariances (intepreted as expectations) have the same values as the mean, variance and covariances (interpreted as descriptive statistics). If a dataset is an IID sample of a probability distribution, the mean, variance and covariances (interpreted as descriptive statistics) are usually very good estimates of the values of the mean, variance and covariances (interpreted as expectations).*

5.5 YOU SHOULD

5.5.1 remember these definitions:

5.5.2 remember these terms:

5.5.3 remember these facts:

5.5.4 remember these procedures:

PROBLEMS

Summaries

5.1. 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.

5.2. 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* as

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

(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?

5.3. You have a dataset $\{\mathbf{x}\}$ of N vectors, $\mathbf{x}_i$, each of which is d-dimensional. Assume $\mathsf{mean}(\{\mathbf{x}\}) = 0$. We will consider a linear function of this dataset. Write $\mathbf{a}$ for some vector; then the value of this linear function evaluated on the i'th data item is $\mathbf{a}^T\mathbf{x}_i$. Write $f_i(\mathbf{a}) = \mathbf{a}^T\mathbf{x}_i$. We can make a new dataset $\{f(\mathbf{a})\}$ out of these f_i (the notation is to remind you that this dataset depends on the choice of vector $\mathbf{a}$).

(a) Show that $\mathsf{var}(\{f(s\mathbf{a})\}) = s^2\mathsf{var}(\{f(\mathbf{a})\})$.

(b) The previous subexercise means that, to choose $\mathbf{a}$ to obtain a dataset with large variance in any kind of sensible way, we need to insist that $\mathbf{a}^T\mathbf{a}$ is kept constant. Show that

$$\text{Maximize } \mathsf{var}(\{f\})(\mathbf{a}) \text{ subject to } \mathbf{a}^T\mathbf{a} = 1$$

is solved by the eigenvector of $\mathsf{Covmat}(\{x\})$ corresponding to the largest eigenvalue. (You need to know Lagrange multipliers to do this, but you should.)

5.4. You have a dataset $\{\mathbf{x}\}$ of N vectors, $\mathbf{x}_i$, each of which is d-dimensional. We will consider two linear functions of this dataset, given by two vectors $\mathbf{a}$, $\mathbf{b}$.

(a) Show that $\mathsf{cov}\left(\left\{\mathbf{a}^T\mathbf{x}\right\}, \left\{\mathbf{b}^T\mathbf{x}\right\}\right) = \mathbf{a}^T\mathsf{Covmat}(\{\mathbf{x}\})\mathbf{b}$. This is easier to do if you show that the mean has no effect on covariance, and then do the math assuming $\mathbf{x}$ has zero mean.

(b) Show that the correlation between $\mathbf{a}^T\mathbf{x}$ and $\mathbf{b}^T\mathbf{x}$ is given by

$$\frac{\mathbf{a}^T\mathsf{Covmat}(\{\mathbf{x}\})\mathbf{b}}{\sqrt{\mathbf{a}^T\mathsf{Covmat}(\{\mathbf{x}\})\mathbf{a}}\sqrt{\mathbf{b}^T\mathsf{Covmat}(\{\mathbf{x}\})\mathbf{b}}}.$$

5.5. It is sometimes useful to map a dataset to have zero mean and unit covariance. Doing so is known as whitening the data (for reasons I find obscure). This can

be a sensible thing to do when we don't have a clear sense of the relative scales of the components of each data vector or whiten the data might be that we know relatively little about the meaning of each component. You have a dataset $\{\mathbf{x}\}$ of N vectors, $\mathbf{x}_i$, each of which is d-dimensional. Write $\mathcal{U}$, Λ for the eigenvectors and eigenvalues of $\mathsf{Covmat}(\{\mathbf{x}\})$.

(a) Show that $\Lambda \geq 0$

(b) Assume that some diagonal element of Λ is zero. How do you interpret this?

(c) Assume that all diagonal elements of Λ are greater than zero. Write $\Lambda^{1/2}$ for the matrix whose diagonal is the non-negative square roots of the diagonal of Λ. Write $\{\mathbf{y}\}$ for the dataset of vectors where $\mathbf{y}_i = (\Lambda^{1/2})^{-1}\mathcal{U}^T(\mathbf{x}_i - \mathsf{mean}(\{x\}))$. Show that $\mathsf{Covmat}(\{y\})$ is the identity matrix.

(d) Write $\mathcal{O}$ for some orthonormal matrix. Using the notation of the previous subexercise, and writing $\mathbf{z}_i = \mathcal{O}\mathbf{y}_i$, show that $\mathsf{Covmat}(\{z\})$ is the identity matrix. Use this information to argue that there is not a unique version of a whitened dataset.

The Multivariate Normal Distribution

5.6. A dataset of points (x, y) has zero mean and covariance

$$\Sigma = \begin{pmatrix} k_1^2 & 0 \\ 0 & k_2^2 \end{pmatrix}$$

with $k_1 > k_2$.

(a) Show that the standard deviation of the x coordinate is $\mathsf{abs}(k_1)$ and of the y coordinate is $\mathsf{abs}(k_2)$.

(b) Show that the set of points that satisfies

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

is an ellipse.

(c) Show that the major axis of the ellipse is the x axis, the minor axis of the ellipse is the y axis, and the center of the ellipse is at $(0, 0)$.

5.7. For Σ a positive definite matrix, μ some two dimensional vector, show that the family of points that satisfies

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

is an ellipse. An easy way to do this is to notice that ellipses remain ellipses when rotated and translated, and exploit the previous exercise.

The Curse of Dimension

5.8. A dataset consists of N IID samples from a multivariate normal distribution with dimension d. The mean of this distribution is zero, and its covariance matrix is the identity. You compute

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

The number you compute is a random variable, because you will compute a slightly different number for each different sample you draw. It turns out that the distribution of X^N is normal because the sum of normally distributed random variables is normal. You should remember (or, if you don't, memorize) the fact that

- a sample of a (1D) normal random variable is within one standard deviation of its mean about 66% of the time;
- a sample of a (1D) normal random variable is within two standard deviations of its mean about 95% of the time;
- a sample of a (1D) normal random variable is within three standard deviations of its mean about 99% of the time.

(a) Show that each component of X^N has expected value zero and variance $1/N$.
(b) Argue that about $d/3$ of the components have absolute value greater than $1/N$.
(c) Argue that about $d/20$ of the components have absolute value greater than $2/N$.
(d) Argue that about $d/100$ of the components have absolute value greater than $3/N$.
(e) What happens when d is very large compared to N?

5.9. For a dataset that consists of N IID samples $\mathbf{x}_i$ from a multivariate normal distribution with mean μ and covariance Σ, you compute

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

The number you compute is a random variable, because you will compute a slightly different number for each different sample you draw.
(a) Show that $\mathbb{E}\left[X^N\right] = \mu$. You can do this by noticing that, if $N = 1$, $\mathbb{E}\left[X^1\right] = \mu$ fairly obviously. Now use the fact that each of the samples is independent.
(b) The random variable $T^N = (X^N - \mu)^T(X^N - \mu)$ is one reasonable measure of how well X^N approximates μ. Show that

$$\mathbb{E}\left[T^N\right] = \frac{\text{Trace}(\Sigma)}{N}.$$

Do this by noticing that $\mathbb{E}\left[T^N\right]$ is the sum of the variances of the components of X^N. This exercise is much easier if you notice that translating the normal distribution to have zero mean doesn't change anything (so it's enough to work out the case where $\mu = \mathbf{0}$).

CHAPTER 6

Principal Component 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 choose a small set of vectors, so that each item in the original dataset can be represented as the mean vector plus a weighted sum of this set. This representation means we can think of the dataset as lying on a low dimensional space inside the original space. It's an experimental fact that this model of a dataset is usually accurate for real high-dimensional data, and it is often an extremely convenient model. Furthermore, representing a dataset like this very often suppresses noise – if the original measurements in your vectors are noisy, the low dimensional representation may be closer to the true data than the measurements are.

6.1 REPRESENTING DATA ON PRINCIPAL COMPONENTS

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. Most high dimensional datasets display another important property: many, or most, of the diagonal entries in the covariance matrix are very small. This means we can build a low dimensional representation of the high dimensional dataset that is quite accurate.

6.1.1 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 5.3; 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

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

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 mean 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

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

which is the sum of the diagonal elements of the covariance matrix from r, r to d, d. Equivalently, writing λ_i for the i'th eigenvalue of $\mathsf{Covmat}(\{x\})$ and assuming the eigenvalues are sorted in descending order, the error is

$$\sum_{j>s}^{j=d} \lambda_j$$

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. In that case, we could think about the data as being s dimensional. Figure 6.1 shows the result of using this approach to represent the blob I'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.

6.1.2 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 6.2 shows the dataset of figure 5.4, 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 6.2 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

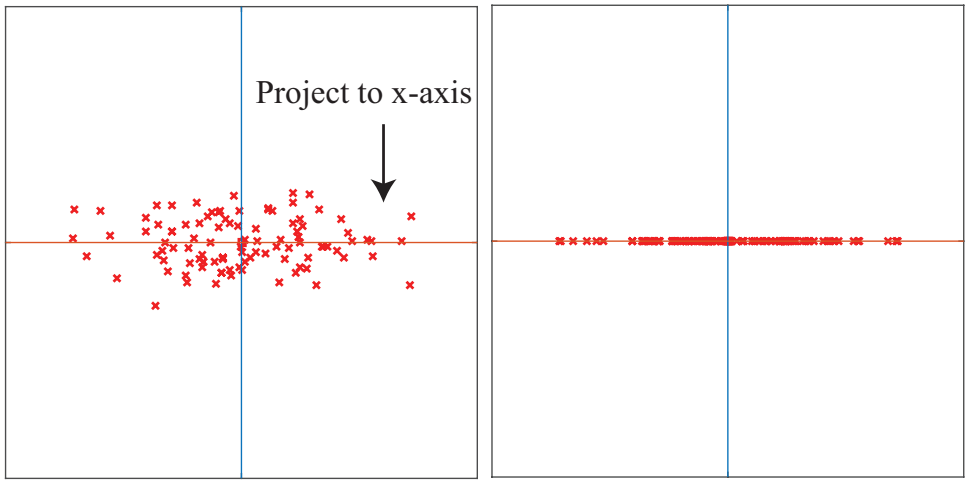


FIGURE 6.1: *On the* **left**, *the translated and rotated blob of figure 5.6. This blob is stretched — one direction has more variance than another. Setting the y coordinate to zero for each of these datapoints results in a representation that has relatively low error, because there isn't much variance in these values. This results in the blob on the* **right**. *The text shows how the error that results from this projection is computed.*

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 (have you seen a four-dimensional cigar recently?), but it's helpful. You can think of each panel of this figure as showing views down each of the four axes of the cigar.

Now look at figure 6.3. 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 6.2 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 coordinates. Figure 6.4 shows what happens to the data of Figure 5.4. 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

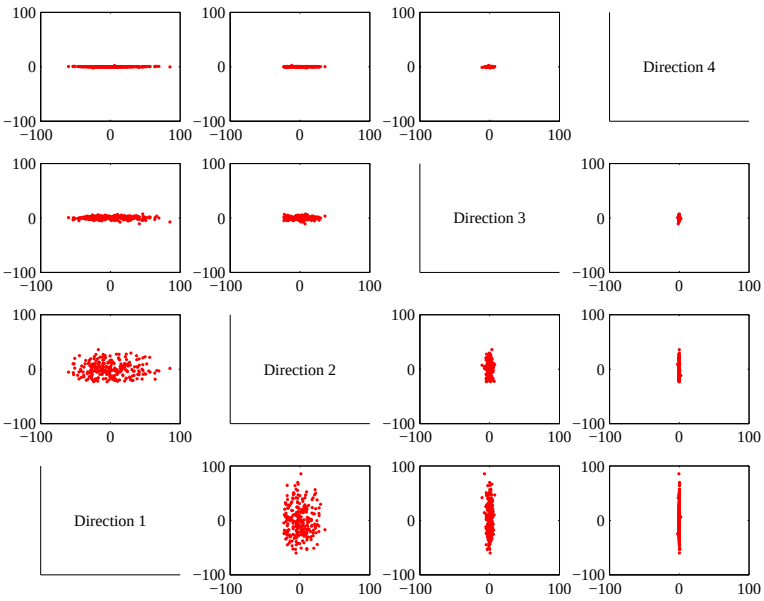


FIGURE 6.2: *A panel plot of the bodyfat dataset of figure 5.4, now rotated so that the covariance between all pairs of distinct dimensions is zero. Now we do not know names for the directions — they're linear combinations of the original variables. Each scatterplot is on the same set of axes, so you can see that the dataset extends more in some directions than in others. You should notice that, in some directions, there is very little variance. This suggests that replacing the coefficient in those directions with zero (as in figure 6.1) should result in a representation of the data that has very little error.*

accurately.

6.1.3 Representing Data on Principal Components

Now consider undoing the rotation and translation for our projected dataset $\{\mathbf{p}\}$. 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 $\mathcal{U}$ and w_{ij} for a weight value, we have

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

What is important about this sum is that s is usually a lot less than d. In turn, this means that we are representing the dataset using a lower dimensional dataset. We

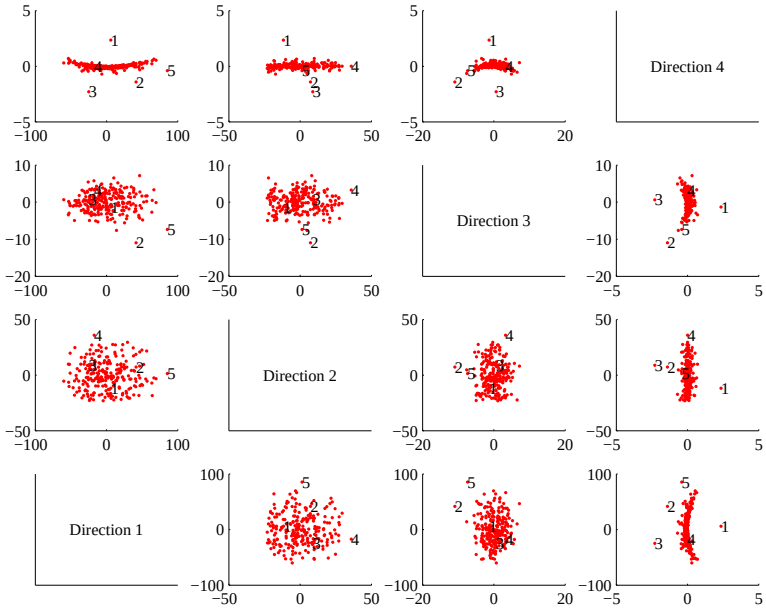


FIGURE 6.3: *A panel plot of the bodyfat dataset of figure 5.4, now rotated so that the covariance between all pairs of distinct dimensions is zero. Now we do not know names for the directions — they're linear combinations of the original variables. Compare this figure with figure 6.3; in that figure, the axes were the same, but in this figure 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.*

choose an s dimensional flat subspace of d dimensional space, and represent each data item with a point that lies on in that subset. The $\mathbf{u}_j$ are known as **principal components** (sometimes **loadings**) of the dataset; the $r_i^{(j)}$ are sometimes known as **scores**, but are usually just called **coefficients**. Forming the representation is called **principal components analysis** or **PCA**. The weights w_{ij} are actually easy to evaluate. We have that

$$w_{ij} = r_i^{(j)} = (\mathbf{x}_i - \mathsf{mean}\,(\{\mathbf{x}\}))^T \mathbf{u}_j.$$

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.*

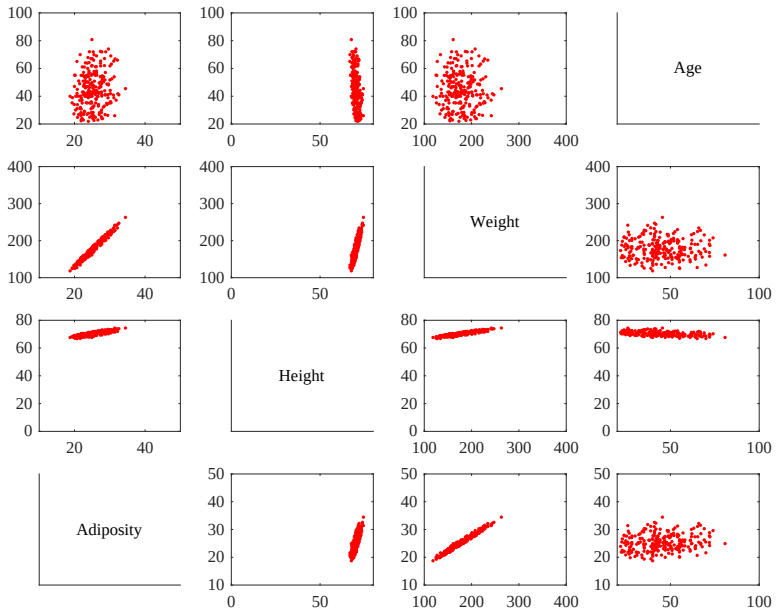


FIGURE 6.4: *The data of Figure 5.4, represented by translating and rotating so that the covariance is diagonal, projecting off the two smallest directions, then undoing the rotation and translation. This blob of data is two dimensional (because we projected off two dimensions – figure 6.2 suggested this was safe), but is represented in a four dimensional space. You can think of it as a thin two dimensional pancake of data in the four dimensional space (you should compare to Figure 5.4 on page 58). 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 flat plate at a slight slant.*

6.1.4 The Error in a Low Dimensional Representation

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

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

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

$$\frac{1}{N}\sum_i \|\mathbf{x}_i - \hat{\mathbf{x}}_i\|^2 = \frac{1}{N}\sum_i \|\mathbf{r}_i - \mathbf{p}_i\|^2 = \sum_{j>s}^{j=d} \lambda_j$$

which is easy to evaluate, because these are the values of the $d-s$ eigenvalues of $\mathsf{Covmat}(\{\mathbf{x}\})$ 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 6.5. You can see that the sum of remaining eigenvalues is small.

Procedure: 6.1 *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 6.5). It is quite usual to do this by hand.
Constructing a low-dimensional representation: For $1 \leq j \leq s$, write $\mathbf{u}_i$ for the i'th column of $\mathcal{U}$. Represent the data point $\mathbf{x}_i$ as

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

The error in this representation is

$$\frac{1}{N}\sum_i \|\mathbf{x}_i - \hat{\mathbf{x}}_i\|^2 = \sum_{j>s}^{j=d} \lambda_j$$

6.1.5 Extracting a Few Principal Components with NIPALS

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.

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. As usual, we have N data items, each a d

dimensional column vector. We will now arrange these into a matrix,

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

where each *row* of the matrix is a data vector. 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

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

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

$$\|\mathcal{X} - \mathbf{w}\mathbf{u}^T\|_F{}^2$$

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. At the right $\mathbf{w}$ and $\mathbf{u}$, the gradient of the cost function will be zero.

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. Notice that, if we know the right $\mathbf{u}$, then the equation $\nabla_{\mathbf{w}} C = 0$ is linear in $\mathbf{w}$. Similarly, if we know the right $\mathbf{w}$, then the equation $\nabla_{\mathbf{u}} C = 0$ is linear in $\mathbf{u}$. 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

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

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

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

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

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

and

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

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).

6.1.6 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 a covariance matrix. 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

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

Now we need a shorthand to ensure that sums run over only known values. Write $\mathcal{V}(k)$ for the set of column (resp. row) indices of known values for a given row (resp. column index) k. So $i \in \mathcal{V}(k)$ means all i such that x_{ik} is known *or* all i such that x_{ki} is known (the context will tell you which). We have

$$\frac{\partial C}{\partial w_k} = \sum_{j \in \mathcal{V}(k)} \left(x_{kj} - w_k u_j\right) u_j$$

and

$$\frac{\partial C}{\partial u_l} = \sum_{i \in \mathcal{V}(l)} \left(x_{il} - w_i u_l\right) w_i.$$

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

$$\hat{w_k} = \frac{\sum_{j \in \mathcal{V}(k)} x_{kj} u_j}{\sum_j u_j^{(n)} u_j^{(n)}}$$

and

$$\hat{u_l} = \frac{\sum_{i \in \mathcal{V}(l)} x_{il} w_l}{\sum_i \hat{w_i} \hat{w_i}}$$

We then normalize as before.

Procedure: 6.2 *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 $\mathcal{X}$. Write $\mathcal{V}(k)$ for the set of indices of known values for a given row or column index k. Now iterate

- compute

$$\hat{w_k} = \frac{\sum_{j \in \mathcal{V}(k)} x_{kj} u_j}{\sum_j u_j^{(n)} u_j^{(n)}}$$

  and

$$\hat{u_l} = \frac{\sum_{i \in \mathcal{V}(l)} x_{il} w_l}{\sum_i \hat{w_i} \hat{w_i}};$$

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

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

  and

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

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

6.1.7 PCA as Smoothing

Assume that each data item $\mathbf{x}_i$ is noisy. We use a simple noise model. Write $\tilde{\mathbf{x}}_i$ for the true underlying value of the data item, and ξ_i for the value of a normal random variable with zero mean and covariance $\sigma^2\mathcal{I}$. Then we use the model

$$\mathbf{x}_i = \tilde{\mathbf{x}}_i + \xi_i$$

(so the noise in each component is independent, and has variance σ^2; this is known as **additive, zero-mean, independent gaussian noise**). You should think of the measurement $\mathbf{x}_i$ as an estimate of $\tilde{\mathbf{x}}_i$. A principal component analysis of $\mathbf{x}_i$ can produce an estimate of $\tilde{\mathbf{x}}_i$ that is closer than the measurements are.

There is a subtlety here, because the noise is random, but we see the values of the noise. This means that $\mathsf{Covmat}(\{\xi\})$ (i.e. the covariance of the observed numbers) is the value of a random variable (because the noise is random) whose mean is $\sigma^2\mathcal{I}$ (because that's the model). The subtlety is that $\mathsf{mean}(\{\xi\})$ will not necessarily be exactly $\mathbf{0}$ and $\mathsf{Covmat}(\{\xi\})$ will not necessarily be exactly $\sigma^2\mathcal{I}$. The weak law of large numbers tells us that $\mathsf{Covmat}(\{\xi\})$ will be extremely close to its expected value (which is $\sigma^2\mathcal{I}$) for a large enough dataset. We will assume that $\mathsf{mean}(\{\xi\}) = \mathbf{0}$ and $\mathsf{Covmat}(\{\xi\}) = \sigma^2\mathcal{I}$.

The first step is to write $\tilde{\Sigma}$ for the covariance matrix of the true underlying values of the data, and $\mathsf{Covmat}(\{\mathbf{x}\})$ for the covariance of the observed data. Then it is straightforward that

$$\mathsf{Covmat}(\{\mathbf{x}\}) = \tilde{\Sigma} + \sigma^2\mathcal{I}$$

because the noise is independent of the measurements. Notice that if $\mathcal{U}$ diagonalizes $\mathsf{Covmat}(\{\mathbf{x}\})$, it will also diagonalize $\tilde{\Sigma}$. Write $\tilde{\Lambda} = \mathcal{U}^T\tilde{\Sigma}\mathcal{U}$. We have

$$\mathcal{U}^T\mathsf{Covmat}(\{\mathbf{x}\})\mathcal{U} = \Lambda = \tilde{\Lambda} + \sigma^2\mathcal{I}.$$

Now think about the diagonal entries of Λ. If they are large, then they are quite close to the corresponding components of $\tilde{\Lambda}$, but if they are small, it is quite likely they are the result of noise. But these eigenvalues are tightly linked to error in a PCA representation.

In PCA (procedure 6.1), the d dimensional data point $\mathbf{x}_i$ is represented by

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

where $\mathbf{u}_j$ are the principal components. This representation is obtained by setting the coefficients of the $d-s$ principal components with small variance to zero. The error in representing $\{\mathbf{x}\}$ with $\{\hat{\mathbf{x}}\}$ follows from section 6.1.4 and is

$$\frac{1}{N}\sum_i \|\mathbf{x}_i - \hat{\mathbf{x}}_i\|^2 = \sum_{j>s}^{j=d}\lambda_j.$$

Now consider the error in representing $\tilde{\mathbf{x}}_i$ (which we don't know) by $\mathbf{x}_i$ (which we do). The average error over the whole dataset is

$$\frac{1}{N}\sum_i \|\mathbf{x}_i - \tilde{\mathbf{x}}_i\|^2.$$

Because the variance of the noise is $\sigma^2\mathcal{I}$, this error must be $d\sigma^2$. Alternatively, we could represent $\tilde{\mathbf{x}}_i$ by $\hat{\mathbf{x}}_i$. The average error of this representation over the whole dataset will be

$$\begin{aligned}\frac{1}{N}\sum_i \|\hat{\mathbf{x}}_i - \tilde{\mathbf{x}}_i\|^2 &= \text{Error in components that are preserved} + \\ &\quad \text{Error in components that are zeroed} \\ &= s\sigma^2 + \sum_{j=s+1}^{d}\tilde{\lambda}_u.\end{aligned}$$

Now if, for $j > s$, $\tilde{\lambda}_j < \sigma^2$, this error is smaller than $d\sigma^2$. We don't know which s guarantees this unless we know σ^2 and $\tilde{\lambda}_j$ which often doesn't happen. But it's usually possible to make a safe choice, and so **smooth** the data by reducing noise. This smoothing works because the components of the data are correlated. So the best estimate of each component of a data item is likely not the measurement – it's a prediction obtained from all measurements. The projection onto principal components is such a prediction.

Remember this: *Given a d dimensional dataset where data items have had independent random noise added to them, representating each data item on $s < d$ principal components can result in a representation which is on average closer to the true underlying data than the original data items. The choice of s is application dependent.*

6.2 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 6.5 shows the mean spectral reflectance of this dataset, and Figure 6.5 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 6.5. This means that there is a mean function $r(\lambda)$ and k functions $\phi_m(\lambda)$ such that, for any $\rho(\lambda)$,

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

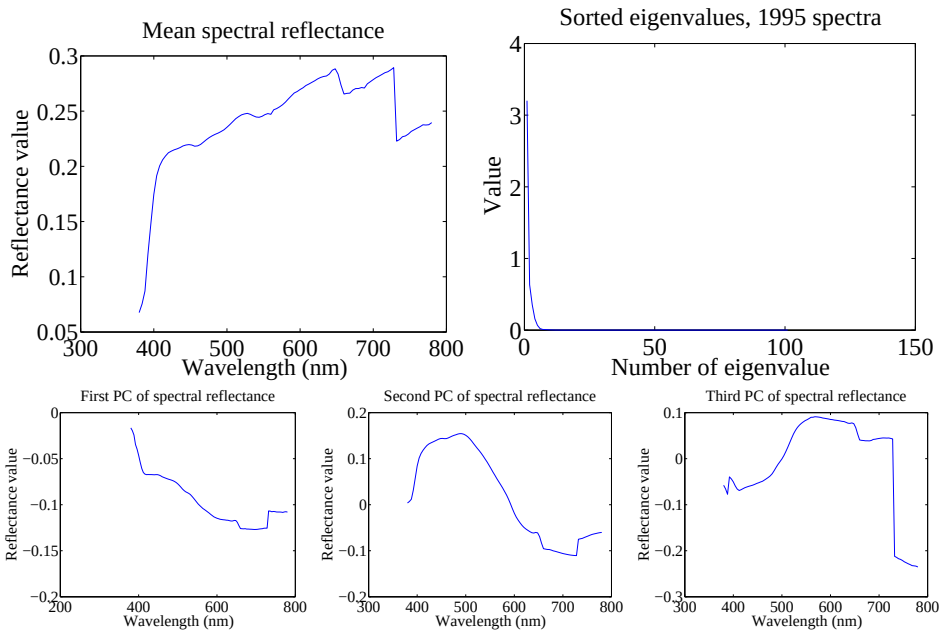


FIGURE 6.5: *On the* **top left**, *the mean spectral reflectance of a dataset of 1995 spectral reflectances, collected by Kobus Barnard (at* http://www.cs.sfu.ca/~colour/data/*). On the* **top right**, *eigenvalues of the covariance matrix of spectral reflectance data, from a dataset of 1995 spectral 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 (***top left***), gives a good representation of the dataset.*

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 6.6). 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 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.

Figures **??** and **??** illustrate the smoothing process. I know neither the noise process nor the true variances (this is quite usual), so I can't say which smoothed

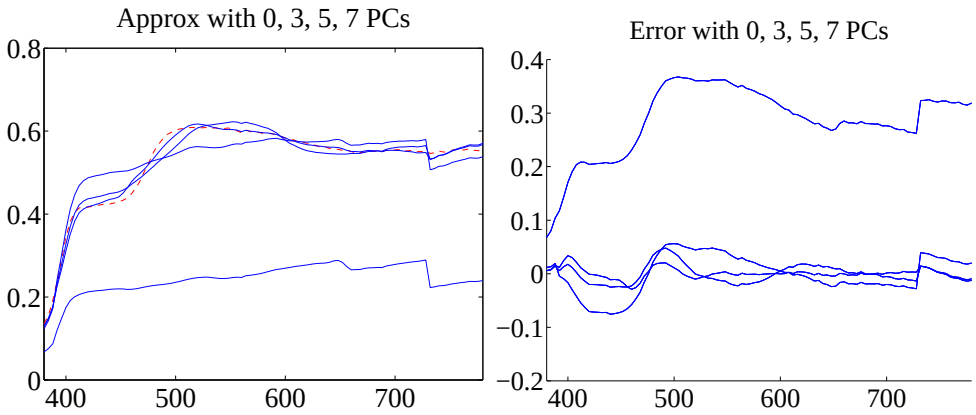


FIGURE 6.6: *On the* **left**, *a spectral reflectance curve (dashed) and approximations using the mean, the mean and 3 principal components, the mean and 5 principal components, and the mean and 7 principal components. Notice the mean is a relatively poor approximation, but as the number of principal components goes up, the mean squared distance between measurements and principal component representation falls rather quickly. On the* **right** *is is this distance for these approximations. A projection onto very few principal components suppresses local wiggles in the data* unless *very many data items have the same wiggle in the same place. As the number of principal components increases, the representation follows the measurements more closely. The best estimate of each component of a data item is likely not the measurement – it's a prediction obtained from all measurements. The projection onto principal components is such a prediction, and you can see the smoothing effects of principal components analysis in these plots. Figure plotted from a dataset of 1995 spectral reflectances, collected by Kobus Barnard (at* http://www.cs.sfu.ca/~colour/data/*).*

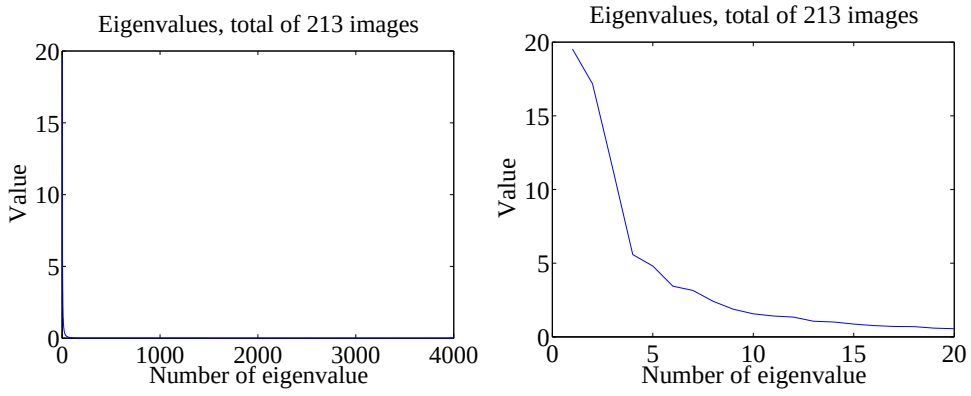


FIGURE 6.7: *On the* **left**,*the eigenvalues of the covariance of the Japanese facial expression dataset; there are 4096, so it's hard to see the curve (which is packed to the left). On the* **right**, *a zoomed version of the curve, showing how quickly the values of the eigenvalues get small.*

FIGURE 6.8: *The mean and first 16 principal components of the Japanese facial expression dataset.*

representation is best. Each figure shows four spectral reflectances and their representation on a set of principal components. Notice how, as the number of principal components goes up, the measurements and the representation get closer together. This *doesn't* necessarily mean that more principal components are better – the measurement itself may be noisy. Notice also how representations on few principal components tend to suppress small local "wiggles" in the spectral reflectance. They are suppressed because these patterns tend not to appear in the same place in all spectral reflectances, so the most important principal components tend not to have them. The noise model tends to produce these patterns, so that the representation on a small set of principal components may well be a more accurate estimate of the spectral reflectance than the measurement is.

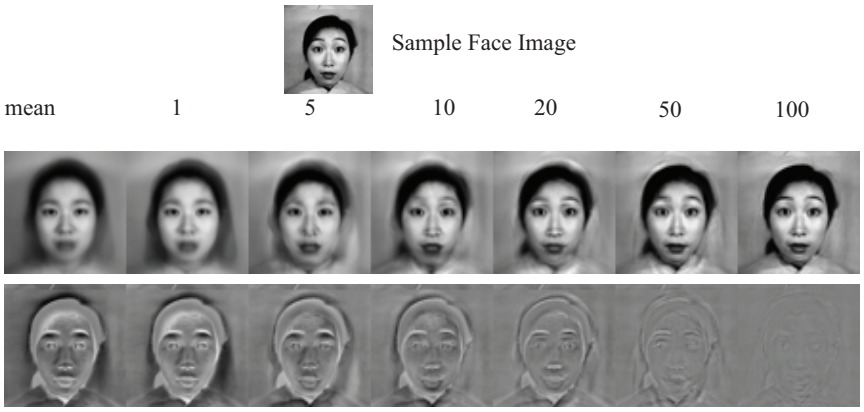


FIGURE 6.9: *Approximating a face image by the mean and some principal components; notice how good the approximation becomes with relatively few components.*

6.3 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 6.7; 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 6.8. Principal components give quite good approximations to real images (figure 6.9).

The principal components sketch out the main kinds of variation in facial expression. Notice how the mean face in Figure 6.8 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 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.

A representation on enough principal components results in pixel values that are closer to the true values than the measurements (this is one sense of the word "smoothing"). Another sense of the word is blurring. Irritatingly, blurring reduces noise, and some methods for reducing noise, like principal components, also blur (figure 6.9). But this doesn't mean the resulting images are better *as images*. In fact, you don't have to blur an image to smooth it. Producing images that are both accurate estimates of the true values and look like sharp, realistic images requires quite substantial technology, beyond our current scope.

6.4 YOU SHOULD

6.4.1 remember these definitions:

6.4.2 remember these terms:

6.4.3 remember these facts:

6.4.4 remember these procedures:

6.4.5 be able to:

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

PROBLEMS

6.1. Using the notation of the chapter, show that

$$w_{ij} = r_i^{(j)} = (\mathbf{x}_i - \mathsf{mean}\,(\{\mathbf{x}\}))^T \mathbf{u}_j.$$

6.2. We have N d-dimensional data items forming a dataset $\{\mathbf{x}\}$. We translate this dataset to have zero mean, compute

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

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

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

Choose some $s < d$, take each data point $\mathbf{r}_i$ and replace the last $d - s$ components with 0. Call the resulting data item $\mathbf{p}_i$.

(a) Show that

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

(b) Sort the eigenvalues of $\mathsf{Covmat}\,(\{\mathbf{x}\})$ in descending order, and write λ_i for the i'th (so that $\lambda_1 \geq \lambda_2 \ldots \geq \lambda_N$). Show that

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

6.3. You have a dataset of N vectors $\mathbf{x}_i$ in d-dimensions, stacked into a matrix $\mathcal{X}$. This dataset has zero mean. You would like to determine the principal component of this dataset corresponding to the largest eigenvalue of its covariance. Write $\mathbf{u}$ for this principal component.

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

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

Show that

$$\| \mathcal{A} \|_F{}^2 = \mathrm{Trace}(\mathcal{A}\mathcal{A}^T)$$

(b) Show that

$$\mathrm{Trace}(\mathcal{A}\mathcal{B}) = \mathrm{Trace}(\mathcal{B}\mathcal{A}).$$

I have found this fact worth remembering. It may help to remember the trace is defined only for square matrices.

(c) Show that, if $\mathbf{u}$ and $\mathbf{w}$ together minimize

$$\| \mathcal{X} - \mathbf{w}\mathbf{u}^T \|_F{}^2$$

then

$$\begin{aligned} (\mathbf{w}^T\mathbf{w})\mathbf{u} &= \mathcal{X}^T\mathbf{w} \\ (\mathbf{u}^T\mathbf{u})\mathbf{w} &= \mathcal{X}\mathbf{u} \end{aligned}$$

Do this by differentiating and setting to zero; the text of the NIPALS section should help.

(d) $\mathbf{u}$ is a unit vector – why?

(e) Show that

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

and so that, if $\mathbf{u}$ minimizes the Frobenius norm as above, it must be some eigenvector of $\mathsf{Covmat}(\{x\})$.

(f) Show that, if $\mathbf{u}$ is a unit vector, then

$$\mathrm{Trace}(\mathbf{u}\mathbf{u}^T) = 1$$

(g) Assume that $\mathbf{u}$, $\mathbf{w}$ satisfy the equations for a minimizer, above, then show

$$\begin{aligned} \|\mathcal{X} - \mathbf{w}\mathbf{u}^T\|_F^2 &= \mathrm{Trace}(\mathcal{X}^T\mathcal{X} - \mathbf{u}(\mathbf{w}^T\mathbf{w})\mathbf{u}^T) \\ &= \mathrm{Trace}(\mathcal{X}^T\mathcal{X}) - (\mathbf{w}^T\mathbf{w}) \end{aligned}$$

(h) Use the information above to argue that if $\mathbf{u}$ and $\mathbf{w}$ together minimize

$$\|\mathcal{X} - \mathbf{w}\mathbf{u}^T\|_F^2$$

then $\mathbf{u}$ is the eigenvector of $\mathcal{X}^T\mathcal{X}$ corresponding to the largest eigenvalue.

6.4. You have a dataset of N vectors $\mathbf{x}_i$ in d-dimensions, stacked into a matrix $\mathcal{X}$. This dataset has zero mean. You would like to determine the principal component of this dataset corresponding to the largest eigenvalue of its covariance. Write $\mathbf{u}$ for this principal component. Assume that each data item $\mathbf{x}_i$ is noisy. We use a simple noise model. Write $\tilde{\mathbf{x}}_i$ for the true underlying value of the data item, and ξ_i for the value of a normal random variable with zero mean and covariance $\sigma^2\mathcal{I}$. Then we use the model

$$\mathbf{x}_i = \tilde{\mathbf{x}}_i + \xi_i$$

We will assume that $\mathsf{mean}(\{\xi\}) = \mathbf{0}$ and $\mathsf{Covmat}(\{\xi\}) = \sigma^2\mathcal{I}$.

(a) Notice that the noise is independent of the dataset. This means that $\mathsf{mean}\left(\left\{\mathbf{x}\xi^T\right\}\right) = \mathsf{mean}(\{\mathbf{x}\})\mathsf{mean}\left(\left\{\xi^T\right\}\right) = 0$. Show that

$$\mathsf{Covmat}(\{\mathbf{x}\}) = \tilde{\Sigma} + \sigma^2\mathcal{I}.$$

(b) Show that if $\mathcal{U}$ diagonalizes $\mathsf{Covmat}(\{\mathbf{x}\})$, it will also diagonalize $\tilde{\Sigma}$.

PROGRAMMING EXERCISES

6.5. 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

6.6. 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.

6.7. 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?

6.8. 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?

6.9. 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?

6.10. 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. We will investigate the use of principal components to smooth data.

(a) Ignore the species names, so you should have 150 data items with four measurements each. For each value in $\{0.1, 0.2, 0.5, 1\}$, form a dataset by adding an independent sample from a normal distribution with this standard deviation to *each* entry in the original dataset. Now for each value, plot the mean-squared-error between the original dataset and an expansion onto 1, 2, 3, and 4 principal components. You should see that, as the noise gets larger, using fewer principal components gives a more accurate estimate of the original dataset (i.e. the one without noise).

(b) We will now try the previous subexercise with a very much different noise model. For each of $w = \{10, 20, 30, 40\}$, construct a mask matrix each of whose entries is a sample of a binomial random variable with probability $p = 1 - w/600$ of turning up 1. This matrix should have about w zeros in it. Ignore the species names, so you should have 150 data items with four measurements each. Now form a new dataset by multiplying each location in the original dataset by the corresponding mask location (so you are randomly setting a small set of measurements to zero). Now for each value of w, plot the mean-squared-error between the original dataset and an expansion onto 1, 2, 3, and 4 principal components. You should see that, as the noise gets larger, using fewer principal components gives

a more accurate estimate of the original dataset (i.e. the one without noise).

CHAPTER 7

Low Rank Approximations

A principal components analysis models high-dimensional data points with an accurate, low-dimensional, model. Form a data matrix from the approximate points; this has low rank (because the model is low dimensional) *and* it's close to the original data matrix (because the model is accurate). Looking for a low rank model of a data matrix is productive.

Assume we have $\mathcal{X}$, with rank d, and we wish to produce $\mathcal{X}_s$ such that (a) the rank of $\mathcal{X}_s$ is s (which is less than d) and (b) such that $\|\mathcal{X} - \mathcal{X}_s\|^2$ is minimized. The resulting $\mathcal{X}_s$ is called a **low rank approximation** to $\mathcal{X}$. Producing a low rank approximation is a straightforward application of the singular value decomposition (SVD).

We have already seen examples of useful low rank approximations. NIPALS – which is actually a form of partial SVD – produces a rank one approximation to a matrix (check this point if you're uncertain). A new, and useful, important application is to use a low rank approximation to make a low-dimensional map of a high dimensional dataset (section **??**).

The link between principal components analysis and low rank approximation suggests (correctly) that you can use a low rank approximation to smooth and suppress noise. Smoothing is extremely powerful, and section **??** describes an important application. The count of words in a document gives a rough representation of the document's meaning. But there are many different words an author could use for the same idea ("spanner" or "wrench", say), and this effect means that documents with quite similar meaning could have quite different word counts. Word counts can be smoothed very effectively with a low rank approximation to an appropriate matrix. There are two quite useful applications. First, this low rank approximation yields quite good measures of how similar documents are. Second, the approximation can yield a representation of the underlying meaning of a word which is useful in dealing with unfamiliar words.

7.1 THE SINGULAR VALUE DECOMPOSITION

For any $m \times p$ matrix $\mathcal{X}$, it is possible to obtain a decomposition

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

where $\mathcal{U}$ is $m \times m$, $\mathcal{V}$ is $p \times p$, and Σ is $m \times p$ and is diagonal. The diagonal entries of Σ are non-negative. Both $\mathcal{U}$ and $\mathcal{V}$ are orthonormal (i.e. $\mathcal{U}\mathcal{U}^T = \mathcal{I}$ and $\mathcal{V}\mathcal{V}^T = \mathcal{I}$). This decomposition is known as the **singular value decomposition**, almost always abbreviated to **SVD**).

If you don't recall what a diagonal matrix looks like when the matrix *isn't* square, it's simple. All entries are zero, except the i, i entries for i in the range 1 to $\min(m, p)$. So if Σ is tall and thin, the top square is diagonal and everything else is zero; if Σ is short and wide, the left square is diagonal and everything else is zero.

The terms on the diagonal of Σ are usually called the **singular values**. There is a significant literature on methods to compute the SVD efficiently, accurately and at large scale, which we ignore: any decent computing environment should do this for you if you find the right function. Read the manual for your environment.

Procedure: 7.1 *Singular Value Decomposition*

Given a matrix $\mathcal{X}$, any halfway decent numerical linear algebra package or computing environment will produce a decomposition

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

and $\mathcal{U}$ and $\mathcal{V}$ are both orthonormal, Σ is diagonal with non-negative entries. Most environments that can do an SVD can be persuaded to provide the columns of $\mathcal{U}$ and rows of $\mathcal{V}^T$ corresponding to the k largest singular values.

There are many SVD's for a given matrix, because you could reorder the singular values and then reorder $\mathcal{U}$ and $\mathcal{V}$. We will always assume that the diagonal entries in Σ go from largest to smallest as one moves down the diagonal. In this case, the columns of $\mathcal{U}$ and the rows of $\mathcal{V}^T$ corresponding to non-zero diagonal elements of Σ are unique.

Notice that there is a relationship between forming an SVD and diagonalizing a matrix. In particular, $\mathcal{X}^T\mathcal{X}$ is symmetric, and it can be diagonalized as

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

Similarly, $\mathcal{X}\mathcal{X}^T$ is symmetric, and it can be diagonalized as

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

Remember this: *A singular value decomposition (SVD) decomposes a matrix $\mathcal{X}$ as $\mathcal{X} = \mathcal{U}\Sigma\mathcal{V}^T$ where $\mathcal{U}$ is $m \times m$, $\mathcal{V}$ is $p \times p$, and Σ is $m \times p$ and is diagonal. The diagonal entries of Σ are non-negative. Both $\mathcal{U}$ and $\mathcal{V}$ are orthonormal. The SVD of $\mathcal{X}$ yields the diagonalization of $\mathcal{X}^T\mathcal{X}$ and the diagonalization of $\mathcal{X}\mathcal{X}^T$.*

7.1.1 SVD and PCA

Now assume we have a dataset with zero mean. 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. The covariance matrix is

$$\text{Covmat}\left(\{X\}\right) = \frac{1}{N}\mathcal{X}^T\mathcal{X}$$

(zero mean, remember). Form the SVD of $\mathcal{X}$, to get

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

But we have $\mathcal{X}^T\mathcal{X} = \mathcal{V}\Sigma^T\Sigma\mathcal{V}^T$ so that

$$\text{Covmat}\left(\{X\}\right)\mathcal{V} = \frac{1}{N}(\mathcal{X}^T\mathcal{X})\mathcal{V} = \mathcal{V}\frac{\Sigma^T\Sigma}{N}$$

and $\Sigma^T\Sigma$ is diagonal. By pattern matching, the columns of $\mathcal{V}$ contains the principal components of $\mathcal{X}$, and

$$\frac{\Sigma^T\Sigma}{N}$$

are the variances on each component. All this means we can read the principal components of a dataset of the SVD of that 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. Remember, these are the columns of $\mathcal{V}$ – it's easy to get mixed up about $\mathcal{V}$ and $\mathcal{V}^T$ here.

We have seen NIPALS as a way of extracting some principal components from a data matrix. In fact, NIPALS is a method to recover a partial SVD of $\mathcal{X}$. Recall that NIPALS produces a vector $\mathbf{u}$ and a vector $\mathbf{w}$ so that $\mathbf{w}\mathbf{u}^T$ is as close as possible to $\mathcal{X}$, and $\mathbf{u}$ is a unit vector. By pattern matching, we have that

- $\mathbf{u}^T$ is the row of $\mathcal{V}^T$ corresponding to the largest singular value;
- $\frac{\mathbf{w}}{\|\mathbf{w}\|}$ is the column of $\mathcal{U}$ corresponding to the largest singular value;
- $\|\mathbf{w}\|$ is the largest singular value.

It is easy to show that if you use NIPALS to extract several principal components, you will get several rows of $\mathcal{V}^T$, several columns of $\mathcal{U}$, and several singular values. Be careful, however: this isn't an efficient or accurate way to extract many singular values, because numerical errors accumulate. If you want a partial SVD with many singular values, you should be searching for specialist packages, not making your own.

Remember this: *Assume $\mathcal{X}$ has zero mean. Then the SVD of $\mathcal{X}$ yields the principal components of the dataset represented by this matrix. NIPALS is a method to recover a partial SVD of $\mathcal{X}$*

7.1.2 SVD and Low Rank Approximations

Assume we have $\mathcal{X}$, with rank d, and we wish to produce $\mathcal{X}_s$ such that (a) the rank of $\mathcal{X}_s$ is s (which is less than d) and (b) such that $\|\mathcal{X} - \mathcal{X}_s\|^2$ is minimized. An SVD will yield $\mathcal{X}_s$. Take the SVD to get $\mathcal{X} = \mathcal{U}\Sigma\mathcal{V}^T$. Now write Σ_s for the matrix obtained by setting all but the s largest singular values in Σ to 0. We have that

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

It is obvious that $\mathcal{X}_s$ has rank s. You can show (exercises) that $\|\mathcal{X} - \mathcal{X}_s\|^2$ is minimized, by noticing that $\|\mathcal{X} - \mathcal{X}_s\|^2 = \|\Sigma - \Sigma_s\|^2$.

There is one potential point of confusion. There are a lot of zeros in Σ_s, and they render most of the columns of $\mathcal{U}$ and rows of $\mathcal{V}^T$ irrelevant. In particular, write $\mathcal{U}_s$ for the $m \times s$ matrix consisting of the first s columns of $\mathcal{U}$, and so on; and write $\Sigma_s^{(s)}$ for the $s \times s$ submatrix of Σ_s with non-zero diagonal. Then we have

$$\mathcal{X}_s = \mathcal{U}\Sigma_s\mathcal{V}^T = \mathcal{U}_s\Sigma_s^{(s)}(\mathcal{V}_s)^T$$

and it is quite usual to switch from one representation to the other without comment. I try not to do this, but it's quite common practice.

7.1.3 Smoothing with the SVD

As we have seen, principal components analysis can smooth noise in the data matrix (section 18). That argument was for one particular kind of noise, but experience shows that PCA can smooth other kinds of noise (there is an example in the exercises for chapter 6). This means that the entries of a data matrix can be smoothed by computing a low-rank approximation of $\mathcal{X}$.

I have already shown that PCA can smooth data. In PCA (procedure 6.1), the d dimensional data point $\mathbf{x}_i$ is represented by

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

where $\mathbf{u}_j$ are the principal components. A low rank approximation represents the i'th row of $\mathcal{X}$ (which is $\mathbf{x}_i^T$) as

$$\hat{\mathbf{x}}_i{}^T = \sum_{j=1}^{r} w_{ij}\mathbf{v}_j^T$$

where $\mathbf{v}_j^T$ is a row of $\mathcal{V}^T$ (obtained from the SVD) and where w_{ij} are weights that can be computed from the SVD. In each case, the data point is represented by

a projection onto a low dimensional space, so it is fair to conclude the SVD can smooth something.

Just like smoothing with a PCA, smoothing with an SVD works for a wide range of noise processes. In one very useful example, each component of the data might be a count. For concreteness, let the entries be counts of roadkill species per mile of highway. Each row would correspond to a species, each column to a particular mile. Counts like this would typically be noisy, because you see rare species only occasionally. At least for rare species, the count for most miles would be 0, but occasionally, you would count 1. The 0 is too low a per-mile estimate, and the 1 is too high, but one doesn't see a fraction of a roadkill. Constructing a low rank approximation tends to lead to better estimates of the counts.

Missing data is a particularly interesting form of noise - the noise process deletes entries in the data matrix - and low rank approximations are quite effective at dealing with this. Assume you know most, but not all, entries of $\mathcal{X}$. You would like to build an estimate of the whole matrix. If you expect that the true whole matrix has low rank, you can compute a low rank approximation to the matrix. For example, the entries in the data matrix are scores of how well a viewer liked a film. Each row of the data matrix corresponds to one viewer; each column corresponds to one film. At useful scales, most viewers haven't seen most films, so most of the data matrix is missing data. However, there is good reason to believe that users are "like" each other – the rows are unlikely to be independent, because if two viewers both like (say) horror movies they might very well also both dislike (say) documentaries. Films are "like" each other, too. Two horror movies are quite likely to be liked by viewers who like horror movies but dislike documentaries. All this means that the rows (resp. columns) of the true data matrix are very likely to be highly dependent. More formally, the true data matrix is likely to have low rank. This suggests using an SVD to fill in the missing values.

Numerical and algorithmic questions get tricky here. If the rank is very low, you could use NIPALS to manage the question of missing entries. If you are dealing with a larger rank, or many missing values, you need to be careful about numerical error, and you should be searching for specialist packages, not making your own with NIPALS.

Remember this: *Taking an SVD of a data matrix usually produces a smoothed estimate of the data matrix. Smoothing is guaranteed to be effective if the entries are subject to additive, zero-mean, independent gaussian noise, but often works very well if the entries are noisy counts. Smoothing can be used to fill in missing values, too.*

7.2 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.

7.2.1 Choosing Low D Points using High D Distances

We will plot the high dimensional point $\mathbf{x}_i$ at $\mathbf{y}_i$, which is an s-dimensional vector (almost always, s will be 2 or 3). 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 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 $\mathbf{y}_i$ to make

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

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 assume that the mean of the points is zero, so

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

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

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

Now you can show that

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

I now argue that, to make $\mathcal{D}^{(2)}(\mathbf{y})$ is close to $\mathcal{D}^{(2)}(\mathbf{x})$, it is enough to choose $\mathbf{y}_i$ so that $\mathcal{Y}\mathcal{Y}^T$ close to $\mathcal{X}\mathcal{X}^T$. Proving this will take us out of our way unnecessarily, so I omit a proof.

7.2.2 Using a Low Rank Approximation to Factor

We need to find a set of $\mathbf{y}_i$ so that (a) the $\mathbf{y}_i$ are s dimensional and (b) $\mathcal{Y}$ (the matrix made by stacking the $\mathbf{y}_i$) minimizes the distance between $\mathcal{Y}\mathcal{Y}^T$ and $\mathcal{X}\mathcal{X}^T$. Notice that $\mathcal{Y}\mathcal{Y}^T$ must have rank s.

Now form an SVD of $\mathcal{X}$, to get

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

Recall $\Sigma_s^{(s)}$ is the $s \times s$ submatrix of Σ_s with non-zero diagonal, $\mathcal{U}_s$ is the $m \times s$ matrix consisting of the first s columns of $\mathcal{U}$, and so on. Consider

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

We have that $\mathcal{X}_s\mathcal{X}_s^T$ is the closest rank s approximation to $\mathcal{X}\mathcal{X}^T$. The rows of $\mathcal{X}_s$ are d-dimensional, so it isn't the matrix we seek. But

$$\mathcal{X}_s\mathcal{X}_s^T = (\mathcal{U}_s\Sigma_s\mathcal{V}_s^T)(\mathcal{V}_s\Sigma_s\mathcal{U}_s^T)$$

and $\mathcal{V}_s^T\mathcal{V}_s$ is the $s \times s$ identity matrix (exercises). This means that

$$\mathcal{Y} = \mathcal{U}_s\Sigma_s$$

is the matrix we seek. We can obtain $\mathcal{Y}$ even if we don't know $\mathcal{X}$. It is enough to know $\mathcal{X}\mathcal{X}^T$. This is because

$$\mathcal{X}\mathcal{X}^T = (\mathcal{U}\Sigma\mathcal{V}^T)(\mathcal{V}\Sigma\mathcal{U}^T) = \mathcal{U}\Sigma^2\mathcal{U}^T$$

so diagonalizing $\mathcal{X}\mathcal{X}^T$ is enough. 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**. I pick up this theme in Chapter 57, which demonstrates more sophisticated methods for the problem.

Procedure: 7.2 *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 s 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. Notice that you need only the top s eigenvalues and their eigenvectors, and many packages can extract these rather faster than constructing all.
- Choose s, the number of dimensions you wish to represent. Form Λ_s, the top left $s \times s$ block of Λ. Form $\Lambda_s^{(1/2)}$, whose entries are the positive square roots of Λ_s. Form $\mathcal{U}_s$, the matrix consisting of the first s columns of $\mathcal{U}$.

Then

$$\mathcal{Y} = \mathcal{U}_s\Sigma_s = \begin{bmatrix} \mathbf{v}_1, \\ \ldots, \\ \mathbf{v}_N \end{bmatrix}$$

is the set of points to plot.

7.2.3 Example: Mapping with Multidimensional Scaling

Multidimensional scaling gets positions (the $\mathcal{Y}$ of section 7.2.1) from distances (the $\mathcal{D}^{(2)}(\mathbf{x})$ of section 7.2.1). 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 7.1).

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 7.2, 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

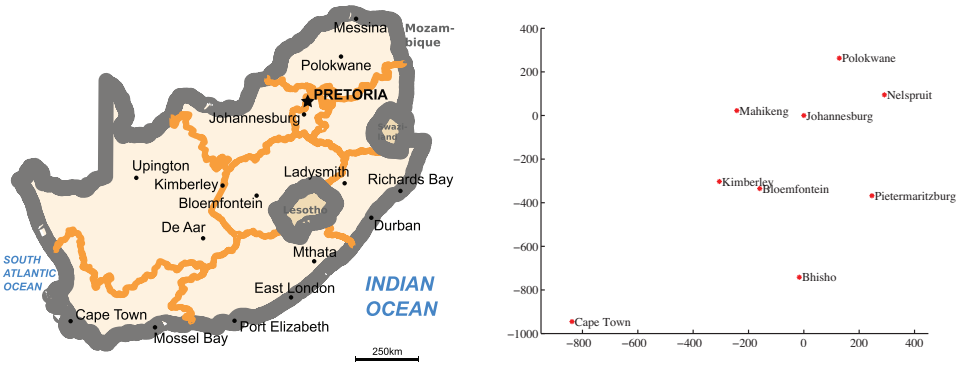


FIGURE 7.1: *On the* **left**, *a public domain map of South Africa, obtained from* http://commons.wikimedia.org/wiki/File:Map_of_South_Africa.svg, *and edited to remove surrounding countries. On the* **right**, *the locations of the cities inferred by multidimensional scaling, rotated, translated and 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).*

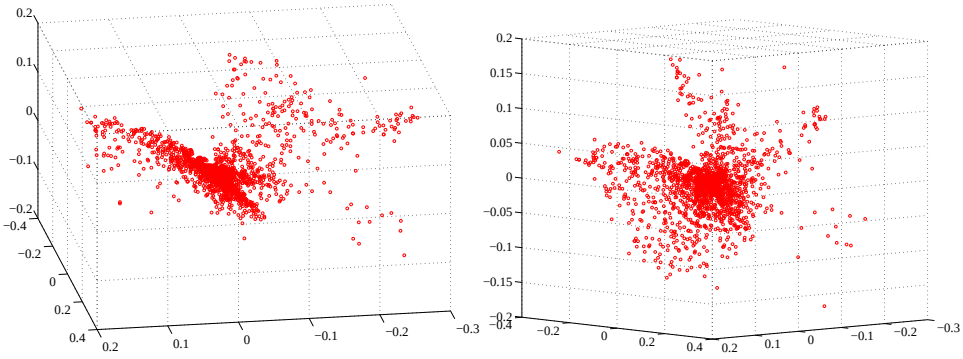


FIGURE 7.2: *Two views of the spectral data of section 6.2, plotted as a scatter plot by applying principal coordinate analysis to obtain a 3D set of points. Notice that the data spreads out in 3D, but seems to lie on some structure; it certainly isn't a single blob. This suggests that further investigation would be fruitful.*

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 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

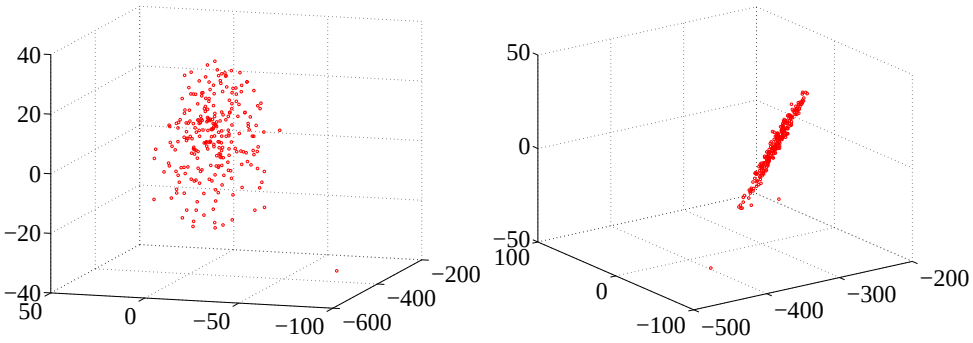


FIGURE 7.3: *Two views of a multidimensional scaling to three dimensions of the height-weight dataset. Notice how the data seems to lie in a flat structure in 3D, with one outlying data point. This means that the distances between data points can be (largely) explained by a 2D representation.*

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 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 $\mathcal{Y}$ 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.

7.3 EXAMPLE: TEXT MODELS AND LATENT SEMANTIC ANALYSIS

It is really useful to be able to measure the similarity between two documents, but it remains difficult to build programs that understand natural language. Experience shows that very simple models can be used to measure similarity between documents without going to the trouble of building a program that understands their content. Here is a representation that has been successful. Choose a vocabulary (a list of different words), then represent 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.

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 isn't always helpful, and 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.

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 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.

7.3.1 The Cosine Distance

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, so the distance from the new document's word vector to other word vectors would have changed a lot. But the meaning of the new document wouldn't have changed. One way to overcome this nuisance is to normalize the vector of word counts in some way. It is usual to normalize the word counts by the magnitude of the count vector.

The distance between two word count vectors, normalized to be unit vectors, is

$$\left\| \frac{\mathbf{x}_i}{\|\mathbf{x}_i\|} - \frac{\mathbf{x}_j}{\|\mathbf{x}_j\|} \right\|^2 = 2 - 2\frac{\mathbf{x}_i^T \mathbf{x}_j}{\|\mathbf{x}_i\|\|\mathbf{x}_j\|}.$$

The expression

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

is often known as the **cosine distance** between documents. 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. Experience has shown that a very effective measure of the similarity of documents i and j is their cosine distance.

7.3.2 Smoothing Word Counts

Measuring the cosine distance for word counts has problems. We have seen one important problem already: if one document uses "car" and the other "auto", the two might be quite similar and yet have cosine distance that is close to zero. Remember, cosine distance close to zero suggests they're far apart. This is because the word counts are misleading. If you count, say, "car" once, you should have a non-zero count for "auto" as well. You could regard the zero count for "auto" as noise. This suggests smoothing word counts.

Arrange the 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; this will smooth the word counts. Take an SVD of $\mathcal{X}$, yielding

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

Write Σ_r for the matrix obtained by setting all but the r largest singular values in Σ to 0, and construct

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

You should think of $\mathcal{X}^{(r)}$ as a smoothing of $\mathcal{X}$. The argument I used to justify seeing principal components as a smoothing method (section 57) doesn't work here, because the noise model doesn't apply. But a qualitative argument supports the idea that we are smoothing. 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. The counts in $\mathcal{X}^{(r)}$ are better estimates of what true word counts should be than one can obtain by simply counting words, because they take into account correlations between words.

Here is one way to think about this. Because word vectors in $\mathcal{X}^{(r)}$ are compelled to occupy a low dimensional space, counts "leak" between words with similar meanings. This happens because most documents that use "car" will tend to have many *other* in common with most documents that use "auto". For example, it's highly unlikely that every document that uses "car" instead of "auto" also uses "spanner" instead of "wrench", and vice-versa. A good low dimensional representation will place documents that use a large number of words with similar frequencies close together, even if they use some words with different frequencies; in turn, a

document that uses "auto" will likely have the count for that word go down somewhat, and the count for "car" go up. Recovering information from the SVD of $\mathcal{X}$ is referred to as **latent semantic analysis**.

We have that

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

so each $\mathbf{x}_i^{(r)}$ is a weighted sum of the first r rows of $\mathcal{V}^T$.

A natural representation for the i'th document is

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

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 (it's $2 -$ cosine distance.

A key application for latent semantic analysis is in search. 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 low dimensional unit vector $\mathbf{d}_q$ for the query word vector, then finding nearby documents by an approximate nearest neighbor search on the document dataset. Computing a $\mathbf{d}_q$ 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.

Now $\mathcal{V}$ is orthonormal, so $\mathbf{v}_k^T\mathbf{v}_m$ is 1 for $k = m$, and zero otherwise. This means that

$$(\mathbf{x}_i^{(r)})^T(\mathbf{x}_j^{(r)}) = (\sum_{k=1}^{r} a_{ik}\mathbf{v}_k^T)(\sum_{m=1}^{r} a_{jm}\mathbf{v}_m) = \sum_{k=1}^{r} a_{ik}a_{jk}.$$

But all the terms we are interested are inner products between document vectors. In turn, this means we could adopt a low dimensional representation for documents explicitly, and so, for example, use

$$\mathbf{d}_i = \frac{[a_{i1}, \ldots, a_{ir}]}{\sum_k a_{ik}^2}.$$

This representation has a much lower dimension than the normalized smoothed document vector, but contains exactly the same information.

7.3.3 Mapping NIPS Documents

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 compute smoothed word counts in documents, and to map documents.

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.

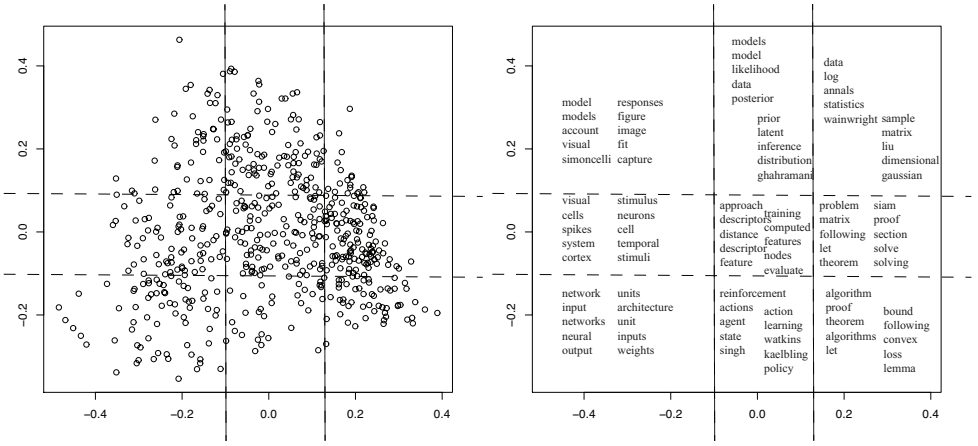


FIGURE 7.4: *On the* **left**, *a multidimensional scaling mapping the NIPS documents into 2D. The distances between points represent (as well as an MDS can) the distances between normalized smoothed word counts. I have plotted every 10'th document, to avoid crowding the plot. Superimposed on the figure is a grid, dividing each coordinate at the 33% (resp. 66%) quantile. On the* **right**, *I have plotted the 10 words most strongly correlated with a document appearing in the corresponding grid block (highest correlation at top left in block, lowest in bottom right). Each block has quite different sets of words, but there is evidence that: changes in the coordinates result in changes in document content; the dataset still has proper names in it, though insiders might notice the names are in sensible places; the horizontal coordinate seems to represent a practical-conceptual axis; and increasing values of the vertical coordinate seems to represent an increasingly statistical flavor. This is (rather rough) evidence that distances between smoothed normalized word counts do capture aspects of meaning.*

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. Figure 7.5 shows a multidimensional scaling of distances between normalized smoothed word counts. You should notice that documents are fairly evenly spread over the space. To give some meaning to the space, I have plotted the 10 words most strongly correlated with a document appearing in the corresponding grid block (highest correlation at top left in block, lowest in bottom right). Notice how the word clusters shade significantly across the coordinates. This is (rather rough) evidence that distances between smoothed normalized word counts do capture aspects of meaning.

7.3.4 Obtaining the Meaning of Words

It is difficult to know what a word means by looking at it, unless you have seen it before or it is an inflected version of a word you have seen before. A high percentage of readers won't have seen "peridot", "incarnadine", "whilom", or "numbat"

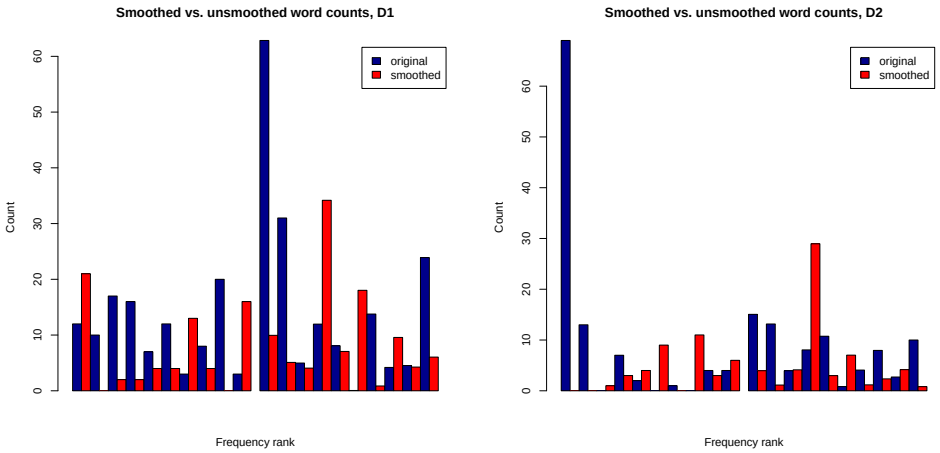


FIGURE 7.5: *Unsmoothed and smoothed word counts for two different documents, where smoothing is by LSA to 1000 intermediate dimensions. Each figure shows one document; the blue bars are unsmoothed counts and the red bars are smoothed counts. The figure shows the counts for the 100 words that appear most frequently in the whole dataset, ordered by the rank of the word count (most common word first, etc.) Notice that generally, large counts tend to go down, and small counts tend to go up, as one would expect.*

before. If any of these are unfamiliar, simply looking at the letters isn't going to tell you what they mean. This means that unfamiliar words are quite different from unfamiliar pictures. If you look at a picture of something you haven't seen before, you're likely to be able to make some sensible guesses as to what it is like (how you do this remains very poorly understood; but that you can do this is everyday experience).

We run into unfamiliar words all the time, but the words around them seem to help us figure out what the unfamiliar words mean. As a demonstration, you should find these texts, which I modified from sentences found on the internet, helpful

- Peridot: "A sweet row of Peridot sit between golden round beads, strung from a delicate plated chain" (suggesting some form of decorative stone).
- Incarnidine: "A spreading stain incarnadined the sea" (a color description of some sort).
- Whilom: "Portions of the whilom fortifications have been converted into promenades." (a reference to the past).
- Numbat: "They fed the zoo numbats modified cat chow with crushed termite" (some form of animal, likely not vegetarian, and perhaps a picky eater).

This is a demonstration of a general point. Words near a particular word give strong and often very useful hints to that word's meaning, an effect known as

distributional semantics. Latent semantic analysis offers a way to exploit this effect to estimate representations of word meaning.

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 is a smoothed count of 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 $\mathbf{n}_j$. This distance gives quite a good representation of word meaning, because two words that are close in this distance will tend to appear in the same documents. For example, "auto" and "car" should be close. As we saw above, the smoothing will tend to reduce counts of "auto" and increase counts of "car" for documents that have only "auto", and so on. In turn, this means that "auto" will tend to appear in the same documents as "car", meaning that the distance between their normalized smoothed counts should be small.

We have that

$$(\mathbf{w}_i^{(r)}) = \sum_{k=1}^{r} (\sigma_k v_{ki}) \mathbf{u}_k = \sum_{k=1}^{r} b_{ik} \mathbf{u}_k$$

so each $\mathbf{w}_i^{(r)}$ is a weighted sum of the first r columns of $\mathcal{U}$.

Now $\mathcal{U}$ is orthonormal, so $\mathbf{u}_k^T \mathbf{u}_m$ is 1 for $k = m$, and zero otherwise. This means that

$$(\mathbf{w}_i^{(r)})^T (\mathbf{w}_j^{(r)}) = (\sum_{k=1}^{r} b_{ik} \mathbf{u}_k^T)(\sum_{m=1}^{r} b_{jm} \mathbf{u}_m) = \sum_{k=1}^{r} b_{ik} b_{jk}.$$

But all the terms we are interested are inner products between word vectors. In turn, this means we could adopt a low dimensional representation for words explicitly, and so, for example, use

$$\mathbf{n}_i = \frac{[b_{i1}, \ldots, b_{ir}]}{\sum_k b_{ik}^2}.$$

This representation has a much lower dimension than the normalized smoothed word vector, but contains exactly the same information. This representation of a word is an example of a **word embedding** — a representation that maps a word to a point in some high dimensional space, where embedded points have good properties. In this case, we seek an embedding that places words with similar meanings near one another.

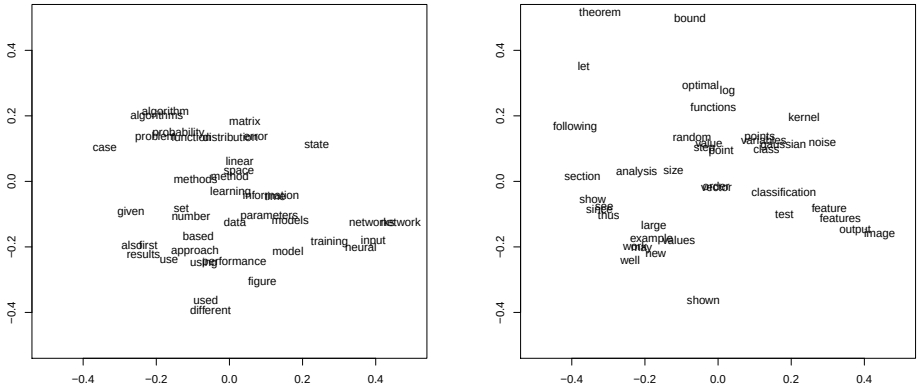


FIGURE 7.6: *On the* **left**, *the 40 most frequent words in the NIPS dataset, plotted using a multidimensional scaling of the document frequencies, smoothed using latent semantic analysis. On the* **right**, *the next 40 most frequent words, plotted in the same way. I used 1000 dimensions for the smoothing. Words that have a similar pattern of incidence in documents appear near one another.*

7.3.5 Mapping NIPS Words

LSA does not give a particularly strong word embedding, as this example will show. I used the dataset of section 7.3.3, and computed a representation on 1000 dimensions. Figure 7.5 shows a multidimensional scaling (using the method of section 7.2.3) onto two dimensions, where distances between points are given by distances between the normalized vectors of section 7.3.4. I have shown only the top 80 words, so that the figures are not too cluttered to read.

Some results are natural. For example, "used" and "using" lie close to one another, as do "algorithm" and "algorithms"; "network" and "networks"; and "features" and "feature". This suggests the data wasn't stemmed or even pre-processed to remove plurals. Most of the pairs that seem to make sense (and aren't explained as plurals or inflections) seem to have more to do with phrases than with meaning. For example, "work" and "well" are close ("work well"); "problem" and "case" ("problem case"); "probability" and "distribution" ("probability distribution"). Some pairs are close because the words likely appear near one another in common phrases. So "classification" and "feature" suggest "feature based classification" or "classification by feature".

This tendency can be seen in the k-nearest neighbors of embedded words, too. Table 7.1 shows the six nearest neighbors for the 20 most frequent words. But there is evidence that the embedding is catching some kind of semantics, too. Notice that "network", "neural", "units" and "weights" are close, as they should be. Similarly, "distribution", "distributions" and "probability" are close, and so are "algorithm" and "problem".

Embedding words in a way that captures semantics is a hard problem. Good

model	models	also	using	used	figure	parameters
learning	also	used	using	machine	results	use
data	using	also	used	use	results	set
algorithm	algorithms	problem	also	set	following	number
set	also	given	using	results	used	use
function	functions	also	given	using	defined	paper
using	used	use	also	results	given	first
time	also	first	given	used	university	figure
figure	shown	shows	used	using	also	different
number	also	results	set	given	using	used
problem	problems	following	paper	also	set	algorithm
models	model	using	also	used	parameters	use
used	using	use	also	results	first	university
training	used	set	using	test	use	results
given	also	using	set	results	university	first
also	results	using	use	used	well	first
results	also	using	used	paper	use	show
distribution	distributions	given	probability	also	university	using
network	networks	neural	input	output	units	weights
based	using	also	use	used	results	given

TABLE 7.1: *The leftmost column gives the top 20 words, by frequency of use in the NIPS dataset. Each row shows the seven closest words to each query word using the cosine distance applied to document counts of the word smoothed using latent semantic analysis. I used 1000 dimensions. Words that have similar patterns of use across documents do have important similarities, but these are not restricted to similarities of meaning. For example, "algorithm" is very similar to "algorithms", and also to "following" (likely because the phrase "following algorithm" is quite common) and to "problem" (likely because it's natural to have an algorithm to solve aproblem).*

recent algorithms use finer measures of word similarity than the pattern of documents a word appears in. Strong recent methods, like Word2Vec or Glove, pay most attention to the words that appear near the word of interest, and construct embeddings that try to explain such similarity statistics. These methods tend to be trained on very large datasets, too; much larger than this one.

7.3.6 TF-IDF

The raw count of the number of times a word appears in a document may not be the best value to use in a term-document matrix. If a word is very common in all documents, then the fact that it appears often in a given document isn't that informative about what the document means. If a word appears only in a few documents, but is quite common in those documents, the number of times the word appears may understate how important it is. For example, in a set of documents about small pets, a word like "cat" is likely to appear often in each document; a word like "tularemia" is unlikely to appear often in many documents, but will tend to be repeated in a document if it appears. You can then argue that observing "cat" five times is a lot less informative about the document than observing "tularemia" five times is. Much time and trouble has been spent on making this very appealing

argument more rigorous, without significant benefits that I'm aware of.

All this suggests that you might use a modified word score. The standard is known as **TF-IDF** (or, very occasionally, **term frequency-inverse document frequency**). Write c_{ij} for the number of times the i'th word appears in the j'th document, N for the number of documents, and N_i for the number of documents that contain at least one instance of the i'th word. Then one TF-IDF score is

$$c_{ij} \log \frac{N}{N_i}$$

(where we exclude cases where $N_i = 0$ because the term then doesn't appear in any document). Notice that a term appears in most documents, the score is about the same as the count; but if the term appears in few documents, the score is rather larger than the count. Using this score, rather than a count, tends to produce improved behavior from systems that use term-document matrices. There are a variety of ingenious variants of this score – the wikipedia page lists many – each of which tends to produce changes in systems (typically, some things get better and some get worse). Don't forget the logarithm, which got dropped from the acronym for no reason I know.

7.4 YOU SHOULD

7.4.1 remember these definitions:

7.4.2 remember these terms:

7.4.3 remember these facts:

7.4.4 remember these procedures:

7.4.5 be able to:

- Use a singular value decomposition to obtain principal components.

PROBLEMS

7.1. You have a dataset of N vectors $\mathbf{x}_i$ in d-dimensions, stacked into a matrix $\mathcal{X}$. This dataset *does not* have zero mean. The data $\mathbf{x}_i$ is noisy. We use a simple noise model. Write $\tilde{\mathbf{x}}_i$ for the true underlying value of the data item, and ξ_i for the value of a normal random variable with zero mean and covariance $\sigma^2\mathcal{I}$. Then we use the model

$$\mathbf{x}_i = \tilde{\mathbf{x}}_i + \xi_i.$$

In matrices, we write

$$\mathcal{X} = \tilde{\mathcal{X}} + \Xi.$$

We will assume that $\mathsf{mean}\left(\{\xi\}\right) = \mathbf{0}$ and $\mathsf{Covmat}\left(\{\xi\}\right) = \sigma^2\mathcal{I}$.

(a) Show that our assumptions mean that the row rank of Ξ is d. Do this by contradiction: show if the row rank of Ξ is $r < d$, there is some rotation $\mathcal{U}$ so that each $\mathcal{U}\xi$ has zeros in the last $d - r$ components; now think about the covariance matrix of $\mathcal{U}\xi$.

(b) Assume that the row rank of $\tilde{\mathcal{X}}$ is $s << d$. Show that the row rank of $\mathcal{X}$ is d. Do this by noticing that the noise is independent of the dataset. This means that $\mathsf{mean}\left(\left\{\mathbf{x}\xi^T\right\}\right) = \mathsf{mean}\left(\{\mathbf{x}\}\right)\mathsf{mean}\left(\left\{\xi^T\right\}\right) = 0$. Now show that

$$\mathsf{Covmat}\left(\{\mathbf{x}\}\right) = \mathsf{Covmat}\left(\{\tilde{x}\}\right) + \sigma^2\mathcal{I}.$$

Now use the results of the previous exercise

(c) We now have a geometric model for both $\tilde{\mathcal{X}}$ and $\mathcal{X}$. The points in $\tilde{\mathcal{X}}$ lie on some hyperplane that passes through the origin in d-dimensional space. This hyperplane has dimension s.

(d) The points in $\mathcal{X}$ lie on a "thickened" version of this hyperplane which has dimension d because the matrix has rank d. Show that the variance of the data in any direction normal to the original hyperplane is σ^2.

(e) Use the previous subexercises to argue that a rank s approximation of $\mathcal{X}$ lies closer to $\tilde{\mathcal{X}}$ than $\mathcal{X}$ does. Use the Frobenius norm.

7.2. Write $\mathcal{D}^{(2)}$ for the matrix whose i,j'th component is

$$D^{(2)}_{ij} = \left(\mathbf{x}_i - \mathbf{x}_j\right)^T\left(\mathbf{x}_i - \mathbf{x}_j\right) = \mathbf{x}_i \cdot \mathbf{x}_i - 2\mathbf{x}_i \cdot \mathbf{x}_j + \mathbf{x}_j \cdot \mathbf{x}_j$$

where $\mathsf{mean}\left(\{x\}\right) = \mathbf{0}$. Now write

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

Show that

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

7.3. You have a dataset of N vectors $\mathbf{x}_i$ in d-dimensions, stacked into a matrix $\mathcal{X}$, and wish to build an s dimensional dataset $\mathcal{Y}_s$ so that $\mathcal{Y}_s{\mathcal{Y}_f}^T$ minimizes $\|\mathcal{Y}_s{\mathcal{Y}_f}^T - \mathcal{X}\mathcal{X}^T\|_F$. Form an SVD, to get

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

and write

$$\mathcal{Y} = \mathcal{U}_s\Sigma_s$$

(the subscript-s notation is in the chapter).

(a) Show that

$$\|\mathcal{Y}_s{\mathcal{Y}_f}^T - \mathcal{X}\mathcal{X}^T\|_F = \|\Sigma_s^2 - \Sigma^2\|_F.$$

Explain why this means that $\mathcal{Y}_s$ is a solution.

(b) For any $s \times s$ orthonormal matrix $\mathcal{R}$, show that $\mathcal{Y}_R = \mathcal{U}_s \Sigma_s \mathcal{R}$ is also a solution. Interpret this geometrically.

PROGRAMMING EXERCISES

7.4. 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 investigate simple document clustering with this dataset.

(a) Reproduce figure 7.5 using approximations with rank 100, 500, and 2000. Which is best, and why?

(b) Now use a TF-IDF weight to reproduce figure 7.5 using approximations with rank 100, 500, and 2000. Which is best, and why?

7.5. 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 investigate simple distributional semantics using this dataset.

(a) Reproduce figure 7.6 using approximations with rank 100, 500, and 2000. Which is best, and why?

(b) Now use a TF-IDF weight to reproduce figure 7.6 using approximations with rank 100, 500, and 2000. Which is best, and why?

7.6. 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?

7.7. 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) 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. This should tell you something about the similarity of the classes. For example, imagine images in class A consist of dark circles that are centered in a light image, but where different images have circles of different sizes; images of class B are dark on the left, light on the right, but different images change from dark to light at different vertical lines. Then the mean of class A should look like a

fuzzy centered blob, and its principal components make the blob bigger or smaller. The principal components of class B will move the dark patch left or right. Encoding an image of class A with the principal components of class B should work very badly. But if class C consists of dark circles that move left or right from image to image, encoding an image of class C using A's principal components might work tolerably. Now define the similarity between classes to be $(1/2)(E(A \rightarrow B) + E(B \rightarrow A))$. 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 8

Canonical Correlation Analysis

In many applications, one wants to associate one kind of data with another. For example, every data item could be a video sequence together with its sound track. You might want to use this data to learn to associate sounds with video, so you can predict a sound for a new, silent, video. You might want to use this data 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, every data item could be a captioned image. You might want to 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 kind of data?

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 of $\{\mathbf{p}\}$. 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 $\mathbf{x}_i$. We assume that $\{\mathbf{x}\}$ and $\{\mathbf{y}\}$ have zero mean, because it will simplify the equations and is easy to achieve. There is a standard procedure for dealing with data like this. This is quite good at, say, predicting words to attach to pictures. However, it can result in a misleading analysis, and I show how to check for this.

8.1 CANONICAL CORRELATION ANALYSIS

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 get to 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

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

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 $(1/\lambda_1)\Sigma_{xx}^{-1}\Sigma_{xy}\mathbf{v} = \mathbf{u}$ 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

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

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. The values of $\text{corr}\left(\left\{\mathbf{u}^T\mathbf{x}, \mathbf{v}^T\mathbf{y}\right\}\right)$ for the different directions are often called **canonical correlations**. The projections are sometimes known as **canonical variables**.

Worked example 8.1 *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

Procedure: 8.1 *Canonical Correlation Analysis*

Given 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. Write Σ for the covariance matrix of $\{\mathbf{p}\}$. We have

$$\Sigma = \begin{bmatrix} \Sigma_{xx} & \Sigma_{xy} \\ \Sigma_{yx} & \Sigma_{yy} \end{bmatrix}.$$

Write $\mathbf{u}_j$ for the eigenvectors of

$$\Sigma_{xx}^{-1}\Sigma_{xy}\Sigma_{yy}^{-1}\Sigma_{xy}^{T}$$

sorted in descending order of eigenvalue. Write $\mathbf{v}_j$ for the eigenvectors of

$$\Sigma_{yy}^{-1}\Sigma_{xy}^{T}\Sigma_{xx}^{-1}\Sigma_{xy}$$

sorted in descending order of eigenvalue. Then $\mathbf{u}_1^T\mathbf{x}_i$ is most strongly correlated with $\mathbf{v}_1\mathbf{y}_i$; $\mathbf{u}_2^T\mathbf{x}_i$ is second most strongly correlated with $\mathbf{v}_2\mathbf{y}_i$; and so on, up to $j = \min(d_x, d_y)$.

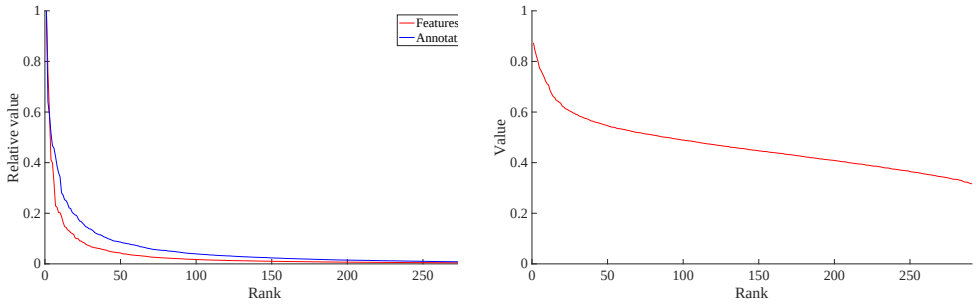


FIGURE 8.1: *On the* **left**, *the 291 largest eigenvalues of the covariance for features and for word vectors, normalized by the largest eigenvalue in each case, plotted against rank. Notice in each case relatively few eigenvalues capture most of the variance. The word vectors are 291 dimensional, so this figure shows all the variances for the word vectors, but there are a total of 3000 eigenvalues for the features. On the* **right**, *the canonical correlations for this data set. Notice that there are some rather large correlations, but quite quickly the values are small.*

8.2 EXAMPLE: CCA OF WORDS AND PICTURES

CCA is commonly used to find good matching spaces. Here is an example. Assume we have a set of captioned images. It is natural to want to build two systems: given an image, caption it; and given a caption, produce a good image. There is a very wide range of methods that have been deployed to attack this problem. Perhaps the simplest – which is suprisingly effective – is to use a form of nearest neighbors in a cleverly chosen space. We have N images described by feature vectors $\mathbf{x}_i$, corresponding to N captions described by word vectors $\mathbf{y}_i$. The i'th image corresponds to the i'th caption. The image features have been constructed using specialized methods (there are some constructions in chapter 57, but coming up with the best construction is still a topic of active research, and way outside the scope of this book). The word vectors are like those of section 7.3.

We would like to map the word vectors and the image features into a new space. We will assume that the features have extracted all the useful properties of the images and captions, and so a linear map of each is sufficient. If an image and a caption correspond, we would like their feature vectors to map to points that are nearby. If a caption describes an image very poorly, we would like its feature vector to map far away from where the image's feature vector maps.

Assume we have this new space. Then we could come up with a caption for a new image by mapping the image into the space, and picking the nearest point that represents a caption. We could come up with an image for a new caption by mapping the caption into the space, then picking the nearest point that represents an image. This strategy (with some tuning, improvements, and so on) remains extremely hard to beat.

For this example, I will use a dataset called the IAPR TC-12 benchmark, which is published by ImageCLEF. A description of the dataset can be found at https://www.imageclef.org/photodata. There are 20,000 images, each of which has

FIGURE 8.2: *Four images with true (in red) and predicted (green) label words. Words are predicted using a CCA of image features and word vectors, as described in the text. Images are from a test set, not used in constructing the CCA. The yellow box gives the cosine distance between the predicted and true word vectors, smoothed by projection to a 150 dimensional space as in section 57. For these images, the cosine distances are reasonably close to one, and the predictions are quite good.*

FIGURE 8.3: *Four images with true (in red) and predicted (green) label words. Words are predicted using a CCA of image features and word vectors, as described in the text. Images are from a test set, not used in constructing the CCA. The yellow box gives the cosine distance between the predicted and true word vectors, smoothed by projection to a 150 dimensional space as in section 57. For these images, the cosine distances are rather far from one, and the predictions are not as good as those in figure 8.2.*

FIGURE 8.4: *Four images with true (in red) and predicted (green) label words. Words are predicted using a CCA of image features and word vectors, as described in the text. Images are from a test set, not used in constructing the CCA. The yellow box gives the cosine distance between the predicted and true word vectors, smoothed by projection to a 150 dimensional space as in section 57. For these images, the cosine distances are rather close to zero, and the predictions are bad.*

a text annotation. The annotations use a vocabulary of 291 words, and the word vectors are binary (ie word is there or not). I used image features published by Mathieu Guillaumin, at https://lear.inrialpes.fr/people/guillaumin/data.php. These features are not the current state-of-the-art for this problem, but they're easily available and effective. There are many different features available at this location, but for these figures, I used the DenseSiftV3H1 feature set. I matched test images to training captions using the 150 canonical variables with largest canonical correlations.

The first thing you should notice (Figure 8.1) is that both image features and text behave as you should expect. There are a small number of large eigenvalues in the covariance matrix, and a large number of small eigenvalues. Because the scaling of the features is meaningless, I have plotted the eigenvalues as fractions of the largest value. Notice also that the first few canonical correlations are large (Figure 8.1).

There are two ways to evaluate a system like this. The first is qualitative, and if you're careless or optimistic, it looks rather good. Figure 8.2 shows a set of images with true word labels and labels predicted using the nearest neighbors procedure. Predicted labels are in green, and true labels are in red. Mostly, these labellings should look quite good to you. Some words are missing from the predictions, true, but most predictions are about right.

A quantitative evaluation reflects the "about right" sense. For each image, I formed the cosine distance between the predicted word vector and the true word vector, smoothed by projection to a 150 dimensional space. This is the number in

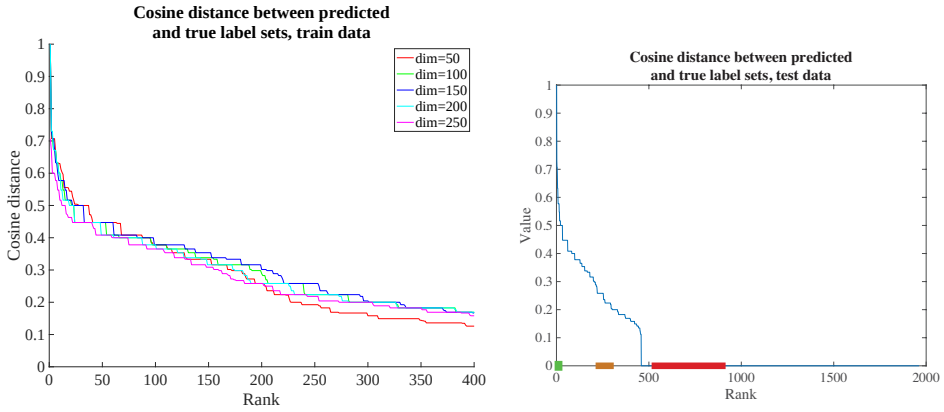


FIGURE 8.5: **Left:** *all values of cosine distance between predicted and true word labels, sorted from best to worst, for the CCA method of the text, for different numbers of canonical variables, for the training data. The distances are fairly good, and 150 seems like a reasonable choice of dimension. On the* **right**, *cosine distances between predicted and true for* test *data; this looks much worse. I have marked the regions where the "good", "medium" and "bad" figures come from. Note that most values are bad – predicting words from images is hard. Accurate predictions require considerably more sophisticated feature constructions than we have used here.*

the yellow box. These numbers are fairly close to one for Figure 8.2, which is a good sign. But Figures 8.3 and 8.4 suggest real problems. There are clearly images for which the predictions are poor. In fact, predictions are poor for most images, as Figure 8.5 shows. This figure gives the cosine distance between predicted and true labels (again, smoothed by projection to a 150 dimensional space), sorted from best to worst, for all test images. Most produce really bad label vectors with very low cosine distance.

Improving this is a matter of image features. The features I have used here are outdated. I used them because it was easy to get many different sets of features for the same set of images (yielding some rather interesting exercises). Modern feature constructions allow improved labelling of images, but modern systems still tend to use CCA, although often in more complex forms than we can deal with here.

8.3 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 is a property of surfaces. A dark surface has low albedo (it reflects relatively little of the light that falls on it) and a light surface has high albedo (it reflects most of the light that falls on it). 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 this), but its albedo doesn't change at all. To change an object's albedo, you need (say) a marker or paint. All this suggests that a CCA of albedo against shading will suggest there is no correlation.

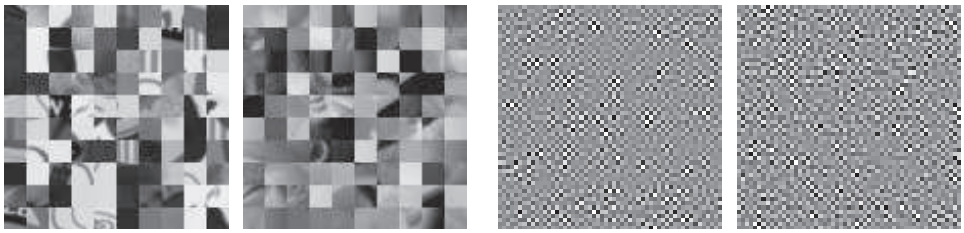
Because this is a classical problem, there are datasets one can download. There is a very good dataset giving the albedo and shading for images, collected by Roger Grosse, Micah K. Johnson, Edward H. Adelson, and William T. Freeman at http://www.cs.toronto.edu/~rgrosse/intrinsic/. These images show individual objects on black backgrounds, and there are masks identifying object pixels. For each image in the dataset, there is an albedo map (basically, an image of the albedos) and a shading map. These maps are constructed by clever photographic techniques. 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. The albedo tiles I chose for a particular image were in the same locations in that image as the shading tiles — each pair of tiles represents a pair of albedo-shading in some image patch. 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.

If this doesn't strike you as ridiculous, then you should check you understand the definitions of albedo and shading. How could albedo and shading be correlated? Do people put dark objects in light places, and light objects in dark places? The correct answer is that they are not correlated, but that this analysis has missed one important, nasty point. The objective function we are maximizing is a ratio

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

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 8.6, 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.



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

FIGURE 8.6: *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.*

8.3.1 Are Correlations Significant?

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 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 8.7 shows what happens for the mouse canonical correlation of example 8.1. 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'

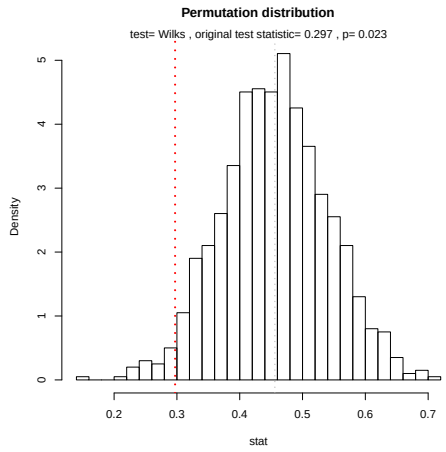


FIGURE 8.7: *A histogram of values of Wilks' lambda obtained from permuted versions of the mouse dataset of example 8.1. The value obtained for the original dataset is shown by the vertical line. Notice that most values are larger (about* 97% *of values), meaning that we would see the canonical correlation values we see only about once in 30 experiments if they were purely a chance effect. There is very likely a real effect here.*

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. As figure 8.8 shows, there is good evidence that the correlations for the words and pictures data of section 8.2 are not accidental, either.

But the albedo-shading correlations really are accidental. Figure 8.9 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 Wilks' lambda — the correlations are entirely an accident, and are of no statistical significance.

Remember this: *A canonical correlation analysis can mislead you. The problem is the division in the objective function. If you're working with data where many principal components have very small variances, you can get large correlations as a result. You should always check whether the CCA is actually telling you something useful. A natural check is the Wilks' lambda procedure.*

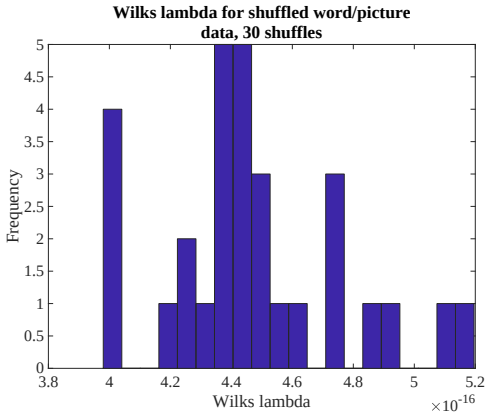


FIGURE 8.8: *A histogram of values of Wilks' lambda obtained from permuted versions of the word and picture dataset of section 8.2. I computed the value for the first 150 canonical variates, and used 30 shuffles (which takes quite a long time). The value for the true dataset is* $9.92e-25$*, which suggests very strongly that the correlations are not accidental.*

8.4 YOU SHOULD

8.4.1 remember these definitions:

8.4.2 remember these terms:

8.4.3 remember these facts:

8.4.4 remember these procedures:

8.4.5 be able to:

- Use a canonical correlation analysis to investigate correlations between two types of data.
- Use Wilks' lambda to determine whether correlations are the result of real effects.

PROGRAMMING EXERCISES

8.1. We investigate CCA to predict words from pictures using Mathieu Guillaumin's

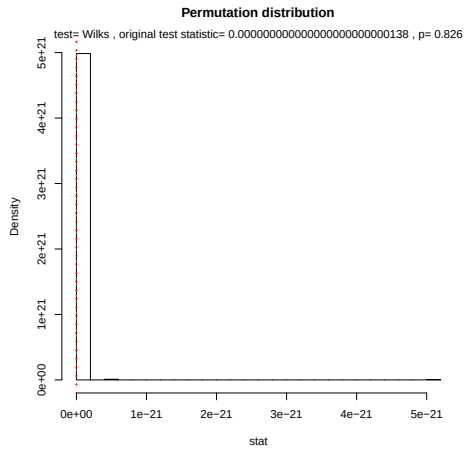


FIGURE 8.9: *A histogram of values of Wilks' lambda obtained from permuted versions of the 400 tile albedo shading dataset discussed in the text. The value obtained for the original dataset is shown by the vertical line, and is really tiny (rather less than 1e-21). But rather more than four-fifths (*82.6%*) 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.*

published features, available at https://lear.inrialpes.fr/people/guillaumin/data.php.

(a) Reproduce Figure 8.1 and Figure 8.5 of section 8.2, using the same features and the same number of canonical variables.

(b) One reasonable summary of performance is the mean of the cosine distance between true and predicted label vectors over all test images. This number will vary depending on how many of the canonical variables you use to match. Plot this number over the range $[1 \dots 291]$, using at least 10 points.

(c) Based on the results of the previous subexercise, choose a good number of canonical variables. For the 30 most common words in the vocabulary, compute the total error rate, the false positive rate, and the false negative rate for predictions over the whole test set. Does this suggest any way to improve the method?

8.2. We investigate image features using Mathieu Guillaumin's published features, available at https://lear.inrialpes.fr/people/guillaumin/data.php.

(a) Compute a CCA of the GIST features against the DenseSiftV3H1 features, and plot the sorted canonical correlations. You should get a figure like figure **??**. Does this suggest that different feature sets encode different aspects of the image?

(b) If you concatenate GIST features with DenseSiftV3H1 features, do you get improved word predictions?

8.3. Here is a much more elaborate exercise investigating CCA to predict words from pictures using Mathieu Guillaumin's published features, available at https://

lear.inrialpes.fr/people/guillaumin/data.php.

(a) Reproduce Figure 8.5 of section 8.2, for *each* of the available image feature sets. Is any particular feature set better overall?

(b) Now take the top 50 canonical variables of each feature set for the images, and concatenate them. This should yield 750 variables that you can use as image features. Reproduce Figure 8.5 for this set of features. Was there an improvement in performance?

(c) Finally, if you can get your hands on some hefty linear algebra software, concatenate all the image feature sets. Reproduce Figure 8.5 for this set of features. Was there an improvement in performance?

PART THREE

CLUSTERING

CHAPTER 9

Clustering

One very good, very simple, model for 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. The blobs are usually called **clusters**, and the 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. Typically, one evaluates clustering by seeing how well it supports an application.

There are many applications of clustering. You can summarize a dataset by clustering it, then reporting a summary of each cluster. Summaries might be either a typical element of each cluster or (say) the mean of each cluster. Clusters can help expose structure in a dataset that is otherwise quite difficult to see. For example, in section 57, I show ways of visualizing the difference between sets of grocery stores by clustering customer records. It turns out that different sets of stores get different types of customer, but you can't see that by looking directly at the customer records. Instead, you can assign customers to types by clustering the records, then look at what types of customer go to what set of store. This observation yields a quite general procedure for building features for complex signals (images, sound, accelerometer data). The method can take signals of varying size and produce a fixed size feature vector, which is then used for classification.

9.1 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 9.1). 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 9.2). The difficulty here is we need to know some criterion for splitting clusters.

Procedure: 9.1 *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: 9.2 *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 9.1).

Another important thing to notice about clustering from the example of figure 9.1 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.

9.1.1 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

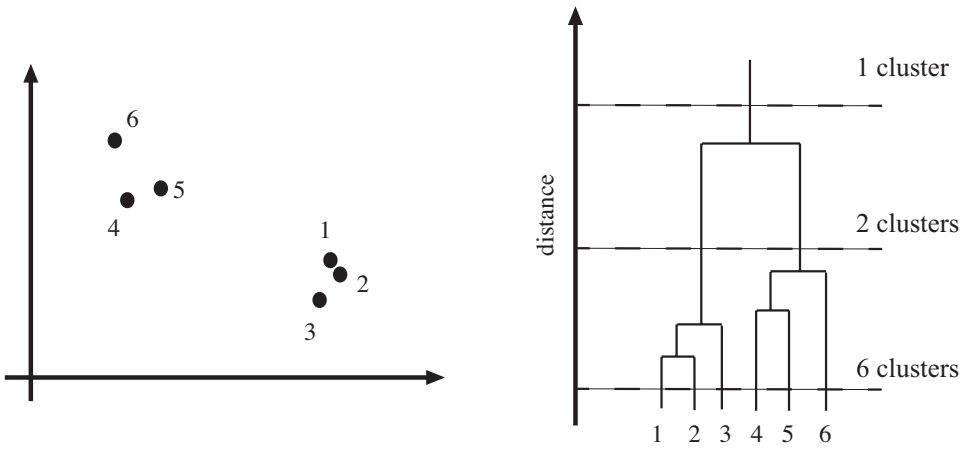


FIGURE 9.1: **Left**, *a data set;* **right**, *a dendrogram obtained by agglomerative clustering using single-link clustering. If one selects a particular value of distance, then a horizontal line at that distance splits the dendrogram into clusters. This representation makes it possible to guess how many clusters there are and to get some insight into how good the clusters are.*

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**; I described how to do this in the exercises (page 77).

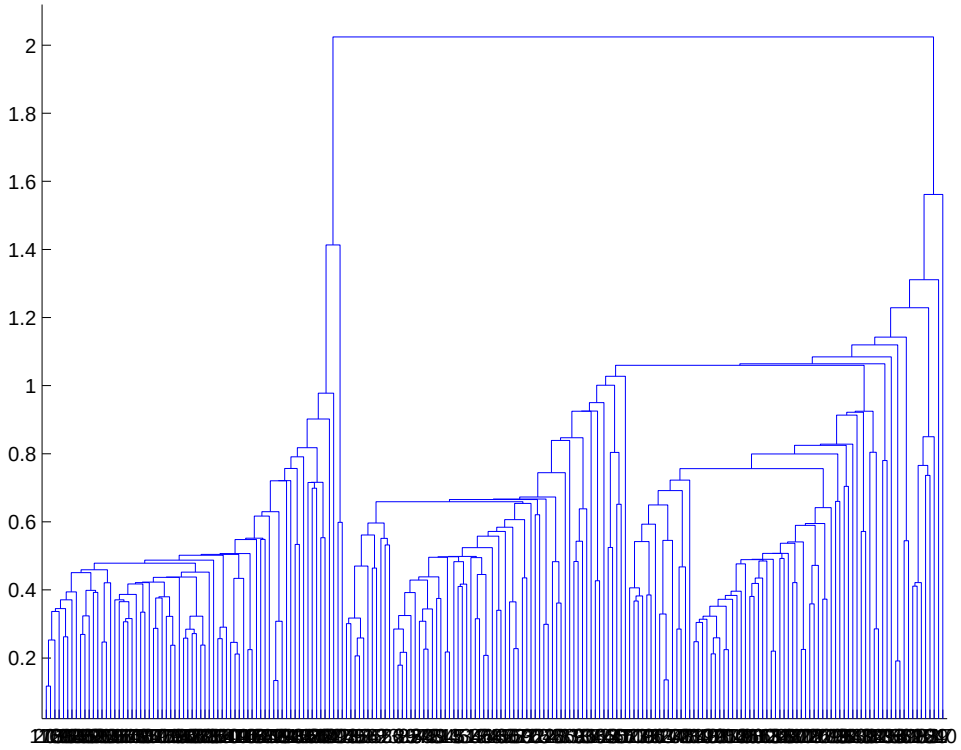


FIGURE 9.2: *A dendrogram obtained from the seed dataset, using single link clustering. Recall that the data points are on the horizontal axis, and that the vertical axis is distance; there is a horizontal line linking two clusters that get merged, established at the height at which they're merged. I 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".*

Worked example 9.1 *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 9.2). I deliberately forced Matlab to plot the whole dendrogram, which accounts for the crowded look of the figure. As you can see from the dendrogram and from Figure 9.3, this data clusters rather well.

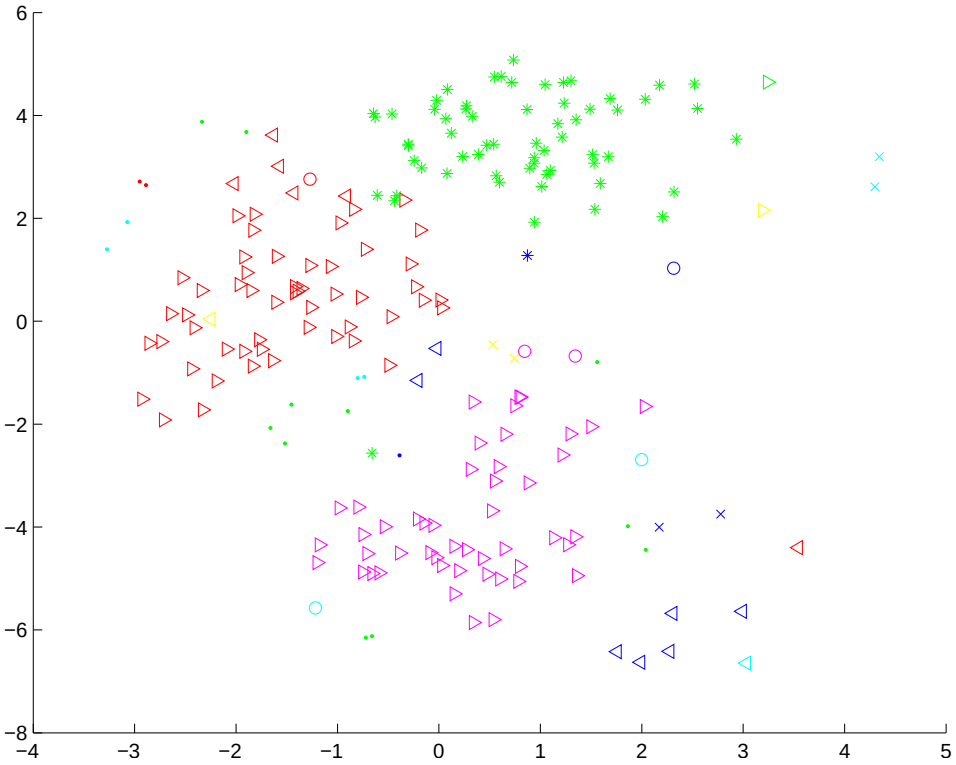


FIGURE 9.3: *A clustering of the seed dataset, using agglomerative clustering, single link distance, and requiring a maximum of 30 clusters. I have plotted each cluster with a distinct marker (though some markers differ only by color). Notice that there are a set of fairly natural isolated clusters. 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).*

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.*

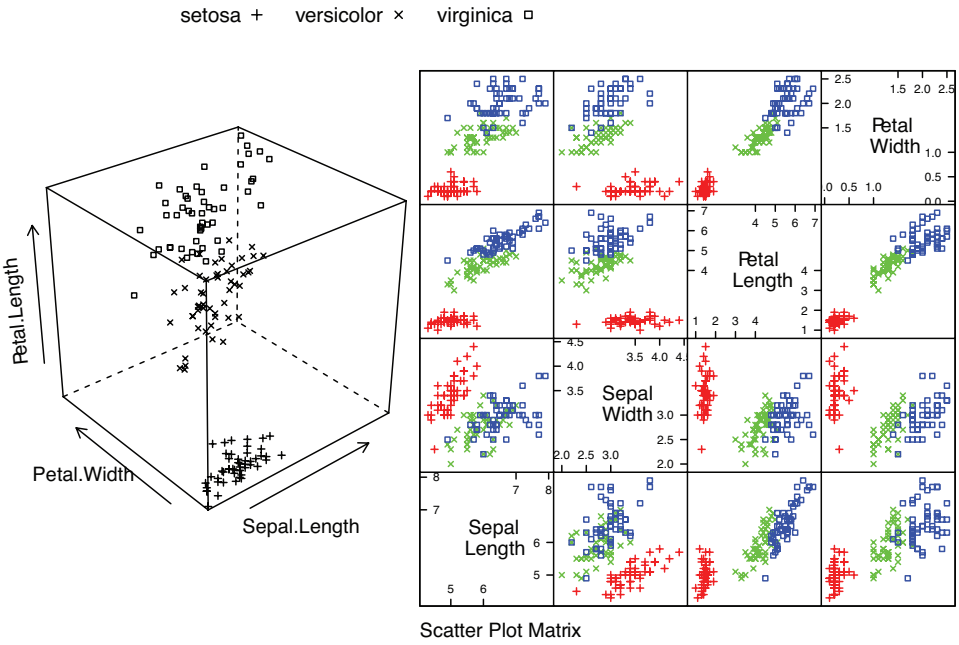


FIGURE 9.4: **Left:** *a 3D scatterplot for the famous Iris data, collected by Edgar Anderson in 1936, and made popular amongst statisticians by Ronald Fisher in that year. I have chosen three variables from the four, and have plotted each species with a different marker. You can see from the plot that 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.*

9.2 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 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 formalize this fairly easily by writing an expression for the squared distance between data points and their cluster centers. Assume that we know how many clusters there are in the data, and write k for this number. There are N data items. 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 write $\delta_{i,j}$ for a discrete variable that records which cluster a data item belongs to, so

$$\delta_{i,j} = \begin{cases} 1 & \text{if } \mathbf{x}_i \text{ belongs to cluster } j \\ 0 & \text{otherwise} \end{cases}$$

We require that every data item belongs to exactly one cluster, so that $\sum_j \delta_{i,j} = 1$. We require that every cluster contain at least one point, because we assumed we knew how many clusters there were, so we must have that $\sum_i \delta_{i,j} > 0$ for every j. We can now write the sum of squared distances from data points to cluster centers as

$$\Phi(\delta, \mathbf{c}) = \sum_{i,j} \delta_{i,j} \left[(\mathbf{x}_i - \mathbf{c}_j)^T (\mathbf{x}_i - \mathbf{c}_j) \right].$$

Notice how the $\delta_{i,j}$ are acting as "switches". For the i'th data point, there is only one non-zero $\delta_{i,j}$ which selects the distance from that data point to the appropriate cluster center. It is natural to want to cluster the data by choosing the δ and $\mathbf{c}$ that minimizes $\Phi(\delta, \mathbf{c})$. This would yield the set of k clusters and their cluster centers such that the sum of distances from points to their cluster centers is minimized.

There is no known algorithm that can minimize Φ exactly in reasonable time. The $\delta_{i,j}$ are the problem: it turns out to be hard to choose the best allocation of points to clusters. The algorithm we guessed above is a remarkably effective approximate solution. Notice that if we know the $\mathbf{c}$'s, getting the δ's is easy – for the i'th data point, set the $\delta_{i,j}$ corresponding to the closest $\mathbf{c}_j$ to one and the others to zer. Similarly, if the $\delta_{i,j}$ are known, it is easy to compute the best center for each cluster – just average the points in the cluster. So we iterate:

- Assume the cluster centers are known and allocate each point to the closest cluster center.
- Replace each center with the mean of the points allocated to that cluster.

We 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 9.3).

Procedure: 9.3 *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.

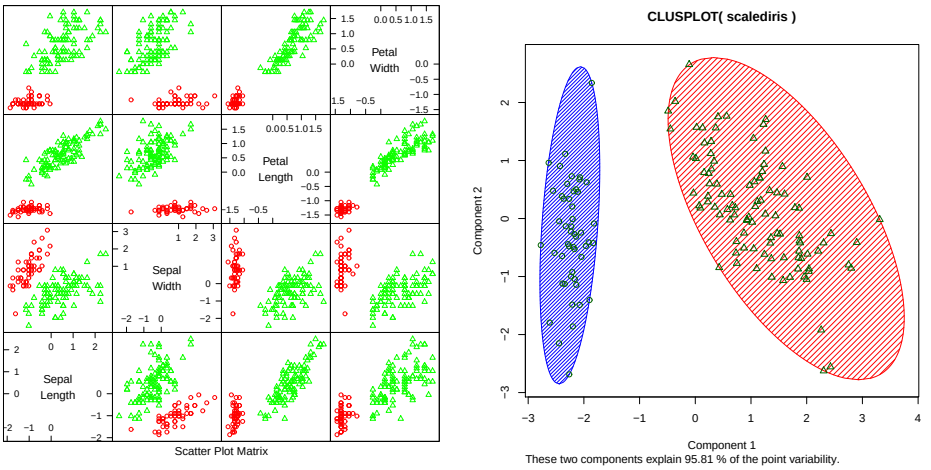


FIGURE 9.5: *On the* **left**, *a panel plot of the iris data clustered using k-means with* $k = 2$. *By comparison with figure 9.8, notice how the* versicolor *and* verginica *clusters appear to have been merged. On the* **right**, *this data set projected onto the first two principal components, with one blob drawn over each cluster.*

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 5.4.2 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 5.1. I reproduce figure 5.3 from that section as figure 9.8, for comparison. Figures 9.5, 9.6 and 9.7 show different k-means clusterings of that data.

One natural strategy for initializing k-means is to choose k data items at random, then use each as an initial cluster center. This approach is widely used, but has some difficulties. The quality of the clustering can depend quite a lot on initialization, and an unlucky choice of initial points might result in a poor clustering. One (again quite widely adopted) strategy for managing this is to initialize several times, and choose the clustering that performs best in your application. Another strategy, which has quite good theoretical properties and a good reputation, is known as **k-means++**. You choose a point $\mathbf{x}$ uniformly and at random from the dataset to be the first cluster center. Then you compute the squared distance between that point and each other point; write $d_i^2(\mathbf{x})$ for the distance from the i'th point to the first center. You now choose the other $k-1$ cluster centers as IID draws from the probability distribution

$$\frac{d_i^2(\mathbf{x})}{\sum_u d_u^2(\mathbf{x})}.$$

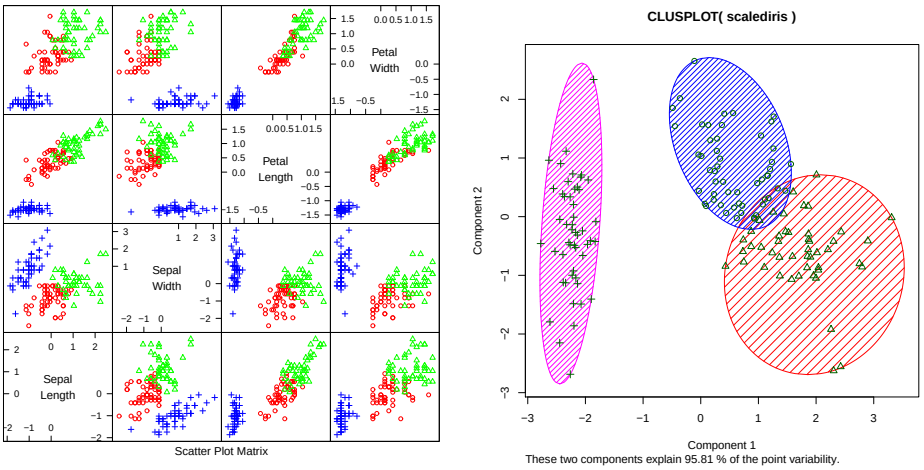


FIGURE 9.6: *On the* **left**, *a panel plot of the iris data clustered using k-means with* $k = 3$. *By comparison with figure 9.8, notice how the clusters appear to follow the species labels. On the* **right**, *this data set projected onto the first two principal components, with one blob drawn over each cluster.*

9.2.1 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 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 9.8 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.

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 9.3). There are usually natural ways to evaluate this application. For example, vector quantization

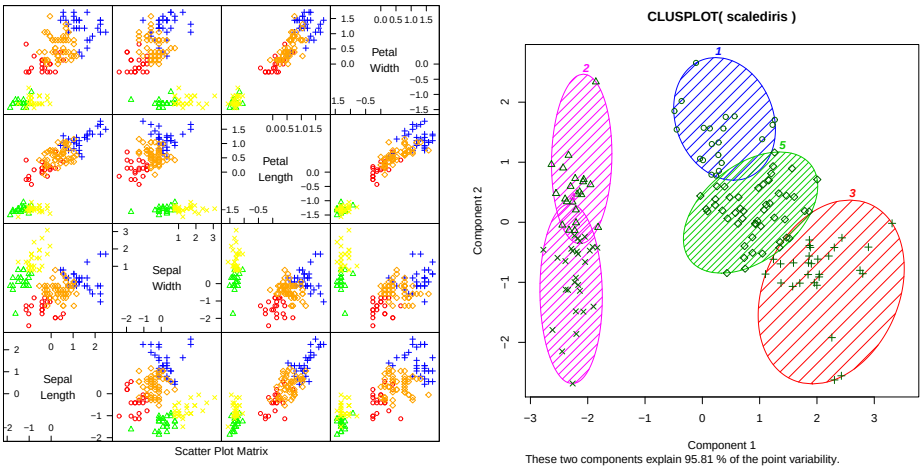


FIGURE 9.7: *On the* **left**, *a panel plot of the iris data clustered using k-means with* $k = 5$. *By comparison with figure 9.8, notice how* setosa *seems to have been broken in two groups, and* versicolor *and* verginica *into a total of three . On the* **right**, *this data set projected onto the first two principal components, with one blob drawn over each cluster.*

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.

9.2.2 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. These weights are different from the original $\delta_{i,j}$ because they are not forced to be either zero or one, however. Write $w_{i,j}$ for the weight connecting point i to cluster center j. Weights should be non-negative (i.e. $w_{i,j} \geq 0$), and each point should carry a total weight of 1 (i.e $\sum_j w_{i,j} = 1$), so that it if the i'th point contributes more to one cluster center, it is forced to contribute less to all others. You should see $w_{i,j}$ as a simplification of the $\delta_{i,j}$ in the original cost function. We can write a new cost function

$$\Phi(w, \mathbf{c}) = \sum_{i,j} w_{i,j} \left[(\mathbf{x}_i - \mathbf{c}_j)^T (\mathbf{x}_i - \mathbf{c}_j) \right],$$

which we would like to minimize by choice of w and $\mathbf{c}$. There isn't any improvement in the problem, because for any choice of $\mathbf{c}$, the best choice of w is to allocate each point to its closest cluster center. This is because we have not specified any relationship between w and $\mathbf{c}$.

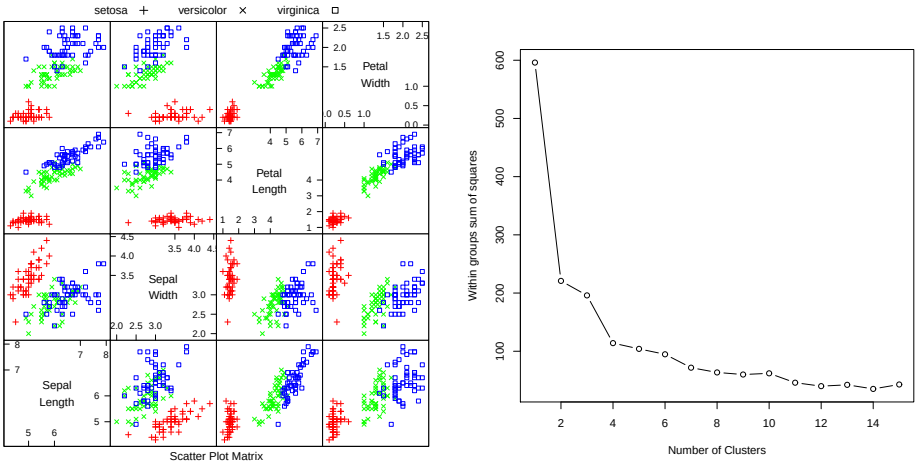


FIGURE 9.8: *On the* **left**, *the scatterplot matrix for the Iris data, for reference. On the* **right**, *a plot of the value of the cost function for each of several different values of k. Notice how there is a sharp drop in cost going from $k = 1$ to $k = 2$, and again at $k = 4$; after that, the cost falls off slowly. This suggests using $k = 2$, $k = 3$, or $k = 4$, depending on the precise application.*

But w and $\mathbf{c}$ should be coupled. 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 $\|\mathbf{x}_i - \mathbf{c}_j\|$, choose a scaling parameter $\sigma > 0$, and write

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

This $s_{i,j}$ is often called the **affinity** between the point i and the center j; it is large when they are close in σ units, and small when they are far apart. 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. The weight linking a point and a cluster center 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 the 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 9.4) for convenience.

Notice one other feature of this procedure. As long as you use sufficient precision for the arithmetic (which might be a problem), $w_{i,j}$ is *always* greater than zero. This means that no cluster is empty. In practice, if σ is small compared to the distances between points, you can end up with empty clusters. You can tell if this is happening by looking at $\sum_i w_{i,j}$; if this is very small or zero, you have a problem.

Procedure: 9.4 *K-Means with Soft Weights*

Choose k. Choose k data points $\mathbf{c}_j$ to act as initial cluster centers. Choose a scale, σ. 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

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

9.2.3 Efficient Clustering and Hierarchical K Means

One important difficulty occurs in applications. We might need to have an enormous dataset (millions of items is 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 helps rather a lot, but not enough if you expect the data will contain many clusters.

A more effective strategy is to build a hierarchy of k-means clusters. We 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

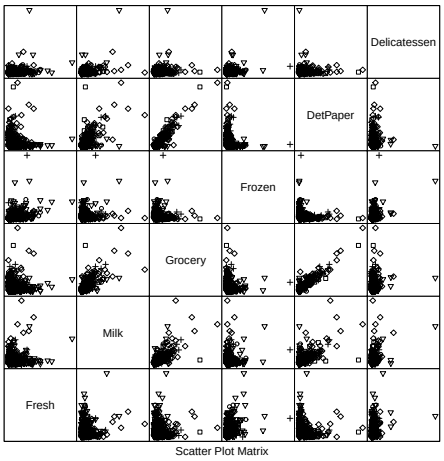


FIGURE 9.9: *A panel plot of the wholesale customer data of http://archive.ics.uci.edu/ml/datasets/Wholesale+customers, which records sums of money spent annually on different commodities by customers in Portugal. This data is recorded for six different blocks (two channels each within three regions). I have plotted each block with a different marker, but you can't really see much structure here, for reasons explained in the text.*

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.

9.2.4 K-Mediods

In some cases, we want to cluster objects that can't be averaged. 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 7.2.3, particularly Figure 7.1), then try and cluster using this data. As another example, you could collect data giving similarities between breakfast items as in Section 7.2.3, 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 the points in the cluster.

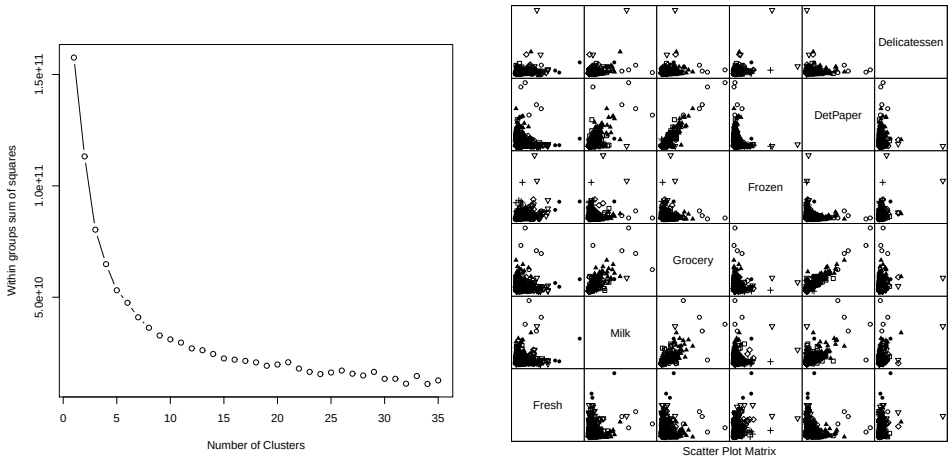


FIGURE 9.10: *On the* **left**, *the cost function (of section 9.2) for clusterings of the customer data with k-means for k running from 2 to 35. This suggests using a k somewhere in the range 10-30; I chose 10. On the* **right**, *I have clustered this data to 10 cluster centers with k-means. The clusters 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.*

9.2.5 Example: Groceries in Portugal

Clustering can be used to expose structure in datasets that isn't visible with simple tools. Here is an example. 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, corresponding to different types of shop) and by region (three regions). You can think of the data as being divided into six blocks (one for each pair of channel and region). There are 440 customer records, and there are many customers in each block. The data was provided by M. G. M. S. Cardoso.

Figure 9.9 shows a panel plot of the customer data; the data has been clustered, and I gave each of 10 clusters its own marker. You (or at least, I) can't see any evidence of the six blocks here. This is due to the form of the visualization, rather than a true property of the data. 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 blocks (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

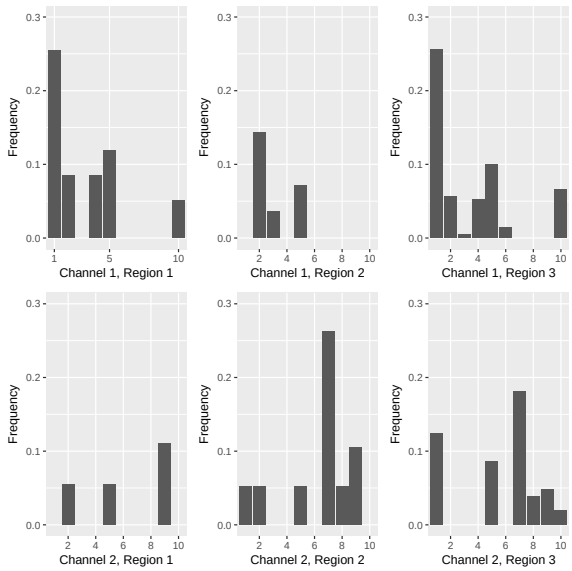


FIGURE 9.11: *The histogram of different types of customer, by block, for the customer data. Notice how the distinction between the blocks is now apparent — the blocks do appear to contain quite different distributions of customer type. It looks as though the channels (rows in this figure) are more different than the regions (columns in this figure).*

much structure at all, and is basically unhelpful.

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 delicatessen; 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 of the clustered data (Figure 9.9). The plot for this dataset is hard to read, because the 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 of the customers (Figure 9.10). 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 an important effect which isn't apparent in the panel 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 blocks 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 block 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 block 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 blocks.

An easy way to see this difference is to look at histograms of the types of customer *within each block*. Figure 9.11 shows this representation for the shopper dataset. The figure shows the histogram of customer types that appears in each block. The blocks do appear to contain quite different distributions of customer type, as you would expect. It looks as though the channels (rows in this figure) are more different than the regions (columns in this figure). Again, you might expect this: regions might contain slightly different customers (eg as a result of regional food preferences), but different channels are intended to cater to different customers.

9.2.6 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.

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.*

9.3 DESCRIBING REPETITION WITH VECTOR QUANTIZATION

The classifiers in Chapter 3 can be applied to simple images (the MNIST exercises at the end of the chapter, for example), but they will annoy you if you try to apply them as described to more complicated signals. All the methods described apply to feature vectors of fixed length. But typical of signals like speech, images, video, or accelerometer outputs is that different versions of the same thing have different lengths. For example, pictures appear at different resolutions, and it seems clumsy to insist that every image be 28×28 before it can be classified. As another example, some speakers are slow, and others are fast, but it's hard to see much future for a speech understanding system that insisted that everyone speak at the same speed so the classifier could operate. We need a construction that will take a signal and produce a useful feature vector of fixed length. This section shows one of the most useful such constructions (but be aware, this is an enormous topic).

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. However, it's fair to model this by saying 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. This suggests that the patches that are used to make an image reveal something about what is in the image. This observation works for speech, for video, and for accelerometer signals too.

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, 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 few) twisting movements is important, but knowing how the movements are linked together in time may not be. As a general rule, one can do quite a good job of classifying video just by knowing what patterns are there (i.e. without knowing where or when the patterns appear). Not all signals are like this. For example, in speech it really matters what sound follows what sound.

9.3.1 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, which can be reshaped into a vector. 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 usually 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, which can be reshaped into a vector. 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. We take a training set of signals, and cut each signal into pieces of fixed size and reshape them into d dimensional vectors. We then build a set of clusters out of these pieces. This set of clusters is often thought of as a dictionary, because we expect many or most cluster centers to look like pieces that occur often in the signals and so are repeated.

We can now now describe any new piece of signal 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** (often **VQ**).

This strategy applies to any kind of signal, and is surprisingly robust to details. We could use d dimensional vectors for a sound file; $\sqrt{d} \times \sqrt{d}$ dimensional patches for an image; or $3 \times (d/3)$ dimensional subarrays for an accelerometer signal. In each case, it is easy to compute the distance between two pieces using sum of squared differences. It seems not to matter much if the signals are cut into overlapping or

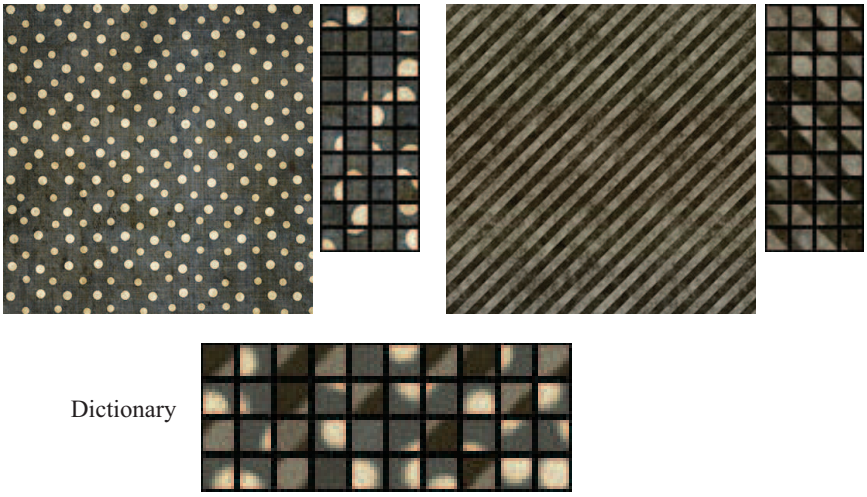


FIGURE 9.12: **Top:** *two images with rather exaggerated repetion, published on flickr.com with a creative commons license by* `webtreats`. *Next to these images, I have placed zoomed sampled 10x10 patches from those images; although the spots (resp. stripes) aren't necessarily centered in the patches, 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 in the dictionary and spot images by spot entries.*

non-overlapping pieces when forming the dictionary, as long as there are enough pieces.

Procedure: 9.5 *Building a Dictionary for VQ*

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 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: 9.6 *Representing a signal using VQ*

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 9.12.

The number of pieces of signal (and so k), might be very big indeed. It is quite reasonable to want to build a dictionary for a million items and use tens to hundreds of thousands of cluster centers. In this case, it is a good idea to use hierarchical k-means, as in Section 9.2.3. Hierarchical k-means produces a tree of cluster centers. 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.

Representing a signal as a histogram of cluster centers loses information in two important ways. First, the histogram has little or no information about how the pieces of signal are arranged. So, for example, the representation can tell whether an image has stripy or spotty patches in it, but not where those patches lie. You should not rely on your intuition to tell you whether this lost information is important or not. For many kinds of image classification task, histograms of cluster centers are much better than you might guess, despite not encoding where patches lie (though still better results are now obtained with convolutional neural networks).

Second, replacing a piece of signal with a cluster center must lose some detail, which might be important, and likely results in some classification errors. There is a surprisingly simple construction that can alleviate these problems. Build three (or more) dictionaries, rather than one, using different sets of training pieces. For example, you could cut the same signals into pieces on a different grid. Now use each dictionary to produce a histogram of cluster centers, and classify with those.

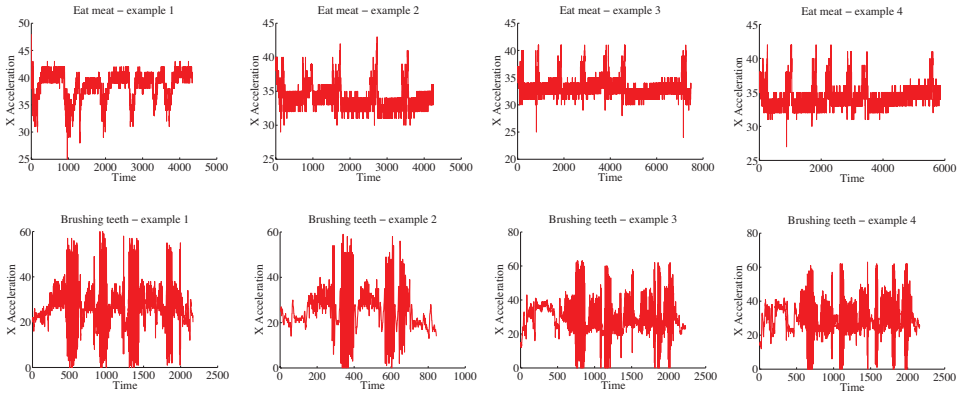


FIGURE 9.13: *Some examples from the accelerometer dataset at https://archive.ics.uci.edu/ml/datasets/Dataset+for+ADL+Recognition+with+Wrist-worn+Accelerometer. I have labelled each signal by the activity. These show acceleration in the X direction (Y and Z are in the dataset, too). There are four examples 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).*

Finally, use a voting scheme to decide the class of each test signal. In many problems, this approach yields small but useful improvements.

9.3.2 Example: Activity from Accelerometer Data

A complex example dataset appears at
https://archive.ics.uci.edu/ml/datasets/Dataset+for+ADL+Recognition+with+Wrist-worn+Accelerometer. 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. The dataset was collected by Barbara Bruno, Fulvio Mastrogiovanni and Antonio Sgorbissa. Figure 9.13 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 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

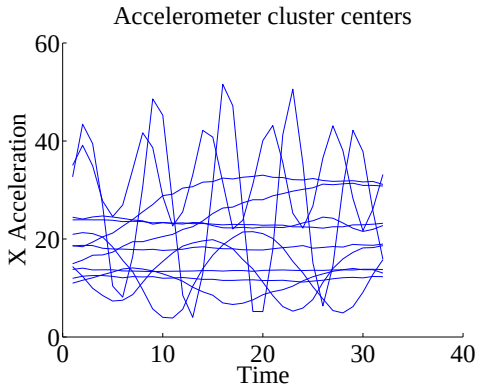


FIGURE 9.14: *Some cluster centers from the accelerometer dataset. Each cluster center represents a one-second burst of activity. There are a total of 480 in my model, which I built using hierarchical k-means. Notice there are a couple of centers that appear to represent movement at about 5Hz; another 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).*

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 segments. I used two levels, with 40 cluster centers at the first level, and 12 at the second. Figure 9.14 shows some cluster centers at the second level.

I then computed histogram representations for different example signals (Figure 9.15). 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

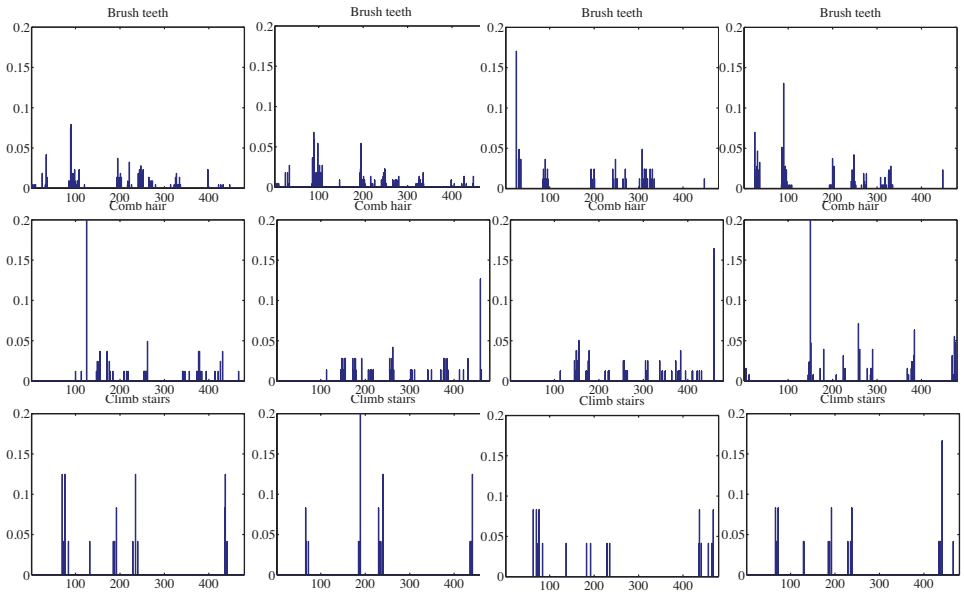


FIGURE 9.15: *Histograms of cluster centers for the accelerometer dataset, for different activities. You should notice that (a) these histograms look somewhat similar for different actors performing the same activity and (b) these histograms look somewhat different for different activities.*

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 9.1 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 get a held-out error rate of 0.79. This suggests that the representation is fairly good at exposing what is important.

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 9.1: *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.*

9.4 YOU SHOULD

9.4.1 remember these definitions:

9.4.2 remember these terms:

9.4.3 remember these facts:

9.4.4 remember these procedures:

PROGRAMMING EXERCISES

9.1. 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?

9.2. Obtain the liver disorder dataset from UC Irvine machine learning website (http://archive.ics.uci.edu/ml/datasets/Liver%20Disorders; data provided by Richard S. Forsyth). The first five values in each row represent various biological measurements, and the sixth is a measure of the amount of alcohol consumed daily. We will analyze this data following the recipe of section 9.2.5. Divide the data into four blocks using the amount of alcohol, where each block is a quantile (so data representing the lowest quarter of consumption rates go into the first block, etc.). Now cluster the data using the first five values and k-means. For each block, compute a histogram of the cluster centers (as in Figure 9.11). Plot these histograms. What do you conclude?

9.3. Obtain the Portugese student math grade dataset from UC Irvine machine learning website (https://archive.ics.uci.edu/ml/datasets/student+performance; data provided by Paulo Cortez). There are two datasets at the URL – you are looking for the one that relates to math grades. Each row contains some numerical attributes (columns 3,7,8,13, 14, 15,24, 25, 26, 27, 28, 29, 30) and some other attributes. We will consider only the numerical attributes. Column 33 contains the grade at the end of the year in numerical form. We will analyze this data following the recipe of section 9.2.5. Divide the data into four blocks using the final grade, where each block is a quantile (so data representing the lowest quarter of grades go into the first block, etc.). Now cluster the data using the numerical values and k-means. For each block, compute a histogram of the cluster centers (as in Figure 9.11). Plot these histograms. What do you conclude?

9.4. 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.

9.5. 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`).

9.6. *This is a substantial exercise.* 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 read-

ers 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. For this exercise, we will use raw pixels in untouched images.

(a) We will use hierarchical k-means to build a dictionary of image patches. For untouched images, construct a collection of 10×10 image patches. You should extract these patches from the training images on an overlapping 4×4 grid, meaning that each training image produces 16 overlapping patches (so you could have 960, 000 training patches!). For each training image, choose one of these patches uniformly and at random. Now subsample this dataset of 60,000 patches uniformly and at random to produce a 6,000 element dataset. Cluster this dataset to 50 centers. Now build 50 datasets, one per cluster center. Do this by taking each element of the 60, 000 patch dataset, finding which of the cluster centers is closest to it, and putting the patch in that center's dataset. Now cluster each of these datasets to 50 centers.

(b) You know have a dictionary of 2,500 entries. For each query image, construct a set of 10×10 patches on an overlapping 4×4 grid. Now for each of the centers, you should extract 9 patches. Assume the center is at (x, y); obtain 9 patches by extracting a patch centered at $(x-1, y-1), (x, y-1), \ldots, (x+1, y+1)$. This means each test image will have 144 associated patches. Now use your dictionary to find the closest center to each patch, and construct a histogram of patches for each test image.

(c) Train a classifier (I'd use a decision forest) using this histogram of patches representation. Evaluate this classifier on the test data.

(d) Can you improve this classifier by modifying how you extract patches, the size of the patches, or the number of centers?

(e) At this point, 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/.

9.7. 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. There are 10 classes, 50, 000 training images, and 10,000 test images.

(a) We will use hierarchical k-means to build a dictionary of image patches. For untouched images, construct a collection of $10 \times 10 \times 3$ image patches. You should extract these patches from the training images at random locations (you don't know where the good stuff in the image is), and you should extract two patches per training image. Now subsample this dataset of 100,000 patches uniformly and at random to produce a 10,000 element dataset. Cluster this dataset to 50 centers. Now build 50 datasets, one per cluster center. Do this by taking each element of the 100, 000 patch dataset, finding which of the cluster centers is closest to it, and putting the patch in that center's dataset. Now cluster each of these datasets to

50 centers.

(b) You know have a dictionary of 2,500 entries. For each query image, construct a set of 10×10 patches. You should extract patches using centers that are on a grid spaced two pixels apart horizontally and vertically, so there will be a lot of overlap between the patches. Now use your dictionary to find the closest center to each patch, and construct a histogram of patches for each test image.

(c) Train a classifier (I'd use a decision forest) using this histogram of patches representation. Evaluate this classifier on the test data.

(d) Can you improve this classifier by modifying how you extract patches, the size of the patches, or the number of centers?

(e) At this point, you're likely tired of CIFAR-10 but very well informed. Compare your methods to Roderigo Benenson's table of methods at http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html.

CHAPTER 10

Clustering using Probability Models

Clustering objects requires some notion of how similar they are. We have seen how to cluster using distance in feature space, which is a natural way of thinking about similarity. Another way to think about similarity is to ask whether two objects have high probability under the same probability model. This can be a convenient way of looking at things when it is easier to build probability models than it is to measure distances. It turns out to be a natural way of obtaining soft clustering weights (which emerge from the probability model). And it provides a framework for our first encounter with an extremely powerful and general algorithm, which you should see as a very aggressive generalization of K-means.

10.1 MIXTURE MODELS AND CLUSTERING

It is natural to think of clustering in the following way. The data was created by a collection of distinct probability models (one per cluster). For each data item, something (nature?) chose which model was to produce a point, and then an IID sample of that model is the point. We see the points: we'd like to know what the models were, but (and this is crucial) 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.

You should notice a resonance with K-means here. In K-means, if we knew the centers, which point belongs to which center would be easy; if we knew which point belongs to which center, the centers would be easy. We dealt with this situation quite effectively by repeatedly fixing one then estimating the other. It is pretty clear that a natural algorithm for dealing with the probability models is to iterate between estimating which model gets which point, and the model parameters. This is the key to 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.

10.1.1 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**, and the π_j terms are usually called **mixing weights**.

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

$$\begin{aligned} \mathcal{L}(\mu_1, \ldots, \mu_k, \pi_1, \ldots, \pi_k; \mathbf{x}, \delta) &= \mathcal{L}(\theta; \mathbf{x}, \delta) \\ &= \sum_{ij} \left\{ \left[\left(-\frac{1}{2}(\mathbf{x}_i - \mu_j)^T(\mathbf{x}_i - \mu_j) \right) \right] + \log \pi_j \right\} \delta_{ij} \\ &\quad + K \end{aligned}$$

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.

10.1.2 Topics and Topic Models

We have already seen that word counts expose similarities between documents (section 7.3). We now assume that documents with similar word counts will come from the same **topic** (mostly, a term of art for cluster used in the natural language processing community). A really useful model is to assume that words are conditionally independent, conditioned on the topic. This means that, once you know the topic, words are IID samples of a multinomial distribution that is given by the topic (the **word probabilities** for that topic). If it helps, you can think of the topic as multi-sided die with a different word on each face. You then make a document by rolling this die – which is likely not a fair die – some number of times.

This model of documents has problems. Word order doesn't matter in this model, nor does where a word appears in a document or what words are near in the document and what others are far away. We've already seen that ignoring word order, word position, and neighbors can still produce useful representations (section 7.3). Despite its problems, this model clusters documents rather well, is easy to work with, and is the basis for more complex models.

A single document is a set of word counts that is obtained by (a) selecting a topic then (b) drawing words as IID samples from that topic. We now have a collection of documents, and we want to know (a) what topic each document came from and (b) the word probabilities for each topic. Now imagine we know which document comes from which topic. Then we could estimate the word probabilities using the documents in each topic by simply counting. In turn, 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 procedure should strike you as being very like k-means, though the details have changed.

To construct a probabilistic model more formally, 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. Each topic is a multinomial probability distribution. The vocabulary is d-dimensional. Write $\mathbf{p}_j$ for the d-dimensional vector of word probabilities for the j'th topic. Now write $\mathbf{x}_i$ for the i'th vector of word counts (there are N vectors in the collection). 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.

10.2 THE EM ALGORITHM

There is a straightforward, natural, and very powerful recipe for estimating θ for both models. 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 57; this is close to 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 an 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

$$\mathcal{Q}(\theta; \theta^{(n)}) = \sum_{\delta} \mathcal{L}(\theta; \mathbf{x}, \delta) p(\delta|\theta^{(n)}, \mathbf{x}) = \mathbb{E}_{p(\delta|\theta^{(n)}, \mathbf{x})}[\mathcal{L}(\theta; \mathbf{x}, \delta)]$$

(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, additive constants may disappear without notice (they do so regularly in the research literature). In the mixture of normals example, below, I've tried to keep track of them; for the mixture of multinomials, I've been looser.

10.2.1 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 t^N 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

$$p(\delta_{ij}=1|\theta^{(n)},\mathbf{x}) = \frac{\left[\exp\left(-\frac{1}{2}(\mathbf{x}_i-\mu_j)^T(\mathbf{x}_i-\mu_j)\right)\right]\pi_j}{\sum_k\left[\exp\left(-\frac{1}{2}(\mathbf{x}_i-\mu_k)^T(\mathbf{x}_i-\mu_k)\right)\right]\pi_k} = w_{ij}.$$

10.2.2 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

$$\mu_j^{(n+1)} = \frac{\sum_i \mathbf{x}_i w_{ij}}{\sum_i w_{ij}}$$

and

$$\pi_j^{(n+1)} = \frac{\sum_i w_{ij}}{N}.$$

You should check these expressions. When you do so, remember that, because π is a probability distribution, $\sum_j \pi_j = 1$ (otherwise you'll get the wrong answer). You need either to use a Lagrange multiplier or to set one probability to (1 − all others).

10.2.3 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 10.2.1. 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

$$p(\delta_{ij}=1|\theta^{(n)},\mathbf{x}) = \frac{\left[\prod_k p_{j,k}^{x_k}\right]\pi_j}{\sum_l\left[\prod_k p_{l,k}^{x_k}\right]\pi_l}$$

10.2.4 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.

10.2.5 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 maximum 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. I have summarized the general algorithm, and the two instances we studied, in boxes below for reference. There are some practical issues.

Procedure: 10.1 *EM*

Given a model with parameters θ, data $\mathbf{x}$, and missing data δ, which gives rise to a log-likelihood $\mathcal{L}(\theta; \mathbf{x}, \delta) = \log P(\mathbf{x}, \delta|\theta)$ and some initial estimate of parameters $\theta^{(1)}$, iterate

- **The E-step:** Obtain

$$\mathcal{Q}(\theta; \theta^{(n)}) = \mathbb{E}_{p(\delta|\theta^{(n)}, \mathbf{x})}[\mathcal{L}(\theta; \mathbf{x}, \delta)]$$

- **The M-step:** Compute

$$\theta^{(n+1)} = \underset{\theta}{\text{argmax}}\ Q(\theta; \theta^{(n)})$$

Diagnose convergence by testing the size of the update to θ.

Procedure: 10.2 *EM for Mixtures of normals: E-step*

Assume $\theta^{(n)} = (\mu_1, \ldots, \mu_t, \pi_1, \ldots, \pi_t)$ is known. Compute weights w_{ij} linking the i'th data item to the j'th cluster center, using

$$w_{ij}^{(n)} = \frac{\left[\exp\left(-\frac{1}{2}(\mathbf{x}_i - \mu_j^{(n)})^T(\mathbf{x}_i - \mu_j^{(n)})\right)\right]\pi_j^{(n)}}{\sum_k \left[\exp\left(-\frac{1}{2}(\mathbf{x}_i - \mu_k^{(n)})^T(\mathbf{x}_i - \mu_k^{(n)})\right)\right]\pi_k^{(n)}}$$

Procedure: 10.3 *EM for Mixtures of normals: M-step*

Assume $\theta^{(n)} = (\mu_1, \ldots, \mu_t, \pi_1, \ldots, \pi_t)$ and weights w_{ij} linking the i'th data item to the j'th cluster center, are known. Then estimate

$$\mu_j^{(n+1)} = \frac{\sum_i \mathbf{x}_i w_{ij}^{(n)}}{\sum_i w_{ij}^{(n)}}$$

and

$$\pi_j^{(n+1)} = \frac{\sum_i w_{ij}^{(n)}}{N}.$$

Procedure: 10.4 *EM for topic models: E-step*

Assume $\theta^{(n)} = (\mathbf{p}_1, \ldots, \mathbf{p}_t, \pi_1, \ldots, \pi_t)$ is known. Compute weights $w_{ij}^{(n)}$ linking the i'th data item to the j'th cluster center, using

$$w_{ij}^{(n)} = \frac{\left[\prod_k \left(p_{j,k}^{(n)}\right)^{x_k}\right]\pi_j^{(n)}}{\sum_l \left[\prod_k \left(p_{j,k}^{(n)}\right)^{x_k}\right]\pi_l^{(n)}}$$

Procedure: 10.5 *EM for topic models: M-step*

Assume $\theta^{(n)} = (\mathbf{p}_1, \ldots, \mathbf{p}_t, \pi_1, \ldots, \pi_t)$ and weights $w_{ij}^{(n)}$ linking the i'th data item to the j'th cluster center are known. Then estimate

$$\mathbf{p}_j^{(n+1)} = \frac{\sum_i \mathbf{x}_i w_{ij}^{(n)}}{\sum_i \mathbf{x}_i^T \mathbf{1} w_{ij}^{(n)}}$$

and

$$\pi_j^{(n+1)} = \frac{\sum_i w_{ij}^{(n)}}{N}.$$

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 $e^{-d^2_{\min}/2}$ terms cancel top and bottom).

The last problem is more substantial. EM will get 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.

Remember this: *You should use the same approach to choosing the number of cluster centers with EM as you use with k-means (try a few different values, and see which yields the most useful clustering). You should initialize an EM clusterer with k-means, but be careful of initial probabilities that are zero when initializing a topic model. You should be careful when computing weights, as it is easy to have numerical problems. Finally, it's a good idea to start EM clustering at multiple start points.*

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.

There is an important, rather embarrassing, secret about EM. In practice, it isn't usually that much better as a clustering algorithm than k-means. You can only really expect improvements in performance if it is really important that many points can make a contribution to multiple cluster centers, and this doesn't happen very often. For a dataset where this does apply, the data itself may not really be an IID draw from a mixture of normal distributions, so the weights you compute are only approximate. Usually, it is smart to start EM with k-means. Nonetheless, EM is an algorithm you should know, because it is very widely applied in other situations, and because it can cluster data in situations where it isn't obvious how you compute distances.

Useful Facts: 10.1 *EM clusterers aren't much better than k-means clusterers, but EM is very general. It is a procedure for estimating the parameters of a probability model in the presence of missing data; this is a scenario that occurs in many applications. In clustering, the missing data was which data item belonged to which cluster.*

EM is a quite general algorithm

10.3 YOU SHOULD

10.3.1 remember these terms:

10.3.2 remember these facts:

EM clusterers aren't much better than k-means clusterers, but EM is very general. It is a procedure

10.3.3 remember these procedures:

10.3.4 be able to

- Use EM to cluster points using a mixture of normals model.
- Cluster documents using EM and a topic model.

PROBLEMS

10.1. You will derive the expressions for the M step for mixture of normal clustering. 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. Show that

$$\mu_j^{(n+1)} = \frac{\sum_i \mathbf{x}_i w_{ij}}{\sum_i w_{ij}}$$

and

$$\pi_j^{(n+1)} = \frac{\sum_i w_{ij}}{N}.$$

maximize $\mathcal{Q}$. When you do so, remember that, because π is a probability distribution, $\sum_j \pi_j = 1$ (otherwise you'll get the wrong answer). You need either to use a Lagrange multiplier or to set one probability to (1 − all others).

10.2. You will derive the expressions for the M step for topic models. 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. Show that

$$\mathbf{p}_j^{(n+1)} = \frac{\sum_i \mathbf{x}_i w_{ij}}{\sum_i \mathbf{x}_i^T \mathbf{1} w_{ij}}$$

and

$$\pi_j^{(n+1)} = \frac{\sum_i w_{ij}}{N}.$$

When you do so, remember that, because π is a probability distribution, $\sum_j \pi_j = 1$ (otherwise you'll get the wrong answer). Furthermore, the $\mathbf{p}_j$ are all probability distributions. You need either to use Lagrange multipliers or to set one probability to (1 − all others).

PROGRAMMING EXERCISES

10.3. Image segmentation is an important application of clustering. One breaks an image into k segments, determined by color, texture, etc. These segments are obtained by clustering image pixels by some representation of the image around the pixel (color, texture, etc.) into k clusters. Then each pixel is assigned to the segment corresponding to its cluster center.

(a) Obtain a color image represented as three arrays (red, green and blue). You should look for an image where there are long scale color gradients (a sunset is a good choice). Ensure that this image is represented so the darkest pixel takes the value $(0, 0, 0)$ and the lightest pixel takes the value $(1, 1, 1)$. Now assume the pixel values have covariance the identity matrix. Cluster its pixels into 10, 20, and 50 clusters, modelling the pixel values as a mixture of normal distributions and using EM. Display the image obtained by replacing each pixel with the mean of its cluster center. What do you see?

(b) The weights linking an image to a cluster center can be visualized as an image. For the case of 10 cluster centers, construct a figure showing the weights linking each pixel to each cluster center (all 10 images). You should notice that the weights linking a given pixel to each cluster center do not vary very much. Why?

(c) Now repeat the previous two subexercises, but now using $0.1 \times \mathcal{I}$ as the covariance matrix. Show the new set of weight maps. What has changed, and why?

(d) Now estimate the covariance of pixel values by assuming that pixels are normally distributed (this is somewhat in tension with assuming they're distributed as a mixture of normals, but it works). Again, cluster the image's pixels into 10, 20, and 50 clusters, modelling the pixel values as a mixture of normal distributions and using EM, but now assuming that each normal distribution has the covariance from your estimate. Display the image obtained by replacing each pixel with the mean of its cluster center. Compare this result from the result of the first exercise. What do you see?

10.4. If you have a careful eye, or you chose a picture fortunately, you will have noticed that the previous exercise can produce image segments that have many connected components. For some applications, this is fine, but for others, we want segments that are compact clumps of pixels. One way to achieve this is to represent each pixel with 5D vector, consisting of its RG and B values *and* its x and y coordinates. You then cluster these 5D vectors.

(a) Obtain a color image represented as three arrays (red, green and blue). You should look for an image where there are many distinct colored objects (for example, a bowl of fruit). Ensure that this image is represented so the darkest pixel takes the value $(0, 0, 0)$ and the lightest pixel takes the value $(1, 1, 1)$. Represent the x and y coordinates of each pixel using the range 0 to 1 as well. Now assume the pixel RGB values have covariance 0.1 times the identity matrix, there is zero covariance between position and color, and the coordinates have covariance σ times the identity matrix where σ is a parameter we will modify. Cluster your image's pixels into 20, 50, and 100 clusters, with $\sigma = (0.01, 0.1, 1)$ (so 9 cases). Again, model the pixel values as a mixture of normal distributions and using EM. For each case, display the image obtained by replacing each pixel with the mean of its cluster center. What do you see?

10.5. EM has applications that don't look like clustering at first glance. Here is one. We will use EM to reject points that don't fit a line well (if you haven't seen least squares line fitting, this exercise isn't for you).

(a) Construct a dataset of 10 2D points which are IID samples from the following mixture distribution. Draw the x coordinate from the uniform distribution on the range $[0, 10]$. With probability 0.8, draw ξ a normal random variable with mean 0 and standard deviation 0.001 and form the y coordinate as $y = x + \xi$. With probability 0.2, draw the y coordinate from the uniform distribution on the range $[0, 10]$. Plot this dataset – you should see about 8 points on a line with about 2 scattered points.

(b) Fit a least squares line to your dataset, and plot the result. It should be bad, because the scattered points may have a significant effect on the line. If you were unlucky, and drew a sample where there were no scattered points or where this line fits well, keep drawing datasets until you get one where the fit is poor.

(c) We will now use EM to fit a good line. Write $N(\mu, \sigma)$ for a normal distribution with mean μ and standard deviation σ, and $U(0, 10)$ for the uniform distribution on the range $0 - 10$. Model the y coordinate using the mixture model $P(y|a, b, \pi, x) = \pi N(ax + b, 0.001) + (1 - \pi)U(0, 10)$. Now associate a variable δ_i with the i'th datapoint, where $\delta_i = 1$ if the datapoint comes from the line model and $\delta_i = 0$ otherwise. Write an expression for $P(y_i, \delta_i|a, b, \pi, x)$.

(d) Assume that $a^{(n)}$, $b^{(n)}$ and $\pi^{(n)}$ are known. Show that

$$Q(a, b, \pi; a^{(n)}, b^{(n)}, \pi^{(n)}) = -\sum_i w_i \frac{(ax_i + b - y_i)^2}{20.001^2} + (1 - w_i)(1/10) + K$$

(where K is a constant). Here

$$w_i = \mathbb{E}_{P(\delta_i|a^{(n)}, b^{(n)}, \pi^{(n)}, x)}[\delta_i]$$

(e) Show that

$$w_i = P(\delta_i|a^{(n)}, b^{(n)}, \pi^{(n)}, x) = \frac{\pi^{(n)} e^{-\frac{(a^{(n)}x_i + b^{(n)} - y_i)^2}{20.001^2}}}{\pi^{(n)} e^{-\frac{(a^{(n)}x_i + b^{(n)} - y_i)^2}{20.001^2}} + (1 - \pi^{(n)})\frac{1}{10}}$$

(f) Now implement an EM algorithm using this information, and estimate the line for your data. You should try multiple start points. Do you get a better line fit? Why?

10.6. This is a 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`).

10.7.

PART FOUR

REGRESSION

CHAPTER 11

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.

11.1 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 11.1 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 11.1). 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 $P(X)$,

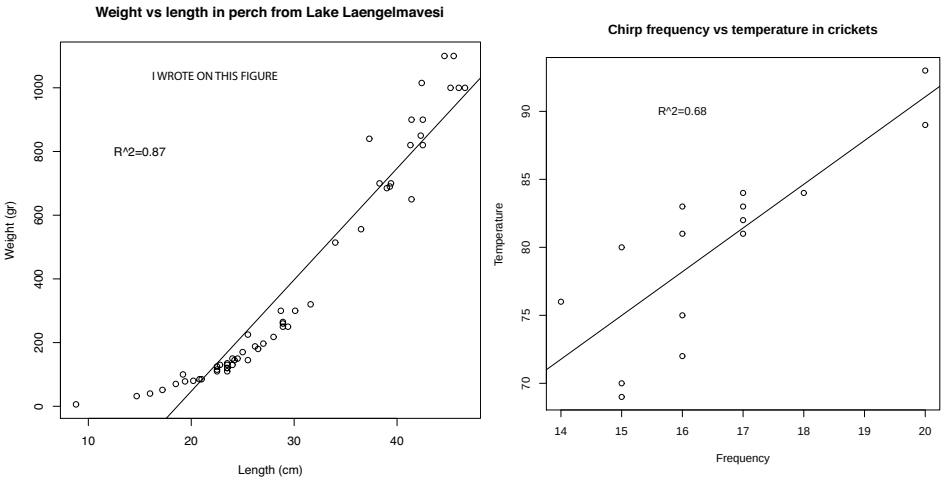


FIGURE 11.1: *On the* **left**, *a regression of weight against length for perch from a Finnish lake (you can find this dataset, and the back story at* http://www.amstat.org/publications/jse/jse_data_archive.htm*; look for "fishcatch" on that page). Notice that the linear regression fits the data fairly 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 11.2.4).*

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

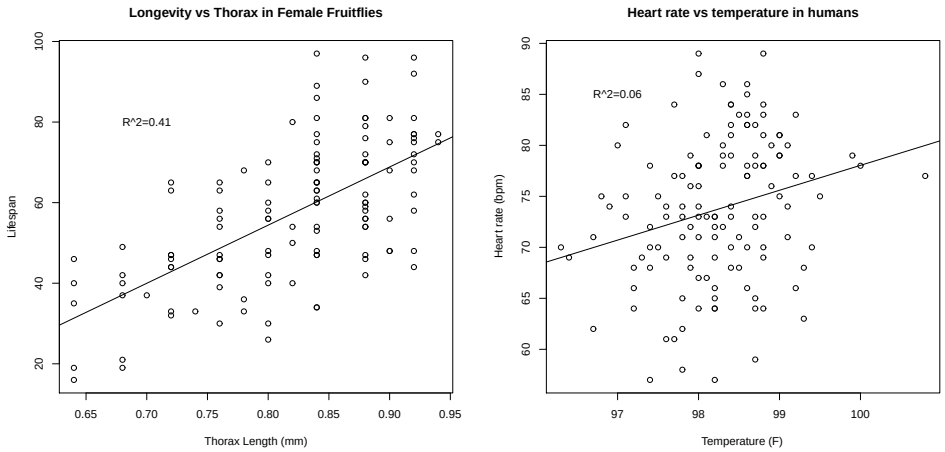


FIGURE 11.2: *Regressions do not necessarily yield good predictions or good model fits. On the* **left**, *a regression of the lifespan of female fruitflies against the length of their torso as adults (apparently, this doesn't change as a fruitfly ages; you can find this dataset, and the back story at 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 $\mathbf{x}$.

11.1.1 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.

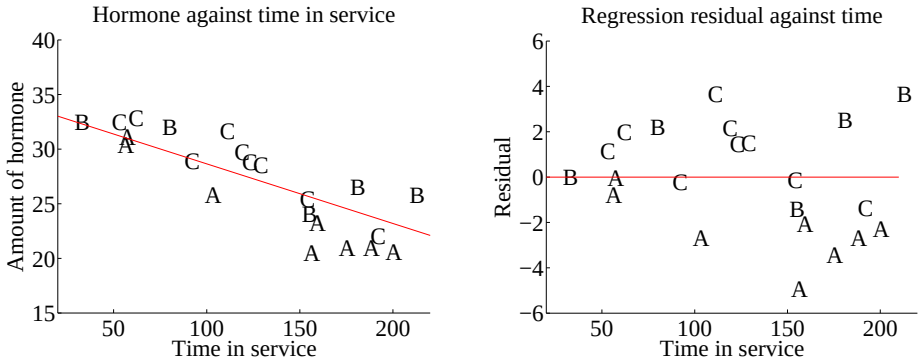


FIGURE 11.3: *On the* **left**, *a scatter plot of hormone against time for devices from tables 11.1 and 11.1. Notice that there is a pretty clear relationship between time and amount of hormone (the longer the device has been in service the less hormone there is). The issue now is to understand that 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 11.2. 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 11.1 and 11.1. 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 11.3 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 11.3). In this case, the plot suggests that lot A is special — all devices from this lot contain less hormone than our model predicts.

Definition: 11.1 *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.

11.2 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.

11.2.1 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: 11.2 *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: 11.1 *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: 11.2 *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.

11.2.2 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 $P(y|\mathbf{x}, \beta)$ 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 11.4.2.

Listing 11.1: R code used for the linear regression example of worked example 11.1

```
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 11.1 *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 11.1. 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.

11.2.3 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.

11.2.4 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: 11.1 *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

$$\begin{aligned}
\mathsf{var}[y] &= (1/N)\left(\left[\mathcal{X}\hat{\beta} - \overline{\mathcal{X}\hat{\beta}}\right] + [\mathbf{e}-\overline{\mathbf{e}}]\right)^T\left(\left[\mathcal{X}\hat{\beta} - \overline{\mathcal{X}\hat{\beta}}\right] + [\mathbf{e}-\overline{\mathbf{e}}]\right)\\
&= (1/N)\left(\left[\mathcal{X}\hat{\beta} - \overline{\mathcal{X}\hat{\beta}}\right]^T\left[\mathcal{X}\hat{\beta} - \overline{\mathcal{X}\hat{\beta}}\right] + 2\,[\mathbf{e}-\overline{\mathbf{e}}]^T\left[\mathcal{X}\hat{\beta} - \overline{\mathcal{X}\hat{\beta}}\right] + [\mathbf{e}-\overline{\mathbf{e}}]^T[\mathbf{e}-\overline{\mathbf{e}}]\right)\\
&= (1/N)\left(\left[\mathcal{X}\hat{\beta} - \overline{\mathcal{X}\hat{\beta}}\right]^T\left[\mathcal{X}\hat{\beta} - \overline{\mathcal{X}\hat{\beta}}\right] + [\mathbf{e}-\overline{\mathbf{e}}]^T[\mathbf{e}-\overline{\mathbf{e}}]\right)\\
&\quad \text{because } \overline{\mathbf{e}} = 0 \text{ and } \mathbf{e}^T\mathcal{X}\hat{\beta} = 0 \text{ and } \mathbf{e}^T\mathbf{1} = 0\\
&= \mathsf{var}\left[\mathcal{X}\hat{\beta}\right] + \mathsf{var}[e].
\end{aligned}$$

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 11.3 has an R^2 value of 0.87. Figures 11.1 and 11.2 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.*

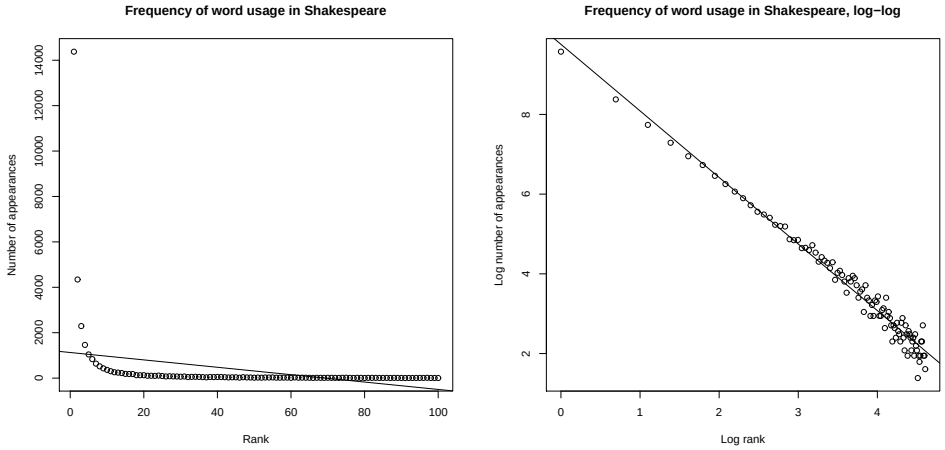


FIGURE 11.4: *On the* **left**, *word count plotted against rank for the 100 most common words in Shakespeare, using a dataset that comes with R (called "bard", and quite likely originating in an unpublished report by J. Gani and I. Saunders). I show a regression line too. This is a poor fit by eye, 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.*

11.2.5 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 11.4 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 11.4, 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 11.4 — 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 11.1. It isn't natural to plot weight$^{0.303}$ against height,

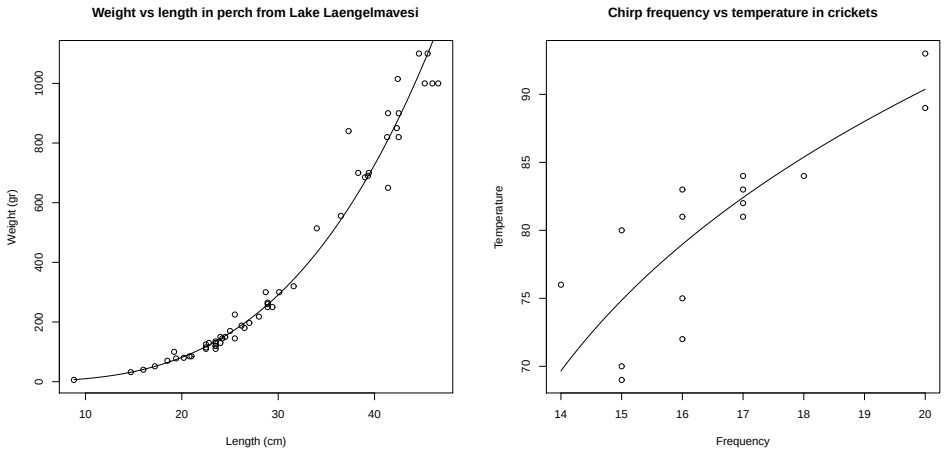


FIGURE 11.5: *The Box-Cox transformation suggests a value of $\lambda = 0.303$ for the regression of weight against height for the perch data of Figure 11.1. You can find this dataset, and the back story at http://www.amstat.org/publications/jse/jse_data_archive.htm; look for "fishcatch" on that page). 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 11.5 shows the result of these transforms.

11.2.6 Can you Trust Your Regression?

Linear regression is useful, but it isn't magic. Some regressions make poor predictions (recall the regressions of figure 11.2). 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 11.1 and Figure 11.2).

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 11.7). In the case of Figure 11.7, 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 11.8).

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: 11.1 *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.

11.3 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.

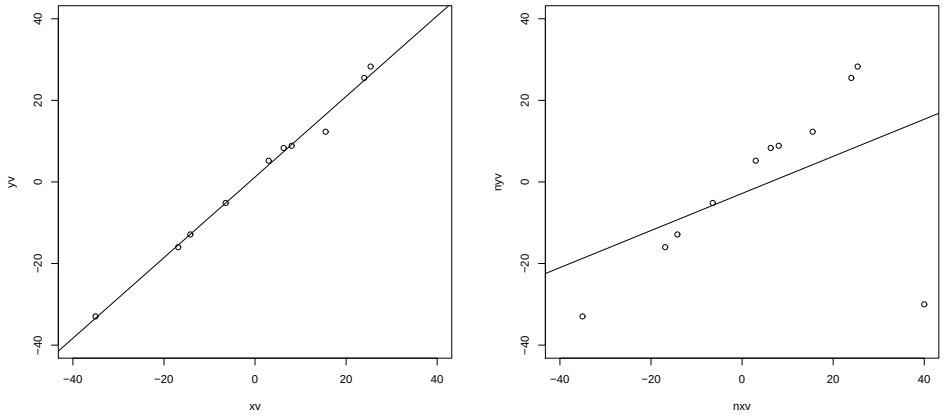


FIGURE 11.6: *On the* **left**, *a synthetic dataset with one independent and one explanatory variable, with the regression line plotted. Notice the line is close to the data points, and its predictions seem likely to be reliable. On the* **right**, *the result of adding a single outlying datapoint to 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.*

11.3.1 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 11.6 illustrates, this means that a data point

that lies far from the others can swing the regression line significantly.

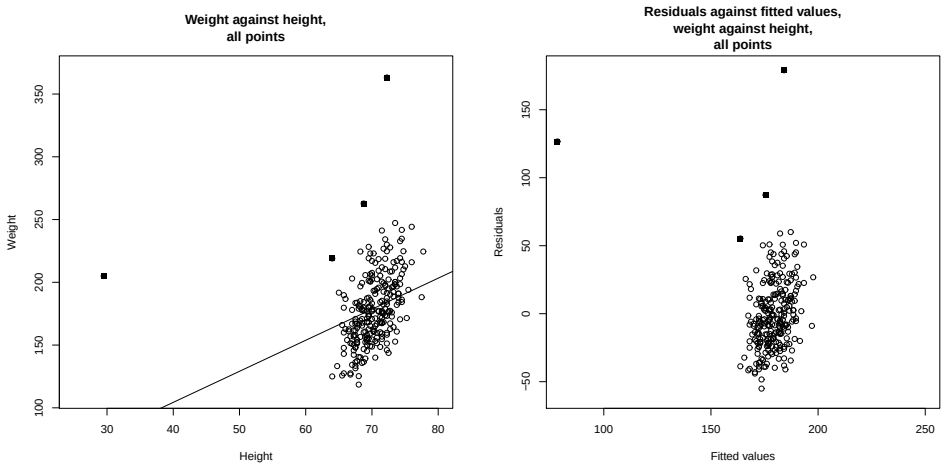


FIGURE 11.7: *On the* **left**, *weight regressed against height for the bodyfat dataset. The line doesn't describe the data particularly well, because it has been strongly affected by a few data points (filled-in markers). On the* **right**, *a scatter plot of the residual against the value predicted 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 11.8 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

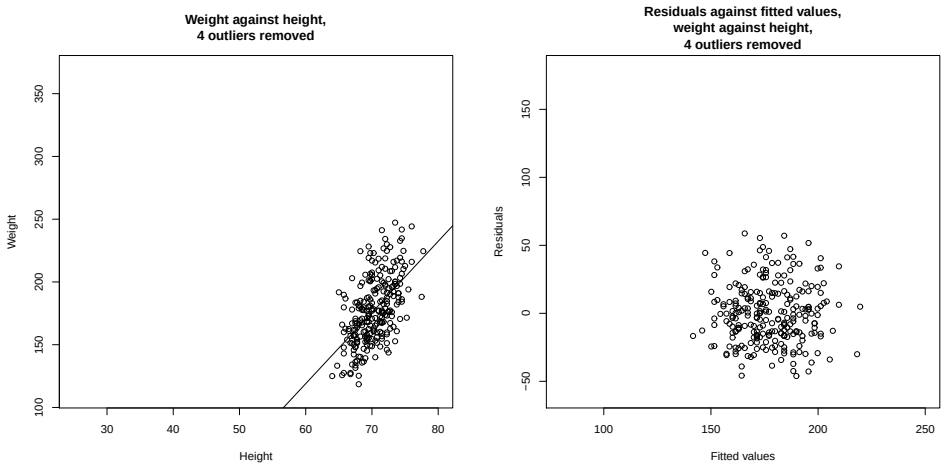


FIGURE 11.8: *On the* **left**, *weight regressed against height for the bodyfat dataset. I have now removed the four suspicious looking data points, identified in Figure 11.7 with filled-in markers; these seemed the most likely to be outliers. On the* **right**, *a scatter plot of the residual against 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.*

11.3.2 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.*

11.3.3 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 35

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: 11.2 *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 11.1), 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.*

11.3.4 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 11.2.1, 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,

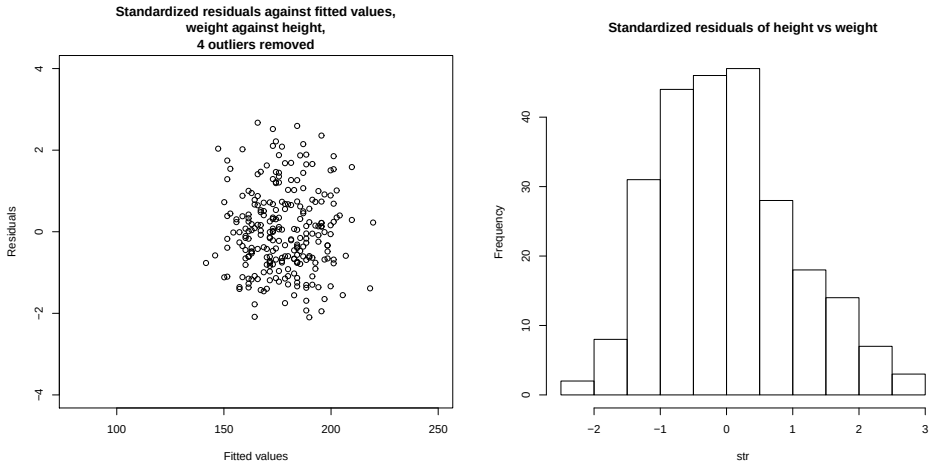


FIGURE 11.9: *On the* **left**, *standardized residuals plotted against predicted value for weight regressed against height for the bodyfat dataset. I removed the four suspicious looking data points, identified in Figure 11.7 with filled-in markers ; these seemed the most likely to be outliers. You should compare this plot with the residuals in figure 11.8, 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 11.10 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

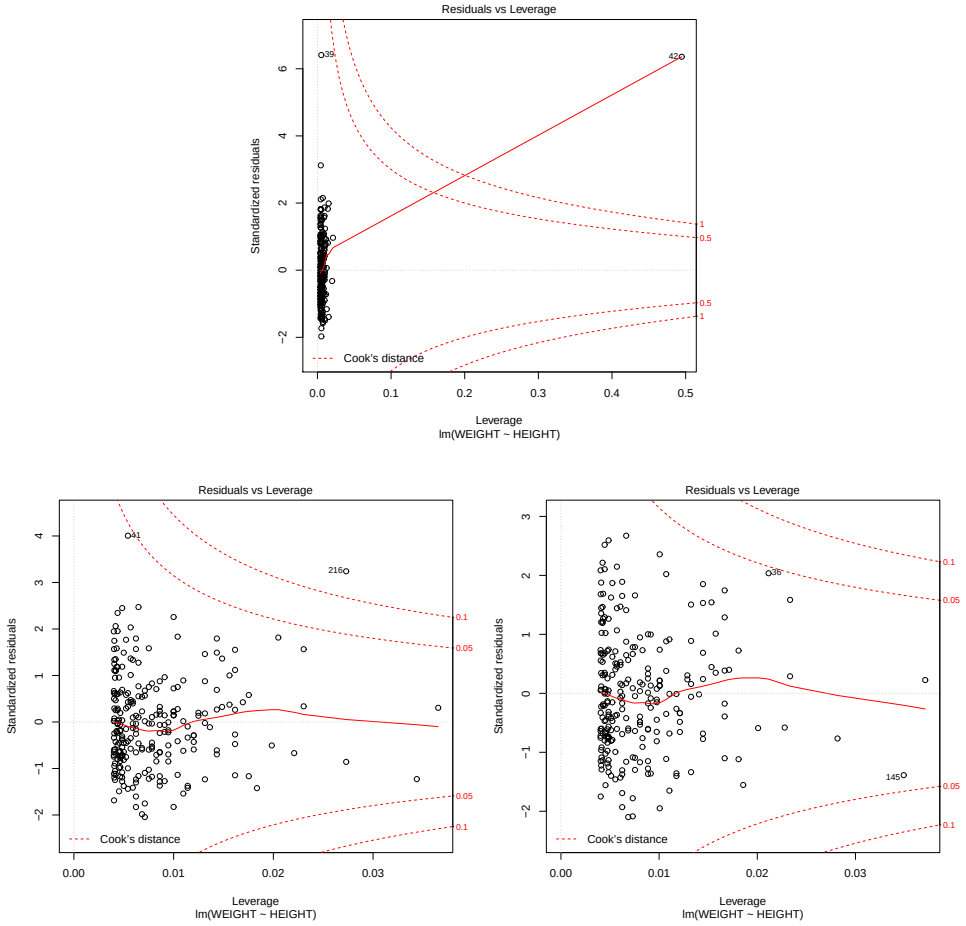


FIGURE 11.10: *A diagnostic plot, produced by R, of a linear regression of weight against height for the bodyfat dataset.* **Top:** *the whole dataset;* **bottom left:** *with the two most extreme points in the top figure removed;* **bottom right:** *with two further points (highest residual) removed. Details in text.*

high 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).

11.4 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 11.4.1), we can add ex-

planatory 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 11.4.2).

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.

11.4.1 Functions of One Explanatory Variable

Imagine we have only one measurement to form explanatory variables. For example, in the perch data of Figure 11.1, 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 11.1 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 11.11.

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 12.1), though it's sometimes obvious that the model is untrustworthy (Figure 11.11).

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.*

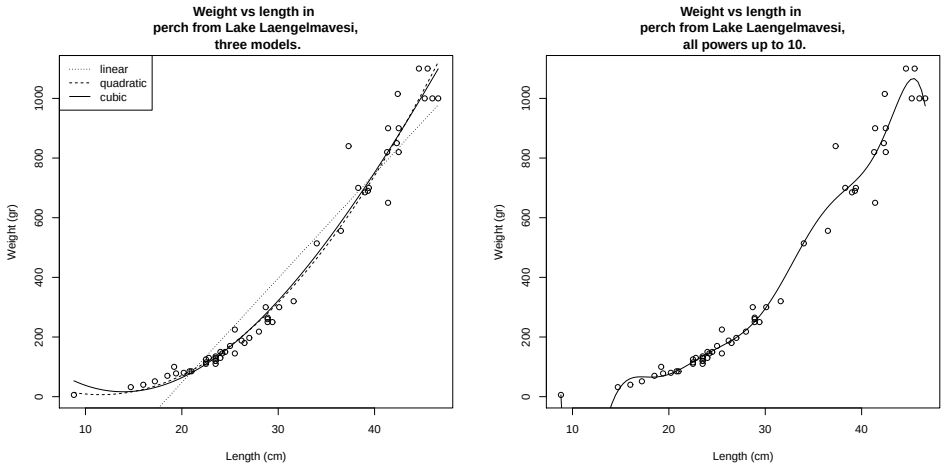


FIGURE 11.11: *On the* **left**, *several different models predicting fish weight from length. The line uses the explanatory variables* 1 *and* x_i*; and the curves use other monomials in* x_i *as well, as shown by the legend. This allows the models to predict curves that lie closer to the data. It is 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.*

11.4.2 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

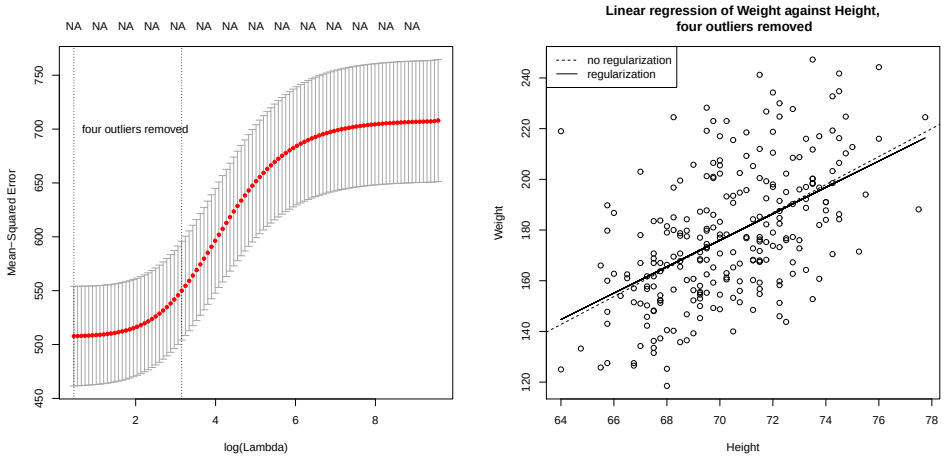


FIGURE 11.12: *On the* **left**, *cross-validated error estimated for different choices of regularization constant for a linear regression of weight against height for the bodyfat dataset, with four outliers removed. The horizontal axis is log regression constant; the vertical is cross-validated error. 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.

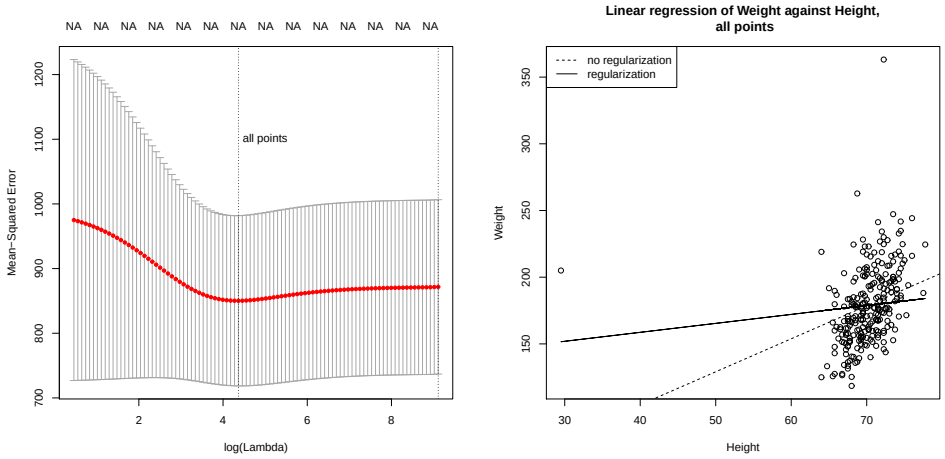


FIGURE 11.13: *Regularization doesn't make outliers go away. On the* **left**, *cross-validated error estimated for different choices of regularization constant for a linear regression of weight against height for the bodyfat dataset, with all points. The horizontal axis is log regression constant; the 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

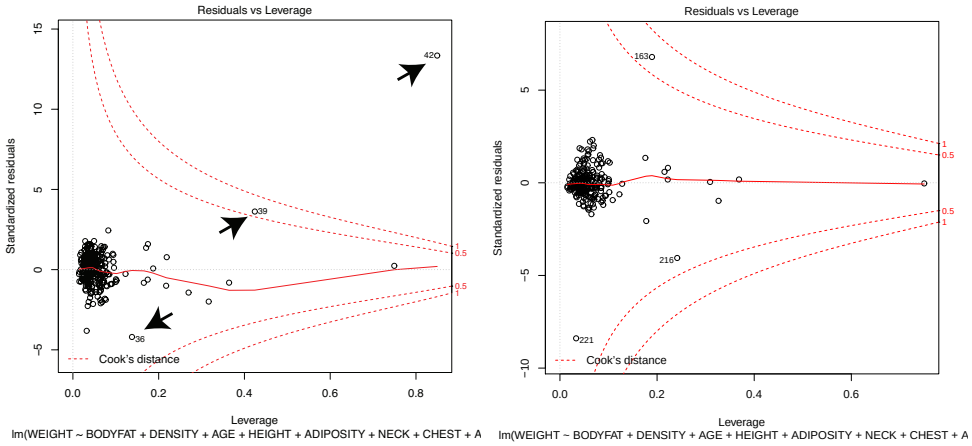


FIGURE 11.14: *On the* **left**, *residuals plotted against leverage for a regression of weight against all other measurements for the bodyfat dataset. I did* not *remove the outliers. The contours on the plot are contours of Cook's distance; I have overlaid arrows showing points with suspiciously large 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 11.12 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 57, I describe the consequences of using other norms.

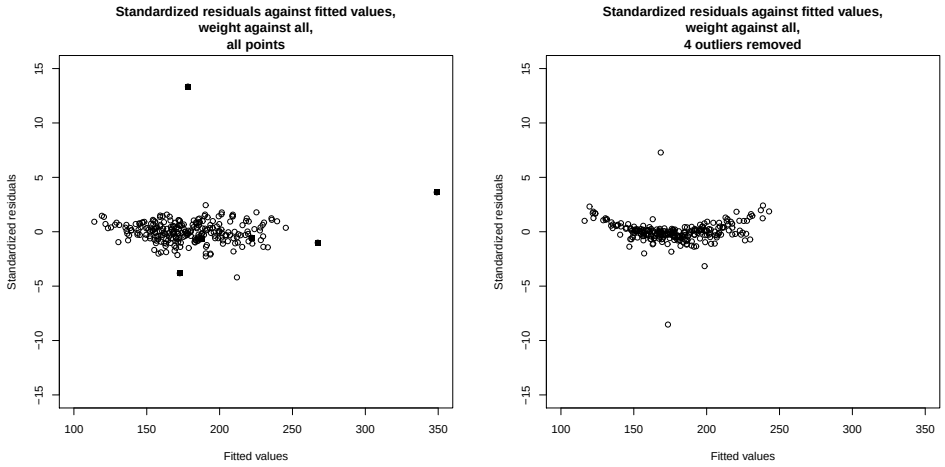


FIGURE 11.15: *On the* **left**, *standardized residuals plotted against predicted value for weight regressed against all variables for the bodyfat dataset. Four data points appear suspicious, and I have marked these with a filled in marker. On the* **right**, *standardized residuals plotted against 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.*

11.4.3 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 11.14 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 11.9). 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 11.15. This is banana-shaped, which is suspicious. There are two points that seem to come from some

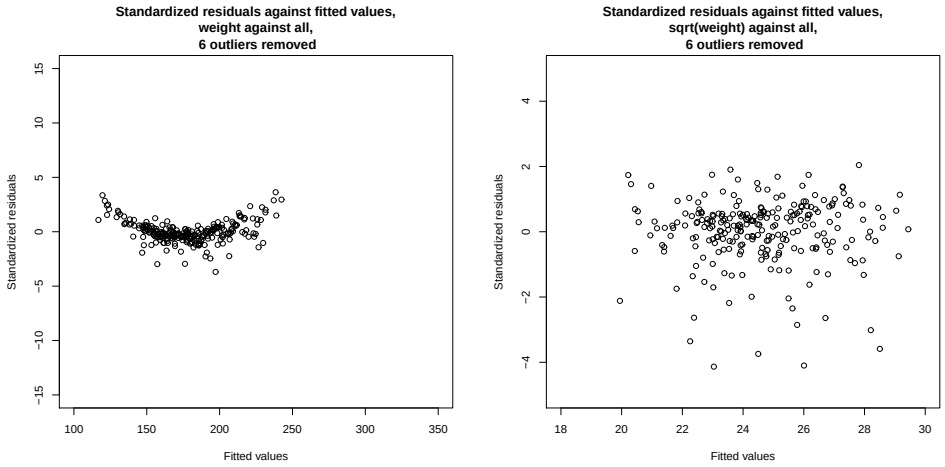


FIGURE 11.16: *On the* **left**, *standardized residuals plotted against predicted value for weight regressed against all variables for the bodyfat dataset. I removed the four suspicious data points of Figure 11.15, and the two others identified in that figure. Notice a suspicious "banana" shape – the 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 11.16.

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 11.16).

Choosing a regularizing value: Figure 11.17 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

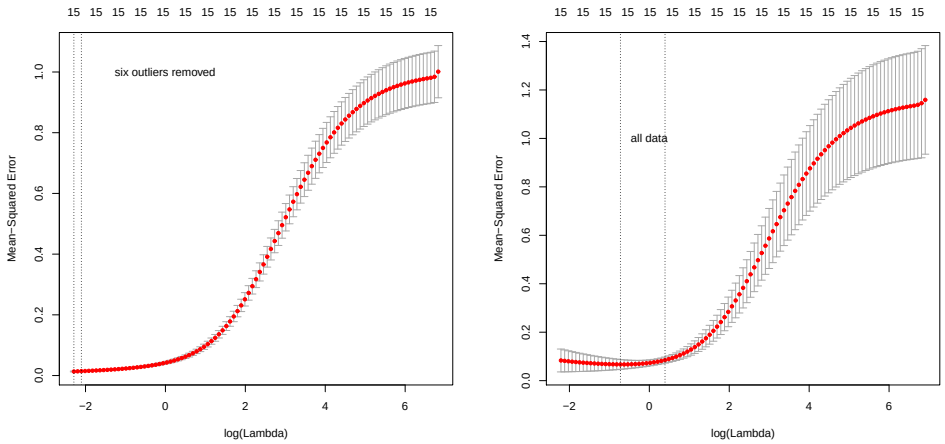


FIGURE 11.17: *Plots of mean-squared error as a function of log regularization parameter (i.e.* $\log \lambda$*) for a regression of weight*$^{1/2}$ *against all variables for the bodyfat dataset. These plots show mean-squared error averaged over cross-validation folds with a vertical one standard deviation bar. On the* **left***, the plot for the dataset with the six outliers identified in Figure 57 removed. On the* **right***, the plot for the whole dataset. Notice how the outliers increase the variability of the error, and the best error.*

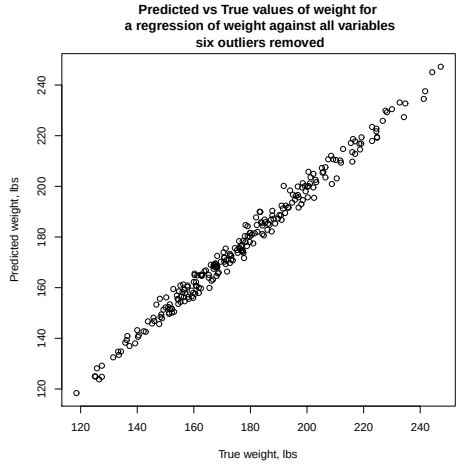


FIGURE 11.18: *A scatter plot of the predicted weight against the true weight for the bodyfat dataset. The prediction is made with all variables, but the six outliers identified above are omitted. I used a Box-Cox transformation with parameter* $1/2$*, and the regularization parameter that yielded the smallest mean square error in Figure 11.17.*

predictions? We already have some information on this point. Figure 11.17 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 11.17, 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 11.18 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 11.17 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.

11.5 YOU SHOULD

11.5.1 remember these definitions:

11.5.2 remember these terms:

11.5.3 remember these facts:

11.5.4 remember these procedures:

11.5.5 be able to:

-

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 11.1: *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

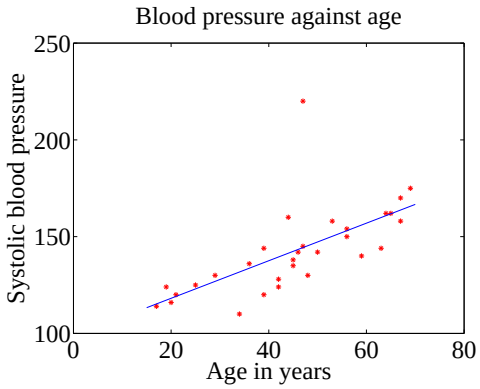


FIGURE 11.19: *A regression of blood pressure against age, for 30 data points.*

11.1. Figure 11.19 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)?

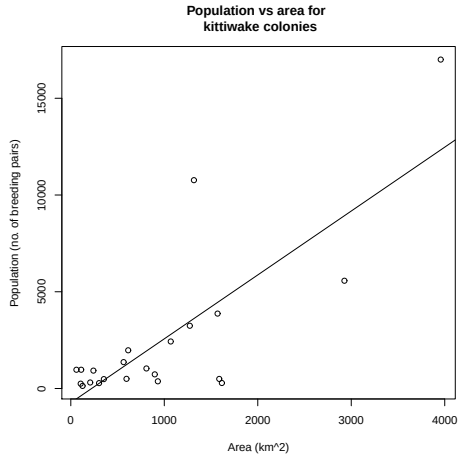


FIGURE 11.20: *A regression of the number of breeding pairs of kittiwakes against the area of an island, for 22 data points.*

11.2. 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 11.20 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?

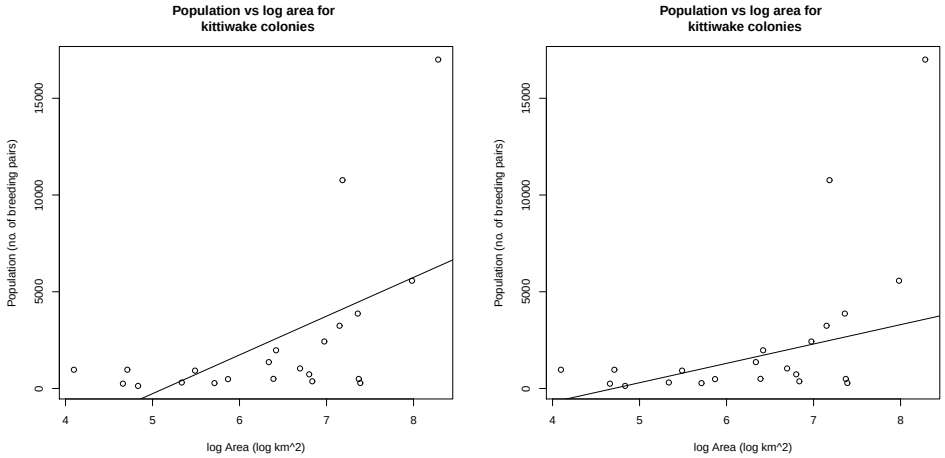


FIGURE 11.21: **Left:** *A regression of the number of breeding pairs of kittiwakes against the log of area of an island, for 22 data points.* **Right:** *A regression of the number of breeding pairs of kittiwakes against the log of area of an island, for 22 data points, using a method that ignores two likely outliers.*

11.3. 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 11.21 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 11.21 shows the result of a linear regression that ignores two likely outliers. Would you trust the predictions of this regression more? Why?

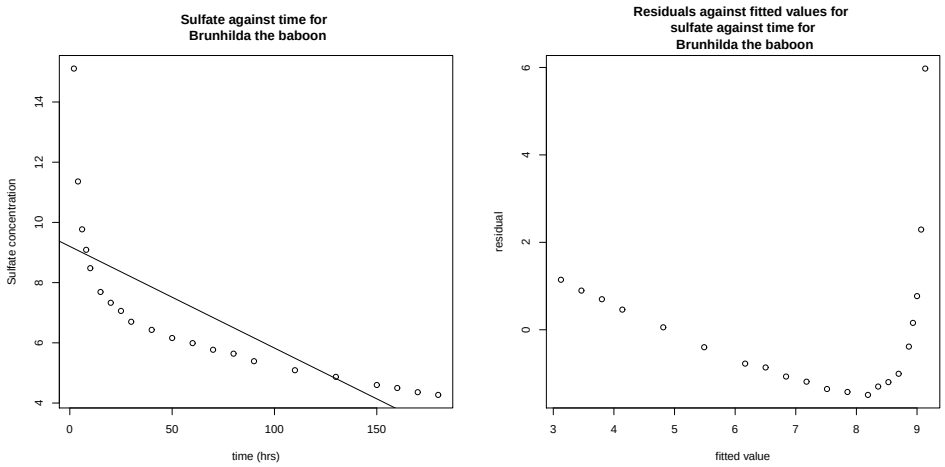


FIGURE 11.22: **Left:** *A regression of the concentration of sulfate in the blood of Brunhilda the baboon against time.* **Right:** *For this regression, a plot of residual against fitted value.*

11.4. 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 11.22 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?

11.5. 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.

11.6. 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.

11.7. 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 $\hat{\beta}$.

11.8. 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

11.9. 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.

11.10. 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.

11.11. 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 12

Regression: Choosing and Managing Models

12.1 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 11.4.1). 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 12.1.1. In the following sections, we describe straightforward methods of doing so.

12.1.1 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 11.4.1 and Figure 11.11. 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.

12.1.2 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 12.1. 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

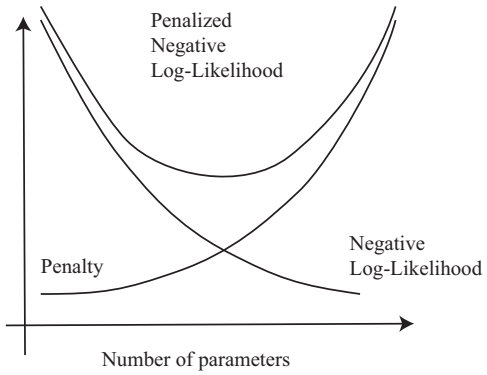


FIGURE 12.1: *When we add explanatory variables (and so parameters) to a model, the value of the negative log-likelihood of the best model can't go up, and usually goes down. This means that we cannot use the value as a guide to how many explanatory variables there should be. Instead, we add a penalty 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 12.1 *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 $d = 2$.

12.1.3 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 12.2 *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 $d = 2$.

12.1.4 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.

12.1.5 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 57 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.

12.2 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**.

12.2.1 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

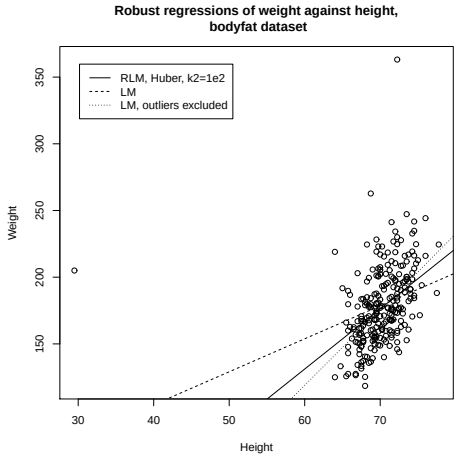


FIGURE 12.2: *Comparing three different linear regression strategies on the bodyfat data, regressing weight against height. Notice that using an M-estimator gives an answer very like that obtained by rejecting outliers by hand. The answer may well be "better" because it isn't certain that each of the 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

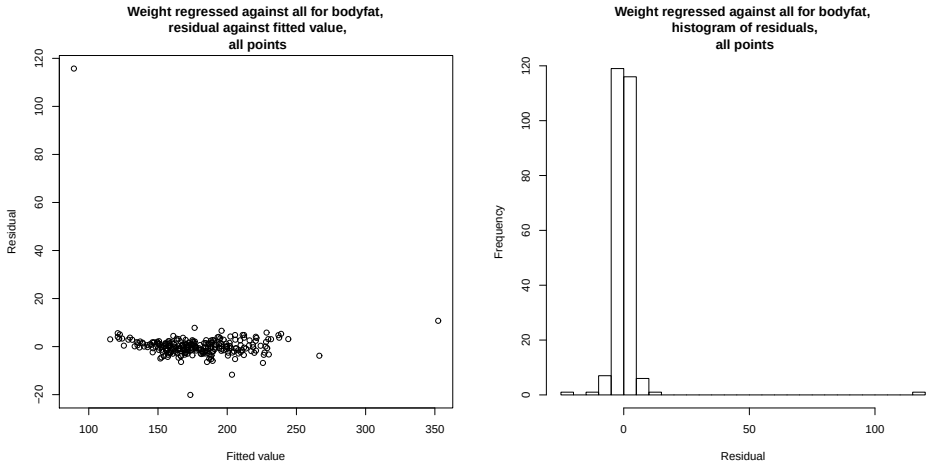


FIGURE 12.3: *A robust linear regression of weight against all variables for the bodyfat dataset, using the Huber loss and all data points. On the* **left**, *residual plotted against fitted value (the residual is not standardized). Notice that there are some points with very large residual, but most 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.

12.2.2 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.

12.3 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.

12.3.1 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: 12.1 *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.

12.3.2 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.

12.3.3 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: 12.2 *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.

12.3.4 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 57), 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.

12.4 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 11.4.2) 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.

12.4.1 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 11.4.2, 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 12.4 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 12.4.

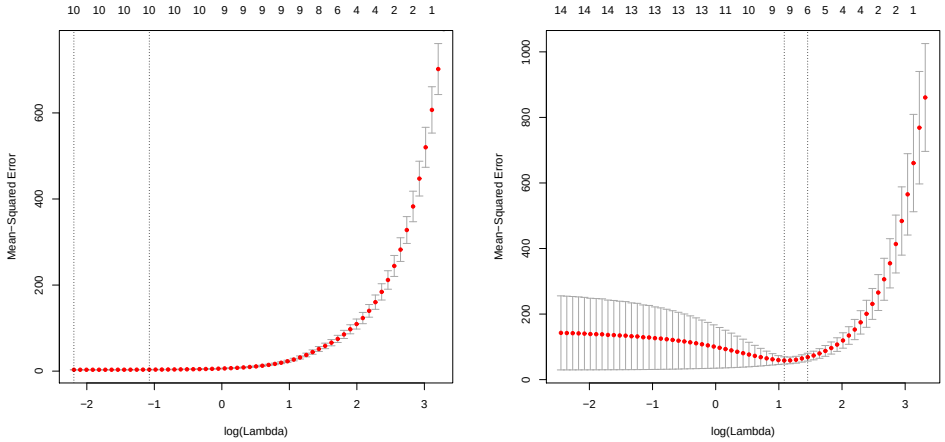


FIGURE 12.4: *Plots of mean-squared error as a function of log regularization parameter (i.e.* $\log \lambda$*) for a regression of weight against all variables for the bodyfat dataset using an L1 regularizer (i.e. a lasso). These plots show mean-squared error averaged over cross-validation folds with a vertical one standard deviation bar. On the* **left***, the plot for the dataset with the six outliers identified in Figure 57 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 12.3 *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 12.7 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 12.1.

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 12.5 shows plots of coefficient values as a function of $\log \lambda$ for a

Listing 12.1: R code used for the lasso regression example of worked example 12.4

```
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 12.4 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 12.6 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

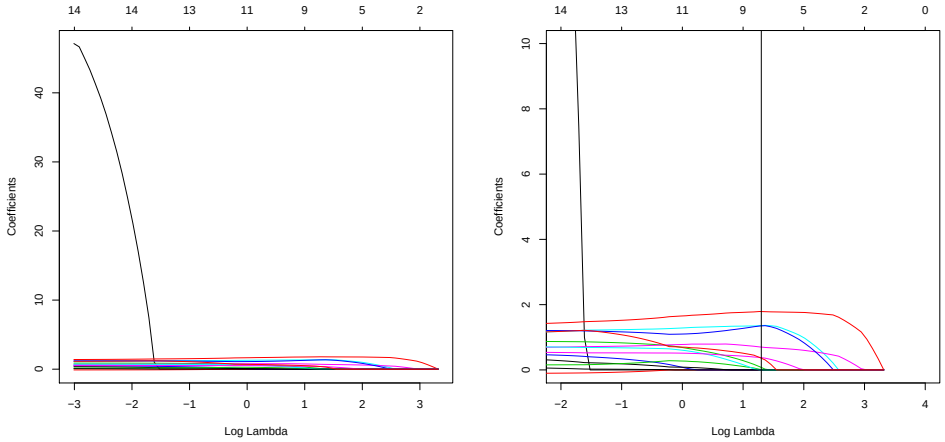


FIGURE 12.5: *Plots of coefficient values as a function of* $\log \lambda$ *for a regression of weight against all variables for the bodyfat dataset, penalised using the* L_1 *norm. In each case, the six outliers identified in Figure 57 were removed. On the* **left**, *the plot of the whole path for 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 12.4). 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).

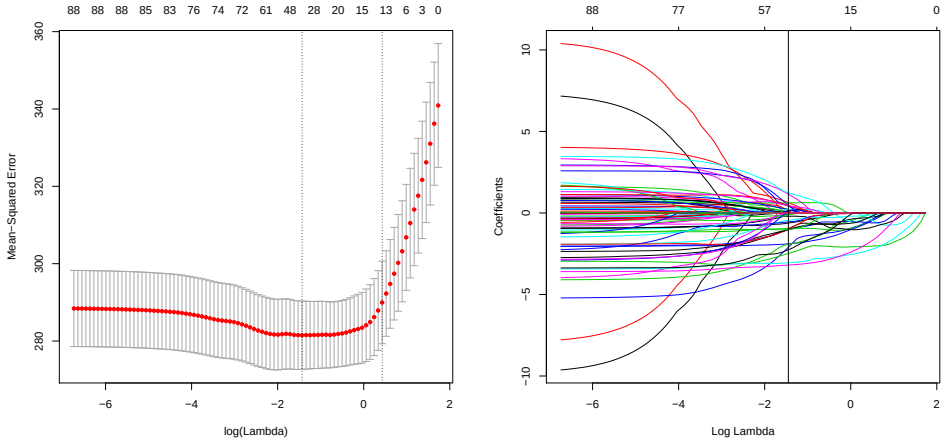


FIGURE 12.6: *Mean-squared error as a function of log regularization parameter (i.e.* $\log \lambda$*) for a regression of latitude against features describing music (details in text), using the dataset at* https://archive.ics.uci.edu/ml/datasets/Geographical+Original+of+Music *and penalized with the* 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 12.4 *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 12.7 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 12.2.

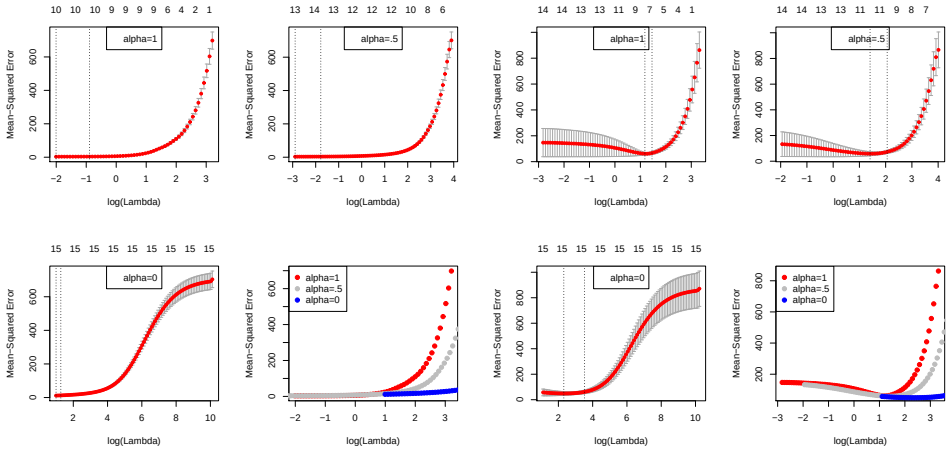


FIGURE 12.7: *Plots of mean-squared error as a function of log regularization parameter (i.e.* $\log \lambda$*) for a regression of weight against all variables for the bodyfat dataset using an elastic net regularizer for various choices of* α*. The case* $\alpha = 1$ *corresponds to a lasso;* $\alpha = 0$ *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 57 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.*

12.4.2 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 12.2: R code used for the elastic net regression example of worked example 12.4

```
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"))
```

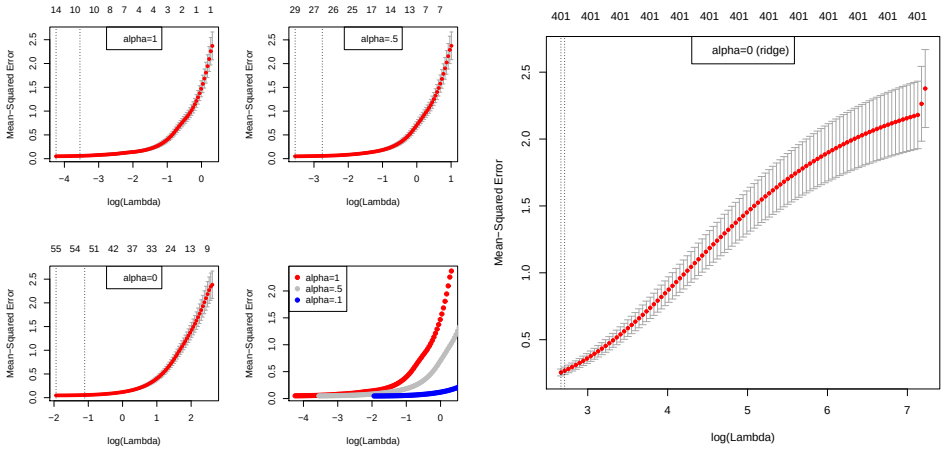


FIGURE 12.8: *On the* **left**, *a comparison between three values of* α *in a* `glmnet` *regression predicting octane from NIR spectra (see Example 12.5). The plots show cross-validated error against log regularization coefficient for* $\alpha = 1$ *(lasso) and two elastic-net cases,* $\alpha = 0.5$ *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.*

Worked example 12.5 *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 12.8 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).

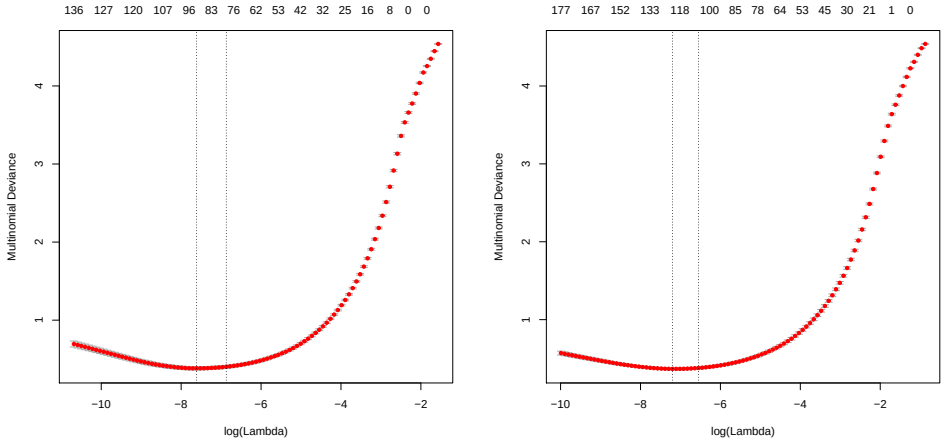


FIGURE 12.9: *Multiclass logistic regression on the MNIST data set, using a lasso and elastic net regularizers. On the* **left**, *deviance of held out data on the digit data set (worked example 12.6), for different values of the log regularization parameter in the lasso case. On the* **right**, *deviance of held out data on the digit data set (worked example 12.6), for different values of the log regularization parameter in the elastic net case,* $\alpha = 0.5$.

12.4.3 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.

Worked example 12.6 *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 12.9 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 12.3: R code used for the digit example of worked example 12.6

```
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))
```

12.5 YOU SHOULD

12.5.1 remember these definitions:

12.5.2 remember these terms:

12.5.3 remember these facts:

12.5.4 remember these procedures:

CHAPTER 13

Regression using Kernels and Neighbors

It is natural to predict y for some query point $\mathbf{x}$ by finding $\mathbf{x}$'s nearest neighbor and reporting the value of y at that point. The usual concerns with nearest neighbors apply (one must find the nearest neighbor, which isn't always easy; and one must use a sensible metric). There is another concern, which is the predicted y is now a piecewise constant function of $\mathbf{x}$. This means that $y(\mathbf{x})$ and $y(\mathbf{x} + \delta\mathbf{x})$ will be the same for almost every $\mathbf{x}$ as long as $\delta\mathbf{x}$ is small enough. Whether this is a problem or not depends on the application for the regression.

13.1 EXAMPLE: FILLING LARGE HOLES WITH NEARBY IMAGES

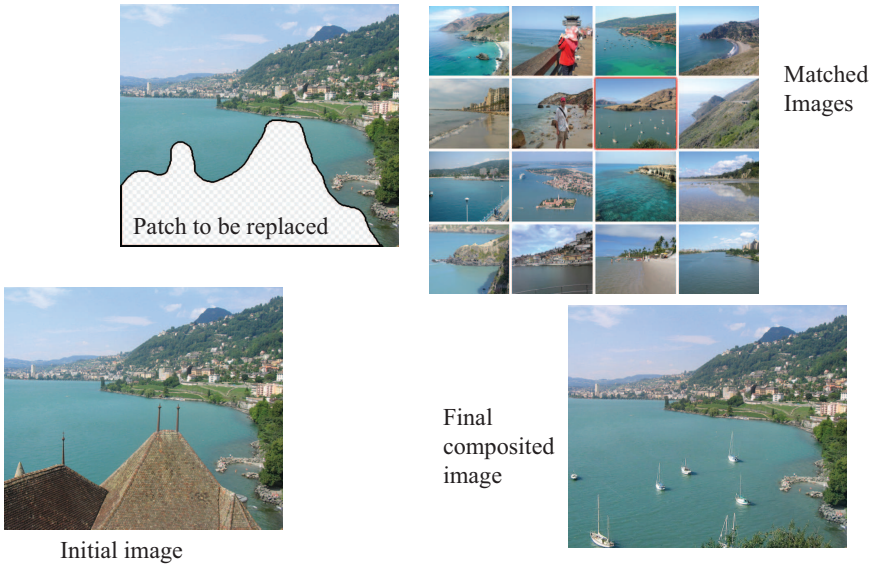


FIGURE 13.1: *We can fill large holes in images by matching the image to a collection, choosing one element of the collection, then cutting out an appropriate block of pixels and putting them into the hole in the query image. In this case, the hole has been made by an artist, who wishes to remove the roofline from the view. Notice how there are a range of objects (boats, water) that have been inserted into the hole. These objects are a reasonable choice, because the overall structures of the query and matched image are largely similar — getting an image that matches most of the query image supplies enough context to ensure that the rest "makes sense".*

A great strength of nearest neighbors is its flexibility, as the following example shows. Many different kinds of user want to remove things from images or from video. Art directors might like to remove unattractive telephone wires; restorers might want to remove scratches or marks; there's a long history of government officials removing people with embarrassing politics from publicity pictures (see the fascinating examples in **?**); and home users might wish to remove a relative they dislike from a family picture. All these users must then find something to put in place of the pixels that were removed. This is a regression problem — the $\mathbf{x}$ is the image with the hole in it, and the y is the material that fills the hole. But it's a regression problem that isn't like any other we've seen so far.

There are several difficulties. The hole in the image might be big (so entire objects might have been there, Figure 13.1), and different query images have holes of different sizes and shapes. This regression problem is easily (and well) handled by nearest neighbors. We match the image to a large collection of images, to find the nearest neighbors. Matching requires some care, because the image has a hole in it, and the images in the collection do not have holes. If the solution must be automatic, then we choose the nearest neighbor and fill in the pixels from that image. If not, we might offer a user several options, and the user gets to choose an image from among those options; the chosen image is used to fill in the hole.

It is straightforward to get a useful distance between images. We have an image with some missing pixels, and we wish to find nearby images. We will assume that all images are the same size. If this isn't in fact the case, we could either crop or resize the example images. A good measure of similarity between two images $\mathcal{A}$ and $\mathcal{B}$ can be measured by forming the **sum of squared differences** (or **SSD**) of corresponding pixel values. You should think of an image as an array of pixels. If the images are grey-level images, then each pixel contains a single number, encoding the grey-level. If the images are color images, then each pixel (usually!) contains three values, one encoding the red-level, one encoding the green-level, and one encoding the blue-level. The SSD is computed as

$$\sum_{(i,j)} (\mathcal{A}_{ij} - \mathcal{B}_{ij})^2$$

where i and j range over all pixels. If the images are grey-level images, then by $(\mathcal{A}_{ij} - \mathcal{B}_{ij})^2$, I mean the squared difference between grey levels; if they are color images, then this means the sum of squared differences between red, green and blue values. This distance is small when the images are similar, and large when they are different (it is essentially the length of the difference vector).

Now we don't know some of the pixels in the query image. Write $\mathcal{K}$ for the set of pixels around a point whose values are known, and $N(\mathcal{K})$ for the size of this set. We can now use

$$\frac{1}{N(\mathcal{K})} \sum_{(i,j)\in\mathcal{K}} (\mathcal{A}_{ij} - \mathcal{B}_{ij})^2$$

as the distance between images.

Filling in the pixels requires some care. One does not usually get the best results by just copying the missing pixels from the matched image into the hole. Instead, it is better to look for a good **seam**. We search for a curve enclosing the

missing pixels which (a) is reasonably close to the boundary of the missing pixels and (b) gives a good boundary between the two images. A good boundary is one where the query image (on one side) is "similar to" the matched image (on the other side). A good sense of similarity requires that pixels match well, and that image gradients crossing the boundary tend to match too. The details of this search can be found in the original paper, which is **********

Remember this: *Nearest neighbors is an extremely flexible regression strategy that can be used to make complex predictions. Examples include blocks of pixels, parse trees, and so on. Nearest neighbors regression can be very flexible about what distance is used, but the usual requirements apply. You need to be able to find the nearest neighbor for your query point, and the distance that you do use needs to reflect the requirements of the problem well.*

13.2 MORE ELABORATE USES OF NEIGHBORS

Figure **??** demonstrates how nearest neighbors can produce piecewise constant regressions. The natural fix is to obtain k nearest neighbors and form an estimate out of them. Averaging them isn't good enough, because the k nearest neighbors of $\mathbf{x}$ and $\mathbf{x} + \delta\mathbf{x}$ are the same for most $\mathbf{x}$ and sufficiently small $\delta\mathbf{x}$. There are a variety of methods to obtain a y that changes fairly smoothly with $\mathbf{x}$ using the k nearest neighbors.

13.2.1 Weighted Nearest Neighbors

The natural way to fix the nuisance of piecewise constant regressions is to form a weighted combination of the k nearest neighbors of the query point $\mathbf{x}$. If the weights change with $\mathbf{x}$, then the regression should not be piecewise constant. Assume that we have already collected the k nearest neighbors, which we write $\mathbf{x}_i$. Write y_i for the value of the dependent variable for the i'th of these points. Notice that some of these neighbors could be quite far from the query point. We don't want distant points to make as much contribution to the model as nearby points. This suggests forming a weighted average of the predictions of each point. The weight at each neighbor should depend on, at least, the location of the query point and the location of the neighbor. We encapsulate this into a weight function, where $w(\mathbf{x}, \mathbf{x}_i)$ represents the value of the weight function computed using $\mathbf{x}$ and the i'th point. Then the estimate is

$$y_{pred}(\mathbf{x}) = \frac{\sum_i y_i w(\mathbf{x}, \mathbf{x}_i)}{\sum_i w(\mathbf{x}, \mathbf{x}_i)}.$$

A variety of weighting functions are reasonable choices. We expect that weights are non-negative, and that weights go down for points that are further

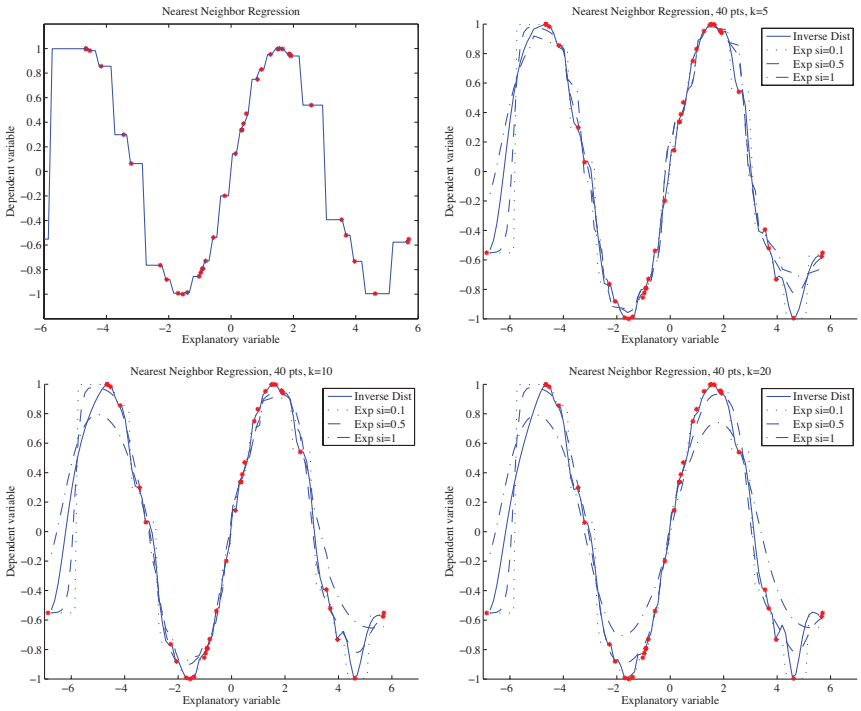


FIGURE 13.2: *Different forms of nearest neighbors regression, predicting y from a one-dimensional x, using a total of 40 training points.* **Top left:** *reporting the nearest neighbor leads to a piecewise constant function.* **Top right:** *improvements are available by forming a weighted average of the five nearest neighbors, using inverse distance weighting or exponential weighting with three different scales. Notice if the scale is small, then the regression looks a lot like nearest neighbors, and if it is too large, all the weights in the average are nearly the same (which leads to a piecewise constant structure in the regression).* **Bottom left** *and* **bottom right** *show that using more neighbors leads to a smoother regression.*

from $\mathbf{x}_i$. Write $d_i = \|(\mathbf{x} - \mathbf{x}_i)\|$ for the distance between the query point and the i'th nearest neighbor. Then **inverse distance weighting** uses $w(\mathbf{x}, \mathbf{x}_i) = 1/d_i$. Alternatively, we could use an exponential function to strongly weight down more distant points, using **gaussian weighting** with

$$w(\mathbf{x}, \mathbf{x}_i) = \exp\left(\frac{-d_i^2}{2\sigma^2}\right).$$

We will need to choose a scale σ, which can be done by cross-validation. This follows the recipe you should expect by now: hold out some examples, make predictions at the held out examples using a variety of different scales, and choose the scale that gives the best held-out error. If σ is small, then each of the weighted averages boil down to evaluating the dependent variable at the nearest neighbor (because all the other neighbors will have very small weight in the average). Large σ will result in

weights that are about the same at each nearest neighbor, and may result in an oversmooth regression.

When $\mathbf{x}$ has high dimension, it is usual that the nearest neighbor is a lot closer than the second nearest neighbor. When this happens, it may not be possible to choose a suitable scale, because most scales are too small for some points and too large for others. This is a manifestation of the curse of dimension.

Remember this: *Nearest neighbors can be used for regression. Predicting the dependent variable at the nearest neighbor is the simplest procedure, but leads to piecewise constant regressions. This can be cured by forming a weighted combination of the dependent variable at each of k nearest neighbors.*

13.2.2 Weighted Least Squares

In the previous section, we computed a value at $\mathbf{x}$ by forming a weighted average of its neighbors' predictions. An alternative is to fit a model to the neighbors. We then evaluate the model at $\mathbf{x}$ and report the result.

In the simplest case, assume the model is a constant and there is no weighting. So y does not depend on $\mathbf{x}$, and we fit a model by choosing the value of y to be the constant α that minimizes

$$\sum_i [y_i - \alpha]^2$$

where i runs over the k nearest neighbors. If you differentiate and set to zero, etc., you will find y is predicted by

$$y = \sum_{i=1}^{k} y_i \frac{1}{k}$$

(i.e. average the k nearest neighbors).

But more distant neighbors should have less effect on the chosen model, which justifies weighting down their contribution to the model's error. The weight we use should depend on (a) the query point and (b) the k nearest neighbors, meaning we need a weight function. Again, write $w(\mathbf{x}, \mathbf{x}_i)$ for the value of the weight function computed using $\mathbf{x}$ and the i'th point. Again, assume that the model is a constant. So we fit a model by choosing the value of y to be the constant α that minimizes

$$\sum_i [y_i - \alpha]^2 w(\mathbf{x}, \mathbf{x}_i).$$

If you differentiate and set to zero, etc., you will find y is predicted by:

$$y = \sum_{i=1}^{N} y_i \left(\frac{w(\mathbf{x}, \mathbf{x}_i)}{\sum_{j=1}^{k} w(\mathbf{x}, \mathbf{x}_j)} \right)$$

which is the expression we had for weighted nearest neighbors. Notice that this results in a y value that *does* depend on $\mathbf{x}$, even though this is usually thought of as a constant model. Though we have a constant model, the constant we choose changes from point to point because the weights change from point to point.

There is a natural way to generalize this procedure: I could use a more complex regression model. Linear models are the most important case. Recall that I simplified notation for linear models by assuming that each vector of independent variables $\mathbf{x}$ had a component that was 1. I will assume that this applies here, too. Notice that this doesn't affect distances or nearest neighbors. In this case, a linear function is $\mathbf{x}^T\beta$, where β is a vector of coefficients. We will choose these coefficients using $\mathbf{x}$'s k nearest neighbors, by minimizing a weighted error function.

In particular, we choose

$$\hat{\beta} = \operatorname*{argmin}_{\beta} \sum_i \left[y_i - \mathbf{x}_i^T\beta\right]^2 w(\mathbf{x}, \mathbf{x}_i)$$

where w is a weighting function, as above, and where i runs over $\mathbf{x}$'s k nearest neighbors. Notice that you can obtain $\hat{\beta}$ by solving a system of linear equations. Write $\mathbf{Y}$ for a vector of y-values, as above, and $\mathcal{X}$ for a matrix of data constructed from the nearest neighbors. Write $\mathcal{W}$ for a diagonal matrix, with $w(\mathbf{x}, \mathbf{x}_i)$ at the i'th diagonal element. Then

$$\mathcal{X}^T\mathcal{W}\mathcal{X}\hat{\beta} = \mathcal{X}^T\mathcal{W}\mathbf{Y}.$$

If k is larger than the dimension of $\mathbf{x}$, we could solve this directly, though it might be a better idea to regularize. If k is smaller than $\mathbf{x}$, we must regularize and so solve

$$(\mathcal{X}^T\mathcal{W}\mathcal{X} + \lambda\mathcal{I})\hat{\beta} = \mathcal{X}^T\mathcal{W}\mathbf{Y}$$

for some λ chosen by cross-validation. Once we have $\hat{\beta}$ we can predict a value for $\mathbf{x}$ as $\mathbf{x}^T\hat{\beta}$. Notice that this *isn't* a linear function of $\mathbf{x}$, because $\hat{\beta}$ depends on $\mathbf{x}$ in a relatively complicated way (the weighting function depends on $\mathbf{x}$).

This procedure can produce quite good regressions in moderate dimension, but the dimension of the data can be a problem. In high dimensions, you need many nearest neighbors. Generally, in high dimensions, you will find that one neighbor is much closer than the others, so that to get weights that smooth well you might need a very large σ, which means the model is about the same as a linear model. The other difficulty with the method is that each prediction is quite expensive (you must find the nearest neighbors, and solve a linear system).

Remember this: *Weighted least squares fits a model to the k nearest neighbors of each query point, using some appropriate set of weights. The most important cases use a constant model (which turns out to be weighted nearest neighbors regression) or a linear model. When the dimension is high, these methods are often badly behaved because one of the k nearest neighbors tends to be a lot closer than all others.*

13.3 KERNELS AND FEATURES

We will now generalize weighted nearest neighbors again, but in a different direction. Weighted nearest neighbors works when we have a weight function that represents how much the value at nearby points should affect the value at $\mathbf{x}$. We assumed that $\mathbf{x}$ is "reasonably" scaled, meaning that distances between points are a good guide to the similarity of their y values. We assumed the similarity between $y(\mathbf{u})$ and $y(\mathbf{v})$ to depend on the distance between these points (i.e. $\|\mathbf{u} - \mathbf{v}\|$) rather than on the direction of the vector joining them (i.e. $\mathbf{u} - \mathbf{v}$). Furthermore, we assumed that the dependency should decline with increasing $\|\mathbf{x} - \mathbf{x}_i\|$.

13.3.1 Kernel Functions

We will formalize the idea of a bump function with a **kernel function**. A kernel function $K(u)$ is a non-negative function such that (a) $K(-u) = K(u)$ and (b) $\int_{-\infty}^{\infty} K(u)du = 1$. Widely used kernel functions are:

- **The Gaussian kernel**, $K(u) = \frac{1}{\sqrt{2\pi}} \exp -\frac{u^2}{2}$. Notice this doesn't have compact support.
- **The Epanechnikov kernel**, $K(u) = \frac{3}{4}(1 - u^2)\mathbb{I}_{[|u|\leq 1]}$. This isn't differentiable at $u = -1$ and at $u = 1$.
- **The Logistic kernel**, $K(u) = \frac{1}{e^{-u}+e^{u}+2}$. This doesn't have compact support, either.
- **The Quartic kernel**, $K(u) = \frac{15}{16}(1 - u^2)^2\mathbb{I}_{[|u|\leq 1]}$.

You should notice that each is a bump function – it's large at $u = 0$, and falls away as $|u|$ increases. It follows from the two properties above that, for $h > 0$, if $K(u)$ is a kernel function, then $K(u; h) = \frac{1}{h}K(\frac{u}{h})$ is also a kernel function. This means we can vary the width of the bump at the origin in a natural way by choice of h; this is usually known as the "scale" of the function. This is useful, because in most problems, we don't know how quickly the weights should decline with increasing distance. The scaling parameter will need to be selected, and cross-validation is a natural strategy.

We can easily use a kernel to make a bump in d dimensions, by looking at $K(\|\mathbf{x} - \mathbf{x}_c\|)$. This will have its bump centered on $\mathbf{x}_c$, and will fall off with distance from $\mathbf{x}_c$. Conveniently, the bump falls off in the same way along each of the directions leaving $\mathbf{x}_c$. We now have a quite general modelling strategy: place a collection of bumps at a variety of points, and build a model of y by adjusting the weights (and perhaps the scales) of the bumps.

13.3.2 Smoothing with Kernels

An important issue with weighted nearest neighbors is that some query points $\mathbf{x}$ may have many nearby neighbors, and others may have nearest neighbors that lie far away from the point. The weighting process we used there does not take this into account. Weights depend on distance (and perhaps a single global scaling in the case of gaussian weights). Ideally, if a point has very close neighbors, the

weighting function will change quite quickly with distance; and if a point has only distant neighbors, the weighting function will change slowly with distance.

This is quite easily achieved with kernel functions. We use kernels to construct the weight functions, but we use a different scale at each kernel, yielding

$$y_{pred}(x) = \sum_{i=1}^{N} y_i \left(\frac{K\left(\frac{\|\mathbf{x} - \mathbf{x}_i\|}{h_i}\right)}{\sum_{j=1}^{k} K\left(\frac{\|\mathbf{x} - \mathbf{x}_j\|}{h_j}\right)} \right).$$

which you should compare with the original weighted least squares. Notice the weights are non-negative, and sum to one at each point. Changing the h_i will change the radius of the bumps, and will cause more (or fewer) points to have more (or less) influence on the shape of $y(\mathbf{x})$. Selecting the h_i is easy in principle. We search for a set of values that minimizes cross-validation error. In practice, this takes an extremely extensive search involving a great deal of computation, particularly if there are lots of points or the dimension is high. For the examples of this section, I used the R package `np`.

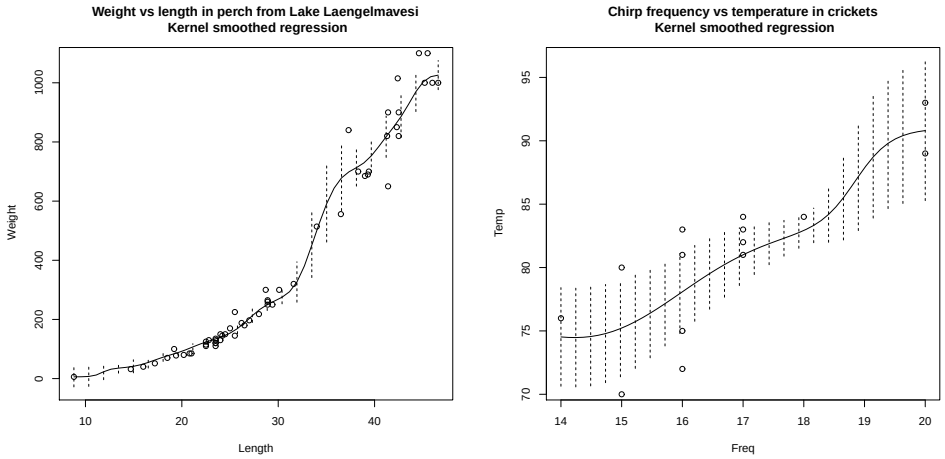


FIGURE 13.3: *A non-parametric regression using a kernel smoothing method for the datasets of Figures 11.1 and 11.5. The curve shows the expected value of the independent variable for each value of the explanatory variable. The vertical bars show the standard error of the predicted density for the independent variable, for each value of the explanatory variable. Notice that, as one would expect, the standard deviation is smaller closer to data points, and larger further away. On the* **left**, *the perch data. On the* **right**, *the cricket data.*

A nice feature of this search is that it generates a great deal of information about the likely accuracy of the model. Each data point is going to be held out multiple times, so that we can estimate the standard error of the prediction at each data point. This is the standard deviation of the error in the estimate of the mean,

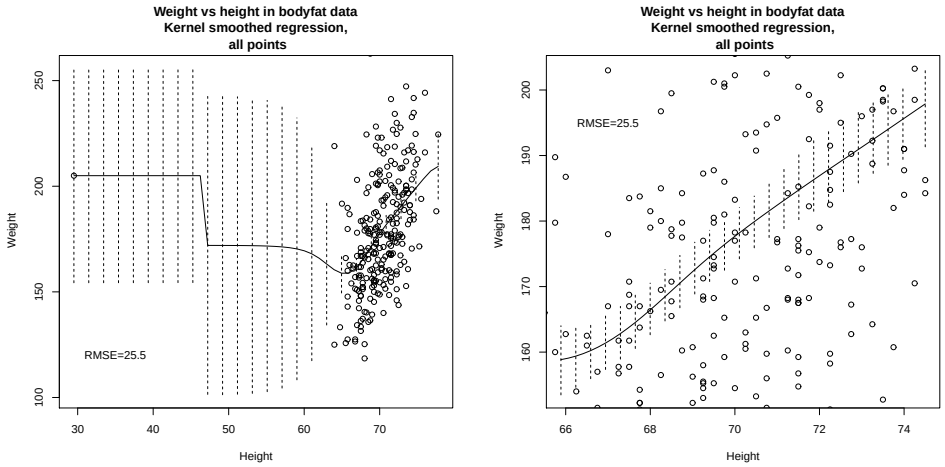


FIGURE 13.4: *A non-parametric regression, using kernel smoothing, of weight against height, using the dataset of Figure 57. On the* **left**, *a curve showing the expected value of the independent variable for each value of the explanatory variable, with axes chosen to show the outlying point. The vertical bars show the standard error of the predicted density for the independent variable, for each value of the explanatory variable. Notice the method can make predictions around this point, but that these predictions have very large standard error as one would expect. On the* **right**, *the same curve, but on axes chosen to display the non-outlying points. The bars are small, because the standard error is small here (this isn't the same as the standard deviation of the predictive distribution).*

due to random sampling effects. The details of this estimate are well beyond the scope of these notes. But `np` produces it, and I have shown standard error bars on the plots. You should notice that this is *not* the standard deviation of the predictive distribution $P(y|\mathbf{x})$ (which we haven't computed, and largely haven't discussed). Instead, the bars are the uncertainty in the prediction produced at a point due to the changes from training set to training set. Large bars indicate predictions you should not trust, because resampling the training data set produces large variations in the predictions at that point.

13.3.3 Kernel Regression

Instead of smoothing nearest neighbors with kernel functions, we can use kernel functions to generate features. In particular, choose a kernel function K, and a scale h. Choose a set of R points $\mathbf{b}_j$. These don't have to be training points, but usually are. They are sometimes referred to as **base points**. Then we can write a

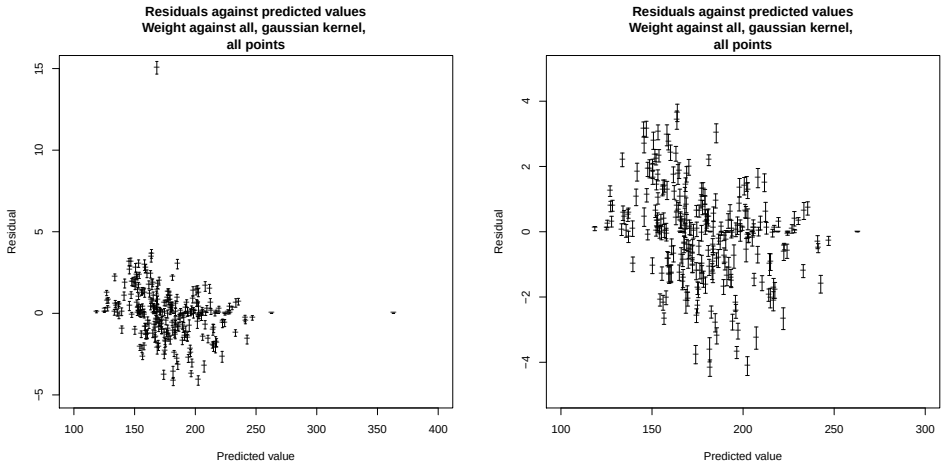


FIGURE 13.5: *A non-parametric regression, using kernel smoothing, of weight against all variables, using the bodyfat dataset of Figure 57. On the* **left**, *a scatterplot of the predicted value against residual, on axes that show all points. The residual has* not *been standardized, because it is unclear how to do so. I have shown a one standard error bar for the residual as a vertical bar, plotted at the mean predicted value. For some points, the standard error is very small, so you can only see a horizontal bar. Notice how some points have a large residual, but most residuals are in a reasonable range. On the* **right**, *the same plot, but on axes chosen to display the non-outlying points. The bars are small, because the standard error is small here (this isn't the same as the standard deviation of the predictive distribution).*

feature vector representing the point $\mathbf{x}$ by

$$\begin{pmatrix} K\left(\frac{\|\mathbf{x}-\mathbf{b}_1\|}{h}\right) \\ K\left(\frac{\|\mathbf{x}-\mathbf{b}_2\|}{h}\right) \\ \cdots \\ K\left(\frac{\|\mathbf{x}-\mathbf{b}_R\|}{h}\right) \end{pmatrix}$$

and we can construct a linear regression using that feature vector. Equivalently, we seek a set of coefficients β_j so that

$$y(\mathbf{x}; K, \beta, h, \mathbf{b}_j) = \sum_{j=1}^{R} \beta_j K\left(\frac{\|\mathbf{x}-\mathbf{b}_j\|}{h}\right)$$

represents the training data well. You can think of this process as placing a weighted bump at each base point. The weights and scale are then chosen by minimizing regression error. This kind of model is particularly valuable for modelling surfaces on the plane.

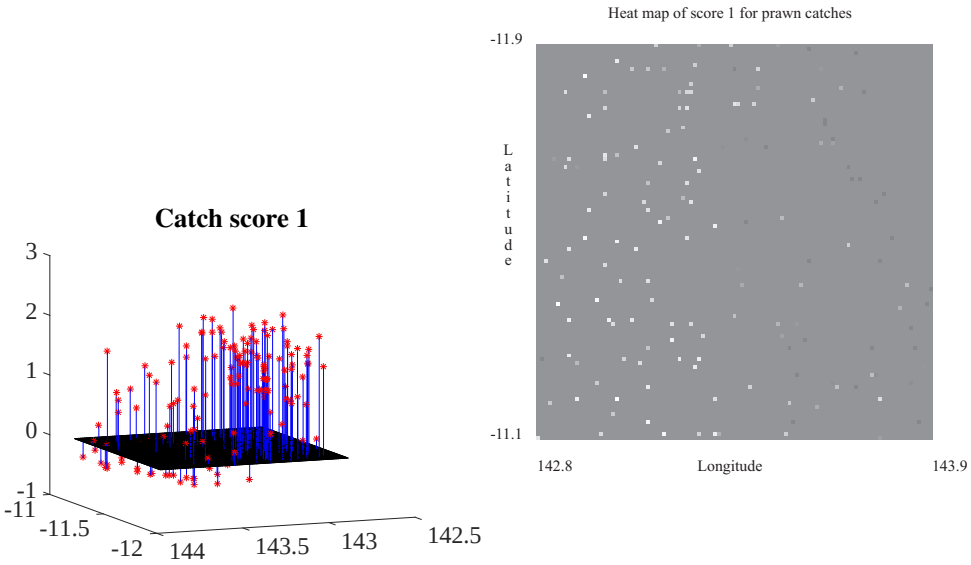


FIGURE 13.6: *A dataset recording scores of prawn trawls around the Great Barrier Reef, from http://www.statsci.org/data/oz/reef.html. There are two scores; this is score 1. On the* **left** *I have plotted the data as a 3D scatter plot. This form of plot isn't usually very successful, though it helps to make it slightly easier to read if one supplies vertical lines from each value to zero, and a zero surface. On the* **right***, a heat map of this data, made by constructing a fine grid, then computing the average score for each grid box (relatively few boxes get one score, and even fewer get two). The brightest point corresponds to the highest score; mid-grey is zero (most values), and dark points are negative. Notice that the scale is symmetric; the reason there is no very dark point is that the smallest value is considerably larger than the negative of the largest value. The x and y dimensions are longitude and latitude, and I have ignored the curvature of the earth, which is pretty small at this scale.*

13.3.4 Example: Smoothing and Interpolation on the Plane

Imagine we have a set of points $\mathbf{x}_i$ on the plane, with a measured height value y_i for each point. We would like to reconstruct a surface from this data, in the form of a height map. There are two important subcases: **interpolation**, where we want a surface that passes through each value; and **smoothing**, where our surface should be close to the values, but need not pass through them. This case is easily generalised to a larger number of dimensions. Particularly common is to have points in 3D, or in space and time.

This is a form of regression problem, but one which has generated an important literature of its own because it has applications that aren't statistical. For example, you might have a set of points that lie on (say) a car's bonnet, and you want to make a smooth surface passing near those points. Being able to do this is important in computer aided design and computer graphics. Interpolation and

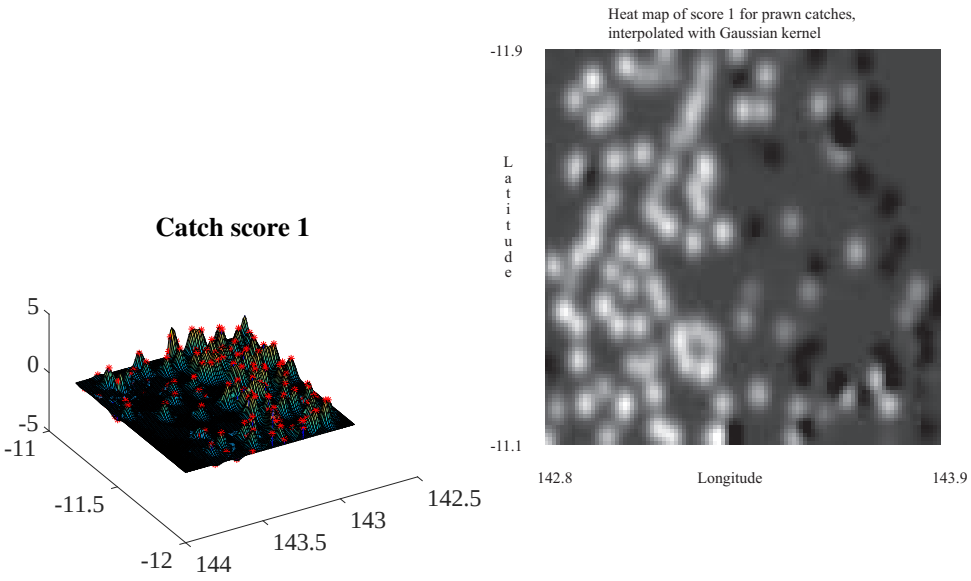


FIGURE 13.7: *The prawn data of figure 13.6, interpolated with radial basis functions (in this case, a Gaussian kernel) with scale chosen by cross-validation. On the* **left**, *a surface shown with the 3D scatter plot. On the* **right**, *a heat map. I ignored the curvature of the earth, small at this scale, when computing distances between points. This figure is a good example of why interpolation is usually not what one wants to do (text).*

smoothing problems in one dimension have the remarkable mathematical property of being very different to those in two and more dimensions (if you've been through a graphics course, you'll know that, for example, interpolating splines in 2D are very different from those in 1D). We will concentrate on the multidimensional case.

Our regression will be

$$y(\mathbf{x}; K, \beta, h, \mathbf{b}_j) = \sum_{j=1}^{R} \beta_j K\left(\frac{\|\mathbf{x} - \mathbf{b}_j\|}{h}\right)$$

Consider the values that this function takes at the training points $\mathbf{x}_i$, and suppress some notation. We have

$$y(\mathbf{x}_i) = \sum_{j=1}^{R} \beta_j K\left(\frac{\|\mathbf{x}_i - \mathbf{b}_j\|}{h}\right)$$

and we would like to minimize $\sum_i (y_i - y(\mathbf{x}_i))^2$. We can rewrite this with the aid of some linear algebra. Write $\mathcal{G}$ for the **Gram matrix**, whose i, j'th entry is $K\left(\frac{\|\mathbf{x}_i - \mathbf{b}_j\|}{h}\right)$; write $\mathbf{Y}$ for the vector whose i'th component is y_i; and $\mathbf{a}$ for the vector whose j'th component is a_j. Then we want to minimize

$$(\mathbf{Y} - \mathcal{G}\mathbf{a})^T(\mathbf{Y} - \mathcal{G}\mathbf{a}).$$

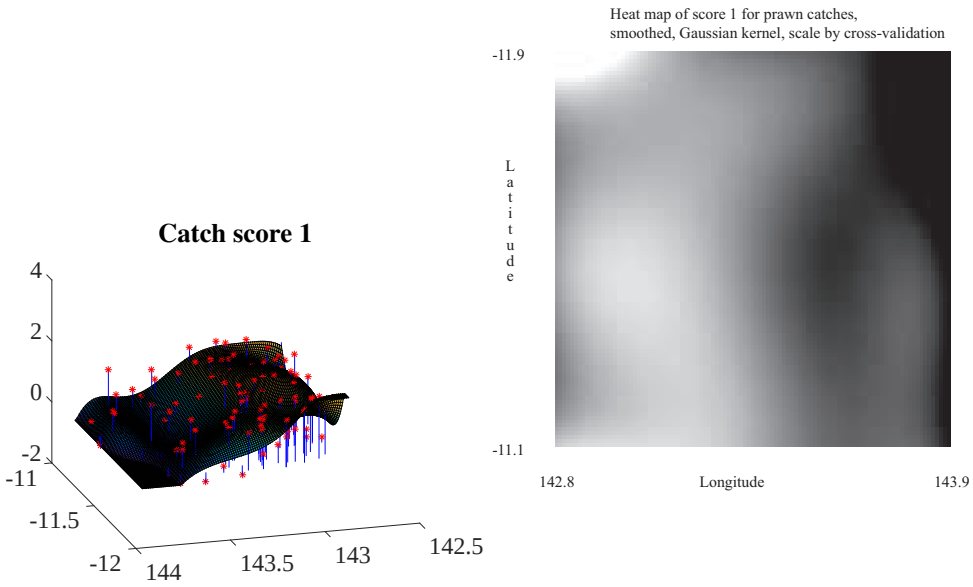


FIGURE 13.8: *The prawn data of figure 13.6, smoothed with radial basis functions (in this case, a Gaussian kernel) with scale chosen by cross-validation. On the* **left**, *a surface shown with the 3D scatter plot. On the* **right**, *a heat map. I ignored the curvature of the earth, small at this scale, when computing distances between points. I used 60 basepoints, constructed by choosing 60 of 155 training points at random, then adding a small random offset.*

There are a variety of cases. For interpolation, we choose the base points to be the same as the training points. A theorem of Micchelli guarantees that for a kernel function that is (a) a function of distance and (b) monotonically decreasing with distance, $\mathcal{G}$ will have full rank. Then we must solve

$$\mathbf{Y} = \mathcal{G}\mathbf{a} \qquad \text{(Interpolation)}$$

for $\mathbf{a}$ and we will obtain a function that passes through each training point. For smoothing, we may choose any set of $R < N$ basepoints, though it's a good idea to choose points that are close to the training points. Cluster centers are one useful choice. We then solve the least-squares problem by solving

$$\mathcal{G}^T\mathbf{Y} = \mathcal{G}^T\mathcal{G}\mathbf{a} \qquad \text{(Smoothing)}$$

for $\mathbf{a}$. In either case, we choose the scale h by cross-validation. We do this by: selecting a set of scales; holding out some training points, and interpolating (resp. smoothing) the values of the others; then computing the error of the predictions for these held-out points. The error is usually the square of the residual, and it's usually a good idea to average over many points when doing this.

Both interpolation and smoothing can present significant numerical challenges. If the dataset is large, $\mathcal{G}$ will be large. For values of h that are large, $\mathcal{G}$

(resp. $\mathcal{G}^T\mathcal{G}$) is usually very poorly conditioned, so solving the interpolation (resp. smoothing) system accurately is hard. This problem can usually be alleviated by adding a small constant (λ) times an identity matrix ($\mathcal{I}$) in the appropriate spot. So for interpolation, we solve

$$\mathbf{Y} = (\mathcal{G} + \lambda\mathcal{I})\mathbf{a} \qquad \text{(Interpolation)}$$

for $\mathbf{a}$ and we will obtain a function that passes through each training point. Similarly, for smoothing, we solve

$$\mathcal{G}^T\mathbf{Y} = (\mathcal{G}^T\mathcal{G} + \lambda\mathcal{I})\mathbf{a} \qquad \text{(Smoothing)}$$

for $\mathbf{a}$. Usually, we choose λ to be small enough to make the linear algebra work ($1e-9$ usually works for me), and ignore it.

As Figure 13.7 suggests, interpolation isn't really as useful as you might think. Most measurements aren't exactly right, or exactly repeatable. If you look closely at the figure, you'll see one location where there are two scores; this is entirely to be expected for the score of a prawn trawl at a particular spot in the ocean. This creates problems for interpolation; $\mathcal{G}$ must be rank deficient, because two rows will be the same. Another difficulty is that the scores look "wiggly" — moving a short distance can cause the score to change quite markedly. This is likely the effect of luck in trawling, rather than any real effect. The interpolating method chooses a very short scale, because this causes the least error in cross-validation, caused by predicting zero at the held out point (which is more accurate than any prediction available at any longer scale). The result is an entirely implausible model.

Remember this: *Interpolation isn't as useful as most people think, because most measurements aren't exactly right or exactly repeatable.*

Now look at Figure 13.8. The smoothed surface is a reasonable guide to the scores; in the section of ocean where scores tend to be large and positive, so is the smoothed surface; where they tend to be negative, the smoothed surface is negative, too.

13.3.5 Model Selection

We have constructed a model of y as a weighted sum of bump functions. We placed one bump function at each of a set of base points, then solved a linear system to get the weights that best represent y. You could see this as a version of linear regression, where we constructed new independent variables (the bump functions) from the original independent variables (the $\mathbf{x}$). Apart from the bump functions, we were following the general scheme of linear regression. This is important. It means that we can apply the material of chapter 57 to this idea. For example, you could choose the number and location of bumps using AIC or BIC and forward or backward stagewise regression. You could choose the number and location of

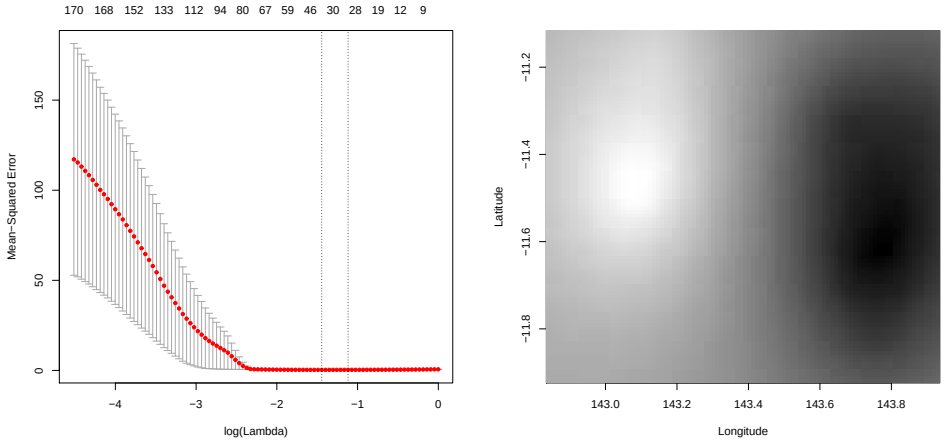


FIGURE 13.9: *The prawn data of figure 13.6, now smoothed with* `glmnet`*. On the* **left***, the plot of held out error vs log regularization weight produced by* `glmnet` *for the prawn data, as described in example 13.1. Notice that the best regression uses between 20 and 30 of 930 possible features. On the* **right***, a heat map. case, a Gaussian kernel) with scale chosen by cross-validation.*

bumps using a lasso penalty. You could change the squared error cost function to a robust cost function. All of these would be relatively routine (and really useful) combinations of methods we already know.

Worked example 13.1 *Using the lasso to predict prawn catches*

One can combine methods, very usefully. It is quite straightforward to persuade `glmnet` to apply the lasso to the prawn catch data.

Solution: I chose a set of base points (in this case, all training data points) and a set of six scales. I then built a bump of each scale at each base point. Write $\mathbf{x}_i$ for each training data point, y_i for the value at the i'th training point, $\mathbf{b}_j$ for the j'th base point, s_k for the k'th scale and $K(\mathbf{x}; \mathbf{b}_j, s_k)$ for the bump of scale s_k centered at $\mathbf{b}_j$, as a function of position $\mathbf{x}$. I then constructed a vector of training features for the i'th training point consisting of

$$\mathbf{f}_i = [K(\mathbf{x}_i; \mathbf{b}_1, s_1), \ldots, K(\mathbf{x}_i; \mathbf{b}_1, s_k), \ldots, K(\mathbf{x}_i; \mathbf{b}_j, s_1), \ldots, K(\mathbf{x}_i; \mathbf{b}_j, s_k), \ldots] .$$

In this dataset, there are 155 points, and I used 6 scales, so this is a 930 dimensional vector. I then used `glmnet` to predict the values of y_i from $\mathbf{f}_i$. I used an elastic net regression (i.e. a convex combination of lasso and ridge penalties) because I expected strong correlations between the components of $\mathbf{f}_i$; I used $\alpha = 0.5$. Figure 13.9 shows the result, which to me seems a better model than that of Figure 13.8 (there is slightly less wiggling). Notice how good `glmnet` is at reducing the number of variables. The best regression uses between 20 and 30 of 930 possible features (depending on precisely what criterion one uses to choose). This is likely because the method gets to select a scale for each kernel function, and can use more than one kernel at a base point.

13.3.6 Parametric and Non-parametric models

All of our regression models to date have been built by choosing some family of parametric functions of the independent variables, then choosing the parameters that give the best prediction. Write θ for a vector of parameters, $\mathbf{x}$ for the independent variables, and $f(\mathbf{x}; \theta)$ for the function in the family chosen by θ; then we obtain a "good choice" of parameters (write $\hat{\theta}$ for that choice), and our prediction is

$$f(\mathbf{x}; \hat{\theta}).$$

As a concrete example, all the regressions we have discussed so far have used the family of linear functions.

There are tremendous advantages to this approach. By choosing a family of functions with a "small" set of parameters, we can control the variance of the estimate. We can make predictions for points that are some way away from the training data, and they are often quite good. We have a collection of strong algorithms for choosing $\hat{\theta}$, and we can choose algorithms that impose properties on the estimate, too (eg sparseness; robustness to outliers). A model of this form is usually called a **parametric model**.

There are some disadvantages. By forcing the prediction to come from a particular family of functions, we are also forced to make inaccurate predictions for

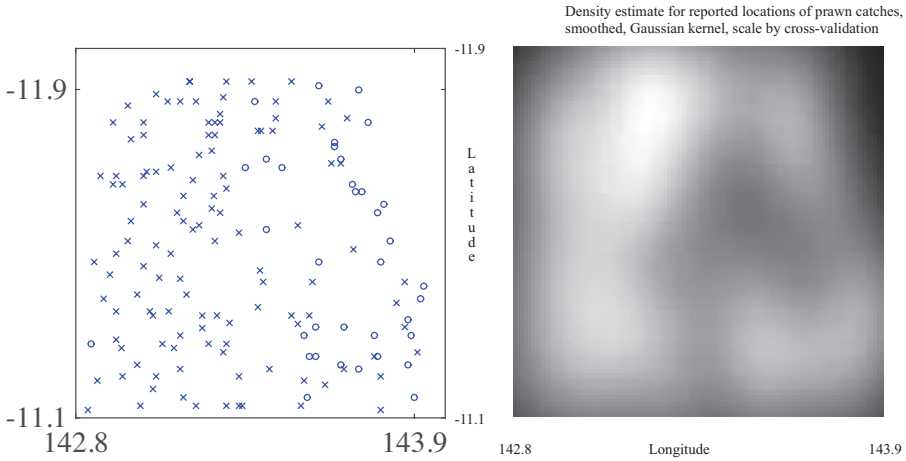


FIGURE 13.10: *The prawn data of figure 13.6, now shown on the* **left** *as a scatter plot of locations from which scores were reported. The 'o's correspond to negative scores, and the '+'s to positive scores. On the* **right***, a density estimate of the probability distribution from which the fishing locations were drawn, where lighter pixels correspond to larger density values. I ignored the curvature of the earth, small at this scale, when computing distances between points.*

some datasets (those where that family doesn't encode the prediction well). It is hard to build a family of functions that is very flexible in regions where there are lots of data points, and quite inflexible where there are few (you could look at my lasso example on the prawn data as a step in that direction).

An alternative approach is to use a **non-parametric model**. The precise delineation between the classes is difficult to state, but a non-parametric model does not estimate parameter values. Instead, the model finds useful training examples and combines them to make predictions.

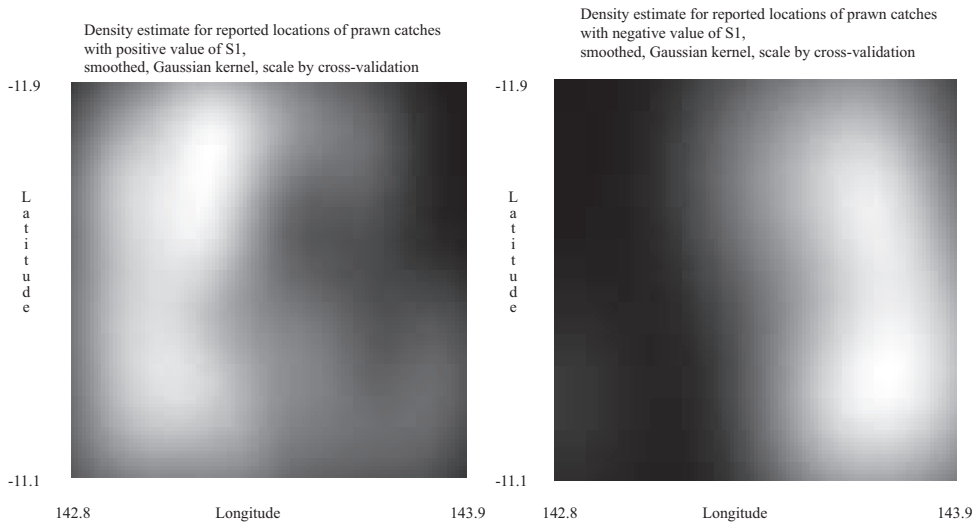


FIGURE 13.11: *Further plots of the prawn data of figure 13.6. On the* **left**, *a density estimate of the probability distribution from which the fishing locations* which achieved positive values of the first score *were drawn, where lighter pixels correspond to larger density values. On the* **right**, *a density estimate of the probability distribution from which the fishing locations* which achieved negative values of the first score *were drawn, where lighter pixels correspond to larger density values. I ignored the curvature of the earth, small at this scale, when computing distances between points.*

13.4 YOU SHOULD

13.4.1 remember these definitions:

13.4.2 remember these terms:

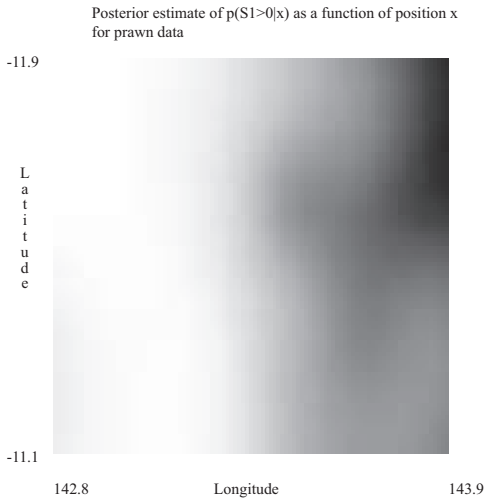


FIGURE 13.12: *An estimate of the posterior probability of obtaining a positive value of the first catch score for prawns, as a function of position, from the prawn data of figure 13.7, using the density estimate of figure* **??**. *Lighter pixels correspond to larger density values. I ignored the curvature of the earth, small at this scale, when computing distances between points.*

13.4.3 remember these facts:

13.4.4 remember these procedures:

13.4.5 be able to:

-

PART FIVE

GRAPHICAL MODELS

CHAPTER 14

Markov Chains

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.

14.1 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.

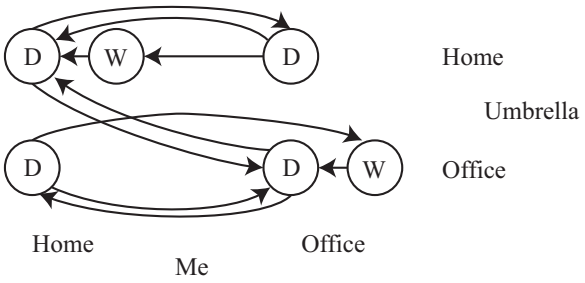


FIGURE 14.1: *A directed graph representing the umbrella example. Notice you can't arrive at the office wet with the umbrella at home (you'd have taken it), and so on. Labelling the edges with probabilities is left to the reader.*

Worked example 14.1 *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 14.1 gives this chain. A more interesting question is with what probability I arrive at my destination wet? Again, we will solve this with simulation.

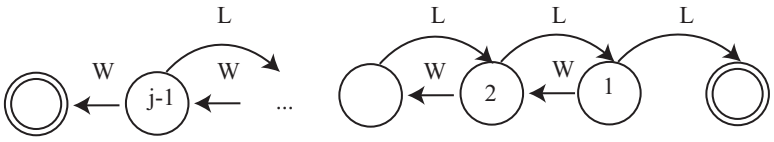


FIGURE 14.2: *A directed graph representing the gambler's ruin example. I have labelled each state with the amount of money the gambler has at that state. There are two end states, where the gambler has zero (is ruined), or has j and decides to leave the table. The problem we discuss is to compute the 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 14.2 *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 14.2 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 14.1 and 14.2. 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 14.3).

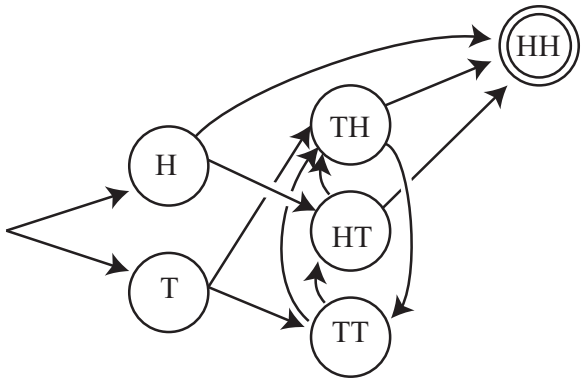


FIGURE 14.3: *A directed graph representing the coin flip example, using the pairs of random variables described in worked example 14.3. A sequence "HTHTHH" (where the last two H's are the last two flips) would be generated by transitioning to H, then to HT, then to TH, then to HT, then to TH, then to HH. By convention, the end state is a double circle. Each edge has probability 1/2.*

Worked example 14.3 *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 14.3 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.

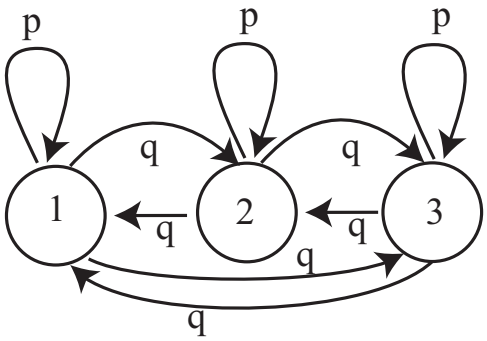


FIGURE 14.4: *A virus can exist in one of 3 strains. At the end of each year, the virus mutates. With probability α, it chooses uniformly and at random from one of the 2 other strains, and turns into that; with probability $1-\alpha$, it stays in the strain it is in. For this figure, we have transition probabilities $p = (1-\alpha)$ and $q = (\alpha/2)$.*

Useful Facts: 14.1 *Markov chains*

A Markov chain is a sequence of random variables X_n with the property that,

$$P(X_n = j|\text{values of all previous states}) = P(X_n = j|X_{n-1}).$$

14.1.1 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 14.4 *Viruses*

Write out the transition probability matrix for the virus of Figure 14.4, 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: 14.2 *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 $\pi_{n-1}^T \mathcal{P}$.

14.1.2 Stationary Distributions

Worked example 14.5 *Viruses*

We know that the virus of Figure 14.4 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 14.5, 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 14.6). 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 14.6 *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 14.1. If I have not, I just get wet. Illustrate this Markov chain with a state diagram.

Solution: Figure 14.5 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: 14.3 *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.

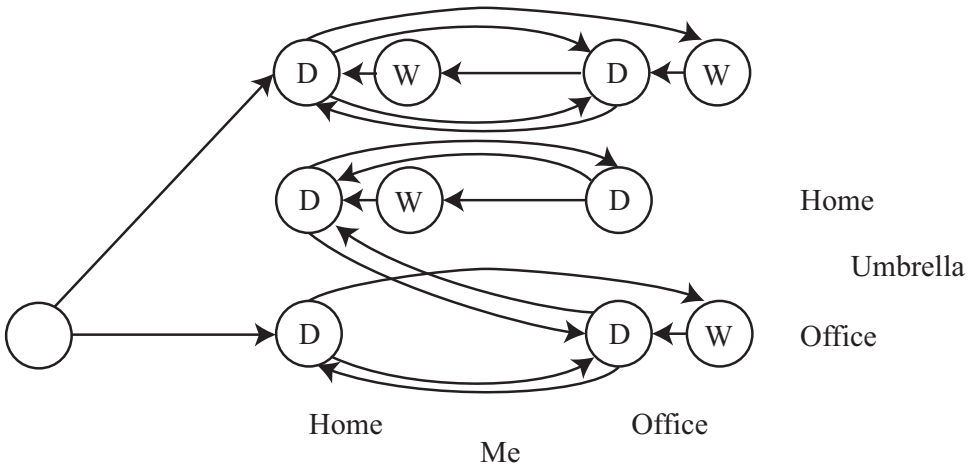


FIGURE 14.5: *In this umbrella example, there can't be a stationary distribution; what happens depends on the initial, random choice of buying/not buying an umbrella.*

14.1.3 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 14.7 *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 14.7). 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 14.8 *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.

14.2 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.

14.2.1 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 14.9 *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 14.10 *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 14.11 *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.

14.2.2 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 14.6 shows how the result of a simulation behaves when the number of runs changes. I used the simulation of example 14.11, 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).

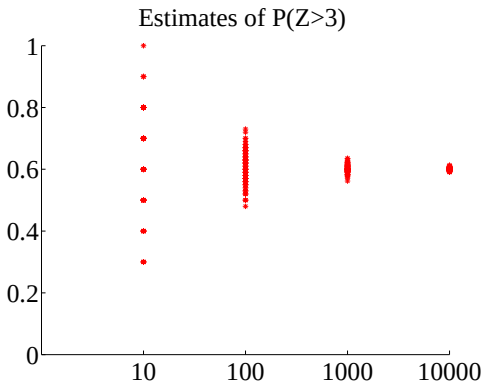


FIGURE 14.6: *Estimates of the probability from example 14.11, obtained from different runs of my simulator using different numbers of samples. In each case, I used 100 runs; the number of samples is shown on the horizontal axis. You should notice that the estimate varies pretty widely when there are 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 14.12 *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 14.11, 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: 14.4 *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 14.13 *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.

14.2.3 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 14.14 *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 57 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 14.15 *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 14.16 *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 14.16, 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.

14.3 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.

14.4 YOU SHOULD

14.4.1 remember these definitions:

14.4.2 remember these terms:

14.4.3 remember these facts:

14.4.4 be able to:

- Estimate various probabilities and expectations for a Markov chain by simulation.
- Evaluate the results of multiple runs of a simple simulation.

PROBLEMS

14.1. 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 $P(N) = (5/6)P(N-1) + (25/36)P(N-2)$.

14.2. 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 $P(N) = (1/2)P(N-2)+(1/4)P(N-3)$.

14.3. For the umbrella example of worked example 14.1, 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

14.4. 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?

14.5. 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 15

Hidden Markov Models

15.1 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.

15.1.1 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 14.1.3. 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.

15.1.2 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 minimises

$$-\log P(X_1, X_2, \ldots, X_N | Y_1, Y_2, \ldots, Y_N, \mathcal{P}, \mathcal{Q}, \pi).$$

This is the same as maximizing the probability, but I chose to minimize the negative log probability because (a) the log turns products into sums, which will be convenient and (b) minimizing the negative log probability gives us a formulation that is consistent with algorithms in chapter **??**. The negative log probability factors as

$$-\log\left(\frac{P(X_1, X_2, \ldots, X_N, Y_1, Y_2, \ldots, Y_N | \mathcal{P}, \mathcal{Q}, \pi)}{P(Y_1, Y_2, \ldots, Y_N)}\right)$$

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 minimize

$$-\log P(X_1, X_2, \ldots, X_N, Y_1, Y_2, \ldots, Y_N | \mathcal{P}, \mathcal{Q}, \pi)$$

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.

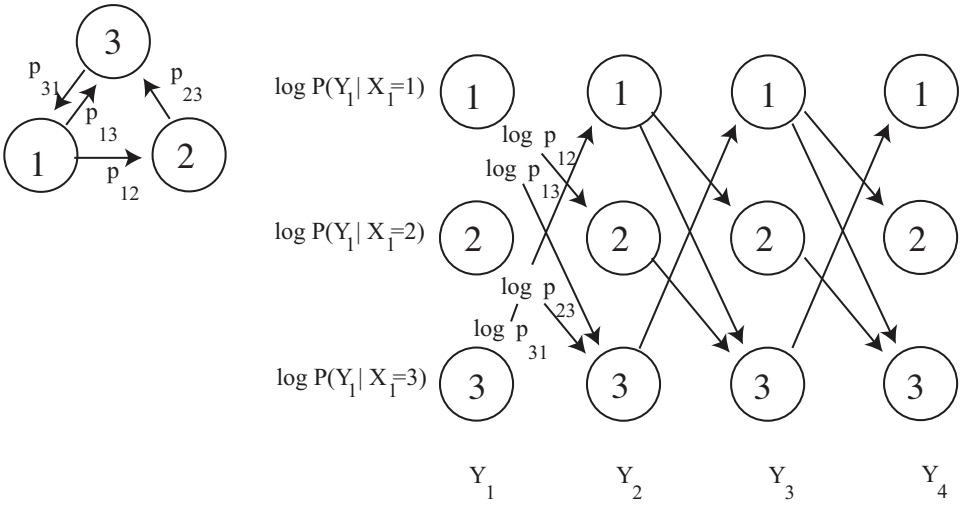


FIGURE 15.1: *At the* **top left**, *a simple state transition model. Each outgoing edge has some probability, though the topology of the model forces two of these probabilities to be* 1. *Below, the trellis corresponding to that model. Each path through the trellis corresponds to a legal sequence of 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*

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 negative 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 15.1)

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 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

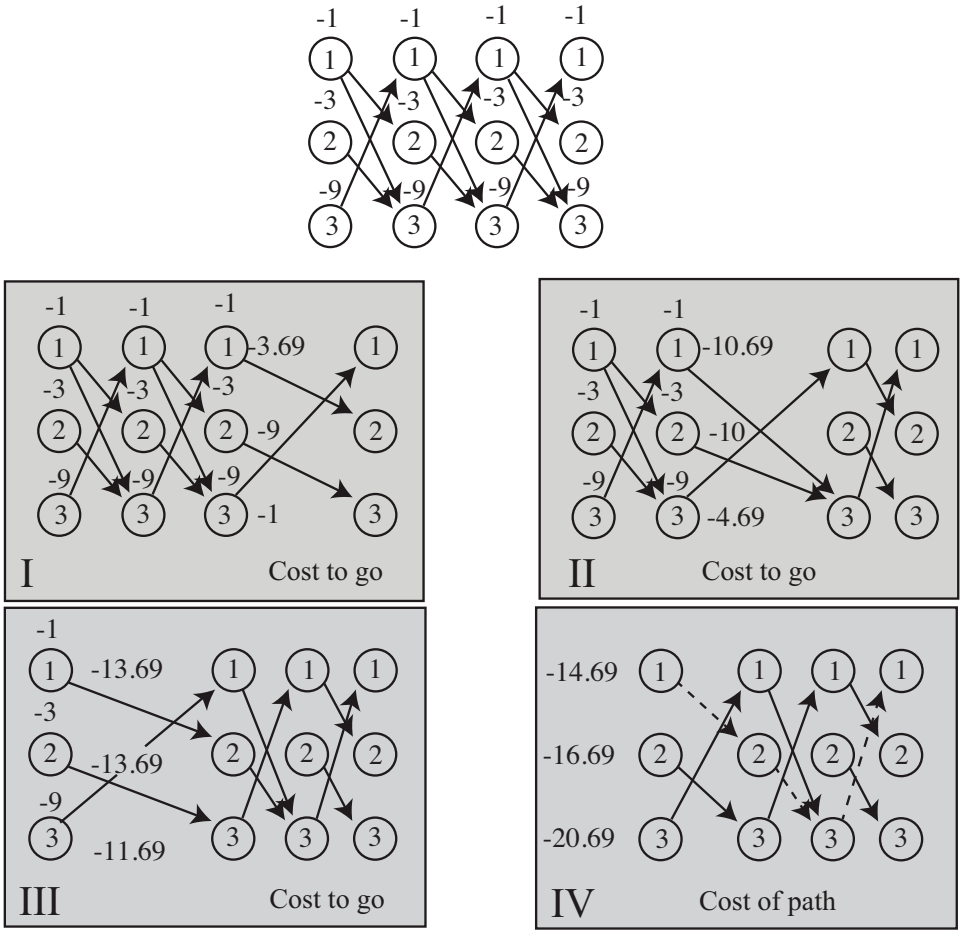


FIGURE 15.2: *An example of finding the best path through a trellis. The probabilities of leaving a node are uniform (and remember,* $\ln 2 \approx -0.69$*). Details in the text.*

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 15.2). 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 15.2). 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 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 15.2) 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 15.2). Finally, to find the value of the best path leaving the first column, we compute the minimum value over all nodes in the first column.

We can also get the path with the minimum 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 15.2.

15.1.3 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

$$C_{N-1}(j) = \min_u \left[-\log P(X_N = u | X_{N-1} = j) - \log P(Y_N | X_N = u) \right]$$

and

$$B_{N-1}(j) = \operatorname*{argmin}_u \left[-\log P(X_N = u | X_{N-1} = j) - \log P(Y_N | X_N = u) \right].$$

You should check this against step I of Figure 15.2

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

$$C_w(j) = \min_u \left[-\log P(X_{w+1} = u | X_w = j) - \log P(Y_{w+1} | X_{w+1} = u) - C_{w+1}(u) \right]$$

and

$$B_w(j) = \operatorname*{argmin}_u \left[-\log P(X_{w+1} = u | X_w = j) - \log P(Y_{w+1} | X_{w+1} = u) - C_{w+1}(u) \right].$$

Check this against steps II and III in Figure 15.2.

15.1.4 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.

*	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 15.1: *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 15.2: *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.*

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

*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 15.3: *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.*

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 sequencezof 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").

15.2 LEARNING AN HMM WITH EM

We have a dataset $\mathbf{Y}$ for which we believe a hidden Markov model is an appropriate model. This dataset consists of R sequences of visible states. The u'th sequence has $N(u)$ elements. We will assume that the observed values lie in a discrete space (i.e. there are O possible values that the Y's can take, and no others). We wish to choose a model that best represents a set of data. Assume, for the moment, that we knew each hidden state corresponding to each visible state. Write $Y_t^{(u)}$ is the observed value for the t'th observed state in the u'th sequence; write $X_t^{(u)}$ for the random variable representing the hidden value for the t'th observed state in the u'th sequence; write s_k for the hidden state values (where k is in the range $1 \dots S$); and write y_k for the possible values for Y (where k is in the range $1 \dots O$).

The hidden Markov model is given by three sets of parameters, π, $\mathcal{P}$ and $\mathcal{Q}$. We will assume that these parameters are not affected by where we are in the sequence (i.e. the model is homogeneous). First, π is an S dimensional vector. The i'th element, π_i of this vector gives the probability that the model starts in state s_i, i.e. $\pi_i = P(X_1 = s_i|\theta)$. Second, $\mathcal{P}$ is an $S \times S$ dimensional table. The i, j'th element of this table gives $P(X_{t+1} = s_j|X_t = s_i)$. Finally, $\mathcal{Q}$ is an $O \times S$ dimensional table. We will write $q_j(y_i) = P(Y_t = y_i|X_t = s_j)$ for the i, j'th element of this table. Note I will write θ to represent all of these parameters together.

Now assume that we *know* the values of $X_t^{(u)}$ for all t, u, (i.e. for each $Y_t^{(u)}$ we know that $X_t^{(u)} = s_i$). Then estimating the parameters is straightforward. We can estimate each by counting. For example, we estimate π_i by counting the number

of sequences where $X_1 = s_i$, then dividing by the total number of sequences. We will encapsulate this knowledge in a function $\delta_t^{(u)}(i)$, where

$$\delta_t^{(u)}(i) = \begin{cases} 1 & \text{if } X_t^{(u)} = s_i \\ 0 & \text{otherwise} \end{cases} .$$

If we know $\delta_t^{(u)}(i)$, we have

$$\begin{aligned} \pi_i &= \frac{\text{number of times in } s_i \text{ at time 1}}{\text{number of sequences}} \\ &= \frac{\sum_{u=1}^{R} \delta_1^{(u)}(i)}{R} \\ \mathcal{P}_{ij} &= \frac{\text{number of transitions from } s_j \text{ to } s_i}{\text{total number of transitions}} \\ &= \frac{\sum_{u=1}^{R} \sum_{t=1}^{N(u)-1} \delta_t^{(u)}(j)\delta_{t+1}^{(u)}(i)}{\sum_{u=1}^{R} [N(u)-1]} \\ q_j(y_i) &= \frac{\text{number of times in } s_j \text{ and observe } Y = y_i}{\text{number of times in } s_j} \\ &= \sum_{u=1}^{R} \sum_{t=1}^{N(u)} \delta_t^{(u)}(j)\delta(Y_t^{(u)}, y_i) \end{aligned}$$

where $\delta(u, v)$ is one if its arguments are equal and zero otherwise.

The problem (of course) is that we *don't know* $\delta_t^{(u)}(i)$. But we have been here before (section 57 and section 57). The situation follows the recipe for EM: we have missing variables (the $X_t^{(u)}$; or, equivalently, the $\delta_t^{(u)}(i)$) where the log-likelihood can be written out cleanly in terms of the missing variables. We assume we know an estimate of the parameters $\hat{\theta}^{(n)}$. We construct

$$Q(\theta; \hat{\theta}^{(n)}) = \mathbb{E}_{\log P(X,Y|\theta)}\left[P(X|Y, \hat{\theta}^{(n)})\right]$$

(the E-step). Then we compute

$$\theta^{(\hat{n+1})} = \underset{\theta}{\text{argmin}}\ Q(\theta; \hat{\theta}^{(n)})$$

(the M-step). As usual, the problem is the E-step. I will not derive this in detail (enthusiasts can easily reconstruct the derivation from what follows together with chapter 57). The essential point is that we need to recover

$$\xi_t^{(u)}(i) = \mathbb{E}_{P(\delta|Y,\hat{\theta}^{(n)})}\left[\delta_t^{(u)}(i)\right] = P(X_t^{(u)} = s_i|Y, \hat{\theta}^{(n)}).$$

For the moment, assume we know these. Then we have

$$\begin{aligned}
\hat{\pi_i}^{(n+1)} &= \text{expected frequency of being in } s_i \text{ at time 1} \\
&= \frac{\sum_{u=1}^{R} \xi_1^{(u)}(i)}{R} \\
\hat{\mathcal{P}}_{ij}^{(n+1)} &= \frac{\text{expected number of transitions from } s_j \text{ to } s_i}{\text{expected number of transitions from state } s_j} \\
&= \frac{\sum_{u=1}^{R}\sum_{t=1}^{N(u)} \xi_t^{(u)}(j)\xi_{t+1}^{(u)}(i)}{\sum_{u=1}^{R}\sum_{t=1}^{N(u)} \xi_t^{(u)}(j)} \\
\hat{q}_j^{(n+1)}(k) &= \frac{\text{expected number of times in } s_j \text{ and observing } Y = y_k}{\text{expected number of times in state } s_j} \\
&= \frac{\sum_{u=1}^{R}\sum_{t=1}^{N(u)} \xi_t^{(u)}(j)\delta(Y_t^{(u)}, y_k)}{\sum_{u=1}^{R}\sum_{t=1}^{N(u)} \xi_t^{(u)}(j)}
\end{aligned}$$

where $\delta(u, v)$ is one if its arguments are equal and zero otherwise.

To evaluate $\xi_t^{(u)}(i)$, we need two intermediate variables: a **forward variable** and a **backward variable**. The forward variable is $\alpha_t^{(u)}(j) = P(Y_1^{(u)}, \ldots, Y_t^{(u)}, X_t^{(u)} = s_j|\hat{\theta}^{(n)})$. The backward variable is $\beta_t^{(u)}(j) = P(\{Y_{t+1}^{(u)}, Y_{t+2}^{(u)}, \ldots, Y_{N(u)}^{(u)}\}|X_t^{(u)} = s_j|\hat{\theta}^{(n)})$. Now assume that we know the values of these variables, we have that

$$\begin{aligned}
\xi_t^{(u)}(i) &= P(X_t^{(u)} = s_i|\hat{\theta}^{(n)}, \mathbf{Y}^{(u)}) \\
&= \frac{P(\mathbf{Y}^{(u)}, X_t^{(u)} = s_i|\hat{\theta}^{(n)})}{P(\mathbf{Y}^{(u)}|\hat{\theta}^{(n)})} \\
&= \frac{\alpha_t^{(u)}(i)\beta_t^{(u)}(i)}{\sum_{i=1}^{k} \alpha_t^{(u)}(i)\beta_t^{(u)}(i)}
\end{aligned}$$

Both the forward and backward variables can be evaluated by induction. We get $\alpha_t^{(u)}(j)$ by observing that:

$$\begin{aligned}
\alpha_1^{(u)}(j) &= P(Y_1^{(u)}, X_1^{(u)} = s_j|\hat{\theta}^{(n)}) \\
&= \pi_j^{(n)} q_j^{(n)}(Y_1).
\end{aligned}$$

Now for all other t's, we have

$$\begin{aligned}
\alpha_{t+1}^{(u)}(j) &= P(Y_1^{(u)}, \ldots, Y_{t+1}^{(u)}, X_{t+1}^{(u)} = s_j | \hat{\theta}^{(n)}) \\
&= \sum_{l=1}^{S} P(Y_1^{(u)}, \ldots, Y_t^{(u)}, Y_{t+1}^{(u)}, X_t^{(u)} = s_l, X_{t+1}^{(u)} = s_j | \hat{\theta}^{(n)}) \\
&= \left(\sum_{l=1}^{S} \left[\begin{array}{c} P(Y_1^{(u)}, \ldots, Y_t^{(u)}, X_t^{(u)} = s_l | \hat{\theta}^{(n)}) \times \\ P(X_{t+1}^{(u)} = s_j | X_t^{(u)} = s_l, \hat{\theta}^{(n)}) \end{array} \right] \right) P(Y_{t+1}^{(u)} | X_{t+1}^{(u)} = s_j, \hat{\theta}^{(n)}) \\
&= \left[\sum_{l=1}^{S} \alpha_t^{(u)}(l) p_{lj}^{(n)} \right] q_j^{(n)}(Y_{t+1})
\end{aligned}$$

We get $\beta_t^{(u)}(j)$ by observing that:

$$\begin{aligned}
\beta_{N(u)}^{(u)}(j) &= P(\text{no further output} | X_{N(u)}^{(u)} = s_j, \hat{\theta}^{(n)}) \\
&= 1
\end{aligned}$$

. Now for all other t we have

$$\begin{aligned}
\beta_t^{(u)}(j) &= P(Y_{t+1}^{(u)}, Y_{t+2}^{(u)}, \ldots, Y_{N(u)}^{(u)} | X_t^{(u)} = s_j, \hat{\theta}^{(n)}) \\
&= \sum_{l=1}^{S} \left[P(Y_{t+1}^{(u)}, Y_{t+2}^{(u)}, \ldots, Y_{N(u)}^{(u)}, X_{t+1}^{(u)} = s_l | X_t^{(u)} = s_j, \hat{\theta}^{(n)}) \right] \\
&= \sum_{l=1}^{S} \left[\begin{array}{l} P(Y_{t+2}^{(u)}, \ldots, Y_{N(u)}^{(u)} | X_{t+1}^{(u)} = s_j, \hat{\theta}^{(n)}) \\ \times P(Y_{t+1}^{(u)}, X_{t+1}^{(u)} = s_l | X_t^{(u)} = s_j, \hat{\theta}^{(n)}) \end{array} \right] \\
&= P(Y_{t+2}^{(u)}, \ldots, Y_{N(u)}^{(u)} | X_{t+1}^{(u)} = s_j, \hat{\theta}^{(n)}) \left(\sum_{l=1}^{S} \left[\begin{array}{l} P(X_{t+1}^{(u)} = s_l | X_t^{(u)} = s_j, \hat{\theta}^{(n)}) \\ \times P(Y_{t+1}^{(u)} | X_{t+1}^{(u)} = s_l \hat{\theta}^{(n)}) \end{array} \right] \right) \\
&= \beta_{t+1}(j) \left(\sum_{l=1}^{S} \left[q_l^{(n)}(Y_{t+1}^{(u)}) p_{lj}^{(n)} \right] \right)
\end{aligned}$$

As a result, we have a simple fitting algorithm, collected in Algorithm 15.1.

Procedure: 15.1 *Fitting Hidden Markov Models with EM*

We fit a model to a data sequence $\mathbf{Y}$ is achieved by a version of EM. We seek the values of parameters $\theta = (\mathcal{P}, \mathcal{Q}, \pi)_i$. We assume we have an estimatel $\hat{\theta}^{(n)}$, and then compute the coefficients of a new model; this iteration is guaranteed to converge to a local maximum of $P(\mathbf{Y}|\hat{\theta})$.

Until $\hat{\theta}^{(n+1)}$ is the same as $\hat{\theta}^{(n)}$
compute the forward variables α and β
using the procedures of algorithms 15.2 and 15.3

compute $\xi_t^{(u)}(i) = \frac{\alpha_t^{(u)}(i)\beta_t^{(u)}(i)}{\sum_{i=1}^{k} \alpha_t^{(u)}(i)\beta_t^{(u)}(i)}$

compute the updated parameters using the procedures of procedure 15.4
end

Procedure: 15.2 *Computing the Forward Variable for Fitting an HMM*

$$\begin{aligned} \alpha_1^{(u)}(j) &= \pi_j^{(n)} q_j^{(n)}(Y_1) \\ \alpha_{t+1}^{(u)}(j) &= \left[\sum_{l=1}^{S} \alpha_t^{(u)}(l) p_{lj}^{(n)}\right] q_j^{(n)}(Y_{t+1}) \end{aligned}$$

Procedure: 15.3 *Computing the Backward Variable for Fitting an HMM*

$$\begin{aligned} \beta_{N(u)}^{(u)}(j) &= 1 \\ \beta_t^{(u)}(j) &= \beta_{t+1}(j) \left(\sum_{l=1}^{S} \left[q_l^{(n)}(Y_{t+1}^{(u)}) p_{lj}^{(n)}\right]\right) \end{aligned}$$

Procedure: 15.4 *Updating Parameters for Fitting an HMM*

$$\begin{aligned}
\hat{\pi}_i^{(n+1)} &= \frac{\sum_{u=1}^{R} \xi_1^{(u)}(i)}{R} \\
\hat{\mathcal{P}}_{ij}^{(n+1)} &= \frac{\sum_{u=1}^{R} \sum_{t=1}^{N(u)} \xi_t^{(u)}(j) \xi_{t+1}^{(u)}(i)}{\sum_{u=1}^{R} \sum_{t=1}^{N(u)} \xi_t^{(u)}(j)} \\
\hat{q}_j(k)^{(n+1)} &= \frac{\sum_{u=1}^{R} \sum_{t=1}^{N(u)} \xi_t^{(u)}(j) \delta(Y_t^{(u)}, y_k)}{\sum_{u=1}^{R} \sum_{t=1}^{N(u)} \xi_t^{(u)}(j)}
\end{aligned}$$

where $\delta(u, v)$ is one if its arguments are equal and zero otherwise.

15.3 YOU SHOULD

15.3.1 remember these definitions:

15.3.2 remember these terms:

15.3.3 remember these facts:

15.3.4 be able to:

- Set up a simple HMM and use it to solve problems.
- Learn a simple HMM from data

CHAPTER 16

Discriminative Learning for Sequence Models

16.1 GRAPHICAL MODELS

We now adopt a convention that allows us to draw some kinds of model as a graph. Assume we have a probability distribution over a collection of R variables, $U_1, \ldots U_R$. Now consider the probability distribution $P(U_1, \ldots, U_R)$. We assume that this distribution can be factored into a set of terms where each term depends on some pair of variables. This means that there are some functions ϕ so that

$$-\log P(U_1, \ldots, U_R) = \sum_{(i,j) \in \text{pairs}} \phi(U_i, U_j) + K$$

where K is the log of the normalizing constant, and is of no interest to us at present. A model of this form can be drawn as a graph. One draws a vertex for each variable, and an edge for each ϕ. It is usual to draw a circle for vertices, rather than a dot (this comes in useful in the next paragraph). These models are known as **graphical models**.

Notice that an HMM has this form. The variables are the hidden *and* the observed states in the HMM. We will fill the circle if the variable's value is observed, and add arrowheads to the edges in this graph according to the following rule. If $\phi(U_i, U_j) = -\log P(U_i|U_j) + C$, then we add an arrowhead from the node representing U_j to the node representing U_i (the arrow points toward the variable that is "generated"). This yields a drawing of an HMM as Figure 16.1.

The advantage of drawing models this way is that it exposes what is important in inference. There are no proofs here (though if you've done a good algorithms course you could likely fill in the details) but we will encounter this point here and in chapter **??**. The drawing helps understand why inference in an HMM is efficient. 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

$$\begin{aligned} P(Y_1, Y_2, \ldots, Y_N, X_1, X_2, \ldots, X_N) = \; & P(X_1)P(Y_1|X_1) \\ & P(X_2|X_1)P(Y_2|X_2) \\ & \ldots \\ & P(X_N|X_{N-1})P(Y_N|X_N). \end{aligned}$$

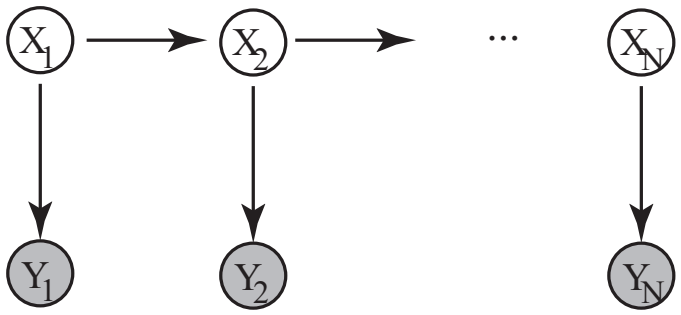


FIGURE 16.1: *An HMM is an example of a graphical model. The joint probability distribution of the hidden variables X_i and the observations Y_i factors as in the text. Each variable appears as a vertex. There is an edge between pairs of variables that appear in the factorization. These edges have an arrowhead drawn according to conventions about conditional probability. Finally, the observed values are shaded.*

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 the reasoning about dynamic programming *required* the costs to actually be log probabilities. You should revisit chapter 57 to check this point. All that matters is that one can recursively update costs.

Now assume we have a graphical model. We partition the U variables into two groups. The X_i are unknown and we need to recover them by inference, and the Y_j are known. In turn, this means that some of the *phi* become constants because both arguments are fixed; these are of no interest, and we can drop them from the drawing. Others of the ϕ become effectively functions of a single argument, because the value of the other argument is known and fixed. It is usual to keep these edges in the drawing, but shade the known argument. In turn, inference involves solving for the values of a set of discrete variables to minimize the value of an objective function $f(X_1, \ldots, X_n)$.

This objective function is a sum of two kinds of term. There are **unary terms** or **vertex terms** which are functions that take one argument, and which we write $V(X_i)$. These are the ϕ for which we know the value of one argument. Notice that the variable identifies *which* vertex function we are talking about; the convention follows probability notation. There are **binary terms** or **edge terms** which are functions that take two arguments, and which we write $E(X_i, X_j)$, using the same convention. For example, in the case of the HMM, the variables would be the

hidden states, the unary terms would be the negative logs of emission probabilities, and binary terms would be the negative logs of transition probabilities.

16.1.1 Graphical Models that allow Easy Inference

It is helpful to represent a probability model as a graph if you can, because doing so gives some insight into how easy or hard inference will be. We know that inference for an HMM is easy (i.e. can be done in polynomial time). It turns out that, if the graph for the probability model is a forest, then inference will be easy. I will sketch this point here.

Start with a graphical model where the graph is a **chain graph**. A chain graph looks like a chain (hence the name), and is the graph that arises from an HMM (Figure 57; remember that the shaded nodes correspond to known values, and produce the vertex functions). The objective function for inference is then

$$f(X_1, \ldots, X_n) = \sum_{i=1}^{i=n} V(X_i) + \sum_{i=1}^{i=n-1} E(X_i, X_{i+1})$$

and we wish to minimize this function. This is actually a family of functions (one function for each n), but it is clear which one is intended, because you just have to count the arguments. Now we define a new function, the **cost-to-go function**, with a recursive definition. Write

$$f^{(n-1)}_{\text{cost-to-go}}(X_{n-1}) = \min_{X_n} E(X_{n-1}, X_n) + V(X_n).$$

This function represents the effect of a choice of value for X_{n-1} on the terms that involve X_n, where one chooses the best possible choice of X_n. This means that

$$\min_{X_1, \ldots, X_n} f(X_1, \ldots, X_n)$$

is equal to

$$\min_{X_1, \ldots, X_{n-1}} \left(f(X_1, \ldots, X_{n-1}) + f^{(n-1)}_{\text{cost-to-go}}(X_{n-1}) \right),$$

which means that we can eliminate the nth variable from the optimization by replacing the term $E(X_{n-1}, X_n) + V(X_n)$ with a function of X_{n-1}. This function is obtained by minimizing 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 $E(X_{n-1}, X_n) + V(X_n)$ 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 minimum, 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

$$\min_{X_{n-1}, X_n} [E(X_{n-2}, X_{n-1}) + V(X_n - 1) + E(X_{n-1}, X_n) + V(X_n)]$$

is equal to

$$\min_{X_{n-1}} \left[E(X_{n-2}, X_{n-1}) + V(X_n - 1) + \left(\min_{X_n} E(X_{n-1}, X_n) + V(X_n) \right) \right].$$

But all this can go on recursively, yielding

$$f^{(k)}_{\text{cost-to-go}}(X_k) = \min_{X_{k+1}} E(X_k, X_{k+1}) + V(X_k) + f^{(k+1)}_{\text{cost-to-go}}(X_{k+1}).$$

This is basically what we did with a trellis in Section 15.1.2. Notice that

$$\min_{X_1, \ldots, X_n} \; f(X_1, \ldots, X_n)$$

is equal to

$$\min_{X_1, \ldots, X_{n-1}} \; \left(f(X_1, \ldots, X_{n-1}) + f^{(n-1)}_{\text{cost-to-go}}(X_{n-1}) \right)$$

which is equal to

$$\min_{X_1, \ldots, X_{n-2}} \; \left(f(X_1, \ldots, X_{n-2}) + f^{(n-2)}_{\text{cost-to-go}}(X_{n-2}) \right),$$

and we can apply the recursive definition of the cost-to-go function to get

$$\min_{X_1, \ldots, X_n} \; f(X_1, \ldots, X_n) = \min_{X_1} \; \left(f(X_1) + f^{1}_{\text{cost-to-go}}(X_1) \right),$$

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. Passing from one tree to a forest is straightforward, as you can do each tree separately. The fact that the algorithm works is a combinatorial insight. In

section 57, we will see graphical models that do not admit easy inference because their graph is not a forest. In those cases, we will need to use approximation strategies for inference.

When easy inference is available, there is a natural and important learning strategy which is very different from using EM. Assume we have a collection of example sequences of observations (write $\mathbf{Y}^{(u)}$ for the u'th such sequence) *and* of hidden states (write $\mathbf{X}^{(u)}$ for the u'th such sequence). We construct a family of cost functions $C(X, Y; \theta)$ parametrized by θ. We then choose the θ so that inference applied to the cost function yields the right answer. So we want to choose θ so that

$$\underset{\mathbf{X}}{\operatorname{argmin}} \; C(\mathbf{Y}^{(u)}, \mathbf{X}; \theta)$$

is $\mathbf{X}^{(u)}$ or "close to" it. The details require quite a lot of work, which we do below. What is important now is that this strategy applies to *any* model where easy inference is available. This means we can generalize quite significantly from HMM's, which is the next step.

16.2 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 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

$$\begin{aligned} P(Y_1, Y_2, \ldots, Y_N, X_1, X_2, \ldots, X_N) \quad = \quad & P(X_1)P(Y_1|X_1) \\ & P(X_2|X_1)P(Y_2|X_2) \\ & \ldots \\ & P(X_N|X_{N-1})P(Y_N|X_N). \end{aligned}$$

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.

16.2.1 MEMM's and Label Bias

One alternative would be to look for a model that factors in a different way. For example, we could consider

$$\begin{aligned} P(X_1, X_2, \ldots, X_N | Y_1, Y_2, \ldots, Y_N) &= P(X_1|Y_1) \times \\ & \quad P(X_2|Y_2, X_1) \times \\ & \quad P(X_3|X_2, Y_2) \times \\ & \quad \ldots \times \\ & \quad P(X_N|X_{N-1}, Y_N). \end{aligned}$$

This means that

$$\begin{aligned} -\log P(X_1, X_2, \ldots, X_N | Y_1, Y_2, \ldots, Y_N) &= -\log P(X_1|Y_1) \\ & \quad -\log P(X_2|Y_2, X_1) \\ & \quad -\log P(X_3|X_2, Y_2) \\ & \quad \ldots \\ & \quad -\log P(X_N|X_{N-1}, Y_N). \end{aligned}$$

This is still a set of edge and vertex functions, but notice there is only one vertex function ($-\log P(X_1|Y_1)$). All the remaining terms are edge functions. Models of this form are known as **maximum entropy markov models** or **MEMM**'s.

These models are deprecated. You should not use one without a special justification. Rather than just ignoring these models, I have decribed them because the reason they are deprecated is worth understanding. These models very often ignore measurements, as a result of their structure. To see this, assume we have fitted a model, and wish to recover the best sequence of X_i corresponding to a given sequence of observations Y_i. We must minimize

$$\begin{aligned} -\log P(X_1, X_2, \ldots, X_N | Y_1, Y_2, \ldots, Y_N) &= -\log P(X_1|Y_1) \\ & \quad -\log P(X_2|Y_2, X_1) \\ & \quad -\log P(X_3|X_2, Y_2) \\ & \quad \ldots \\ & \quad -\log P(X_N|X_{N-1}, Y_N). \end{aligned}$$

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 16.2. Notice that when the model is in state 1, it can only transition to state 2. In turn, this means that $-\log P(X_{i+1} =$

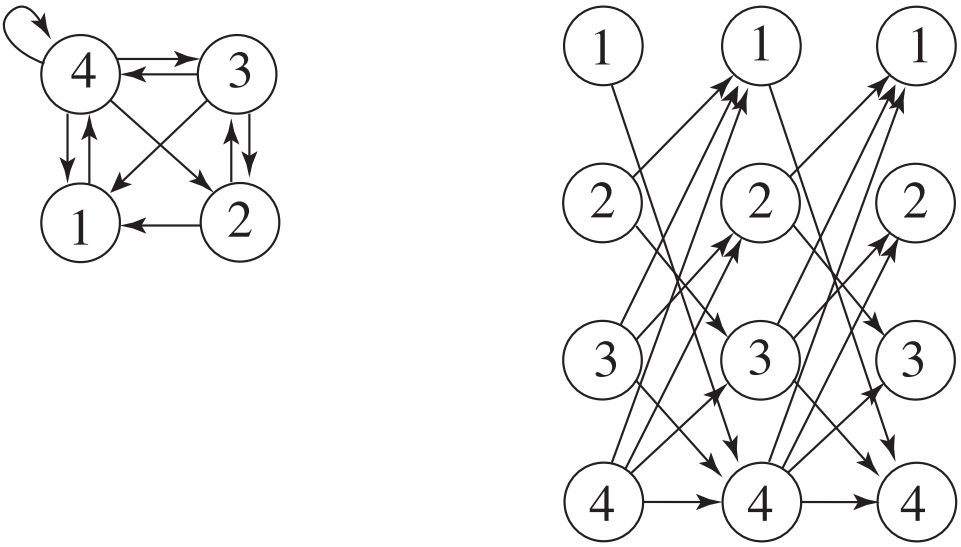


FIGURE 16.2:

$2|X_i = 1, Y_i) = 0$ *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 most other paths must share weights between many outgoing edges; but $1 \to 4$ is very cheap, and $2 \to 1$ 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.

16.2.2 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. We can achieve this by ensuring that the graphical representation of the model is a chain graph. We'd like the resulting model to be discriminative in form — i.e. we should be able to interpret the vertex functions as $-\log P(X_i|Y_i)$ — but we don't want an MEMM.

We start with the cost functions. 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 use $V_i(a) = -\log P(X_i = a|Y_i)$ as a vertex cost function. We will assume that $E_i(a, b)$ and $V_i(a)$ are bounded (straightforward, because the variables are all discrete anyhow), but we will not apply any direct probabilistic interpretation to

this function. Instead, we interpret the whole model as

$$-\log P(X_1 = x_1, X_2 = x_2, \ldots, X_N = x_n | Y_1, Y_2, \ldots, Y_N) = [V_1(x_1) + E_1(x_1, x_2) + V_2(x_2) + E_2(x_2, x_3) + \ldots V_N$$

where K is the log of the 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**.

Notice the minus sign. This means that the best sequence has the smallest value of

$$\begin{bmatrix} V_1(x_1) + E_1(x_1, x_2) + \\ V_2(x_2) + E_2(x_2, x_3) + \\ \ldots \\ V_N(x_n) \end{bmatrix},$$

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 have a probabilistic interpretation of E_i, so we can't reconstruct an appropriate table of values by (say) counting. 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

$$P(Y_1, \ldots, Y_N, |\theta)$$

because we don't know $P(Y_1, \ldots, Y_N, |\theta)$. Instead, our model encodes $P(X|Y)$. To use Bayes rule, we'd need to know $P(Y)$, which brings us back where we started. Rather than wrestle with this issue, we restate the problem.

16.3 DISCRIMINATIVE LEARNING OF CRFS

A really powerful strategy for learning a CRF follows by obtaining both the observations and the 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.

16.3.1 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 choose V to be a weighted sum of these basis functions, so 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.

Worked example 16.1 *Constructing V for sequences of digits*

Construct $\phi_j^{(v)}(X, Y)$ for sequences of digits, assuming that the observations are inked numerals (like MNIST), and that they appear in a window of fixed size (like MNIST).

Solution: There are a variety of possible approaches. Here are three.

- Multinomial logistic regression works quite well on MNIST using just the pixel values as features. This means that you 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. Each of these functions is a $\phi_j^{(v)}$.
- For each possible numeral x and each pixel location p 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 $\phi_j^{(v)}$.
- For each class x, we will build several different 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 $\phi^{(v)}(X, Y) = g_{i,X}(Y)$. We index these feature functions in any way that seems convenient.

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 $\phi^{(e)}_{ab}(U,V) = \mathbb{I}_{[U=a]}\mathbb{I}_{[V=b]}$. 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 $\theta^{(e)}_{ab}$, and construct $E(U,V) = \sum_{a,b} \theta^{(e)}_{ab}\phi^{(e)}_{ab}(U,V)$. 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 $\phi^{(e)}_{ab}(U,V;Y_i,Y_{i+1})$. 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 $\theta^{(e)}_j$, and construct $E(U,V) = \sum_j \theta^{(e)}_j g_{j,UV}(Y_i,Y_{i+1})$. 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

$$C(\mathbf{x};\mathbf{y},\theta) = \sum_{i=1}^{N}\left[\sum_j \theta^{(v)}_j \phi^{(v)}_j(x_i,y_i)\right] + \sum_{i=1}^{N-1}\left[\left(\sum_l \theta^{(e)}_l \phi^{(e)}_l(x_i,x_{i+1})\right)\right].$$

Notice that this cost function is linear in θ. We will use this to build a search for the best setting of θ.

16.3.2 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 $x^{(i)}_j$ 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

$$\mathbf{x}^{+,i} = \operatorname*{argmin}_{\mathbf{x}} C(\mathbf{x};\mathbf{y}^{(i)},\theta)$$

(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

$$\mathbf{x}^{(*,i)} = \underset{\mathbf{x}}{\text{argmin}} \; C(\mathbf{x}; \mathbf{y}^{(i)}, \theta) - d(\mathbf{x}, \mathbf{x}^{(i)}).$$

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.

16.3.3 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.

16.4 YOU SHOULD

16.4.1 remember these terms:

16.5 YOU SHOULD

16.5.1 remember these definitions:

16.5.2 remember these terms:

16.5.3 remember these facts:

16.5.4 remember these procedures:

16.5.5 be able to:

-

CHAPTER 17

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.

17.1 USEFUL BUT INTRACTABLE EXAMPLES

17.1.1 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 17.1), 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

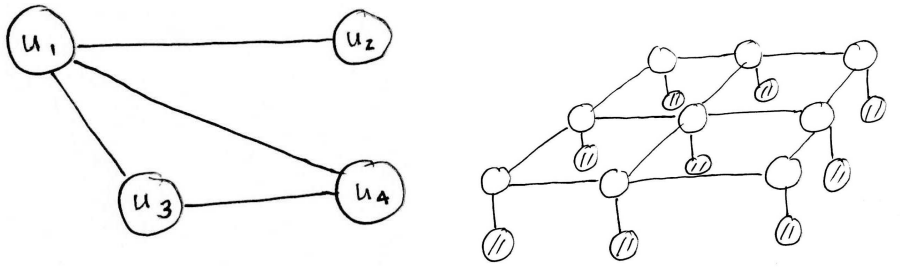


FIGURE 17.1: *On the* **left**, *a simple Boltzmann machine. Each U_i has two possible states, so the whole thing has 16 states. Different choices of the constants coupling the U's along each edge lead to different probability distributions. On the* **right**, *this Boltzmann machine adapted to denoising 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

$$Z(\theta) = \sum_{\text{all values of U}} \left[\exp\left(-E(U|\theta)\right) \right].$$

17.1.2 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 17.1).

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.

17.1.3 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.

17.1.4 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.

17.1.5 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 17.1). 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 17.1 *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 17.1). 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 17.2 *Denoising MRF - II*

Write out $\Theta^{(ij)}$ for the $\theta(H_i, H_j)$ with the form given in example 17.1 using the one-hot vector notation.

Solution: This is more a check you have the notation. $c\mathcal{I}$ is the answer.

Worked example 17.3 *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

$$\beta_1 = \begin{pmatrix} 128^2 & \text{first component} \\ \dots & \\ (i-128)^2 & i\text{'th component} \\ \dots & \\ 127^2 & \end{pmatrix}$$

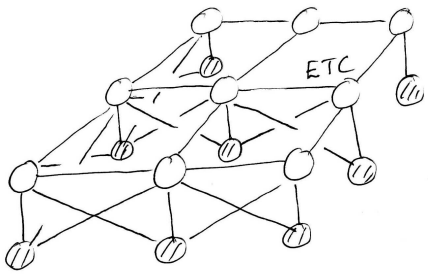


FIGURE 17.2: *The graph of an MRF adapted to image segmentation. The shaded nodes represent the known pixel values (X_i in the text) and the open nodes represent the (unknown, and to be inferred) labels H_i. A particular hidden node may depend on many pixels, because we will use all these pixel values 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 17.2). 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.

17.1.6 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 17.4 *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 17.4 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 17.4) an MRF where inference is equivalent to max-cut, which is NP hard.

Worked example 17.5 *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 17.5 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.

17.2 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**.

17.2.1 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) $P(X)$.

17.2.2 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 $\mathbb{D}(Q(H) \parallel P(H|X))$. But, because $\log P(X)$ is fixed, when F_Q goes down, $\mathbb{D}(Q(H) \parallel P(H|X))$ must go down too. Furthermore, a minimum of F_Q will correspond to a minimum of $\mathbb{D}(Q(H) \parallel P(H|X))$. 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 $\mathbb{D}(Q(H) \parallel P(H|X))$. We use that $Q(H)$ as our approximation to $P(H|X)$, and extract whatever information we want from $Q(H)$.

17.3 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 $Q(H) = q_1(H_1)q_2(H_2)\dots q_N(H_N)$. 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

$$\begin{aligned} \mathbb{E}_Q[\log Q] &= \mathbb{E}_{q_1(H_1)\dots q_N(H_N)}[\log q_1(H_1) + \dots \log q_N(H_N)] \\ &= \mathbb{E}_{q_1(H_1)}[\log q_1(H_1)] + \dots \mathbb{E}_{q_N(H_N)}[\log q_N(H_N)] \end{aligned}$$

where we get the second step by noticing that

$$\mathbb{E}_{q_1(H_1)\dots q_N(H_N)}[\log q_1(H_1)] = \mathbb{E}_{q_1(H_1)}[\log q_1(H_1)]$$

(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

$$\mathbb{E}_Q[\log P(H,X)] = \left(\begin{array}{c} q_i(-1)\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1,\dots,H_i=-1,\dots,H_N,X)] + \\ q_i(1)\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1,\dots,H_i=1,\dots,H_N,X)] \end{array} \right).$$

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

$$\mathbb{E}_{Q_{\hat{i}}}[\log P(H_1, \ldots, H_i = 1, \ldots, H_N, X)]$$

(which works the same) you will get

$$\begin{aligned}
\log q_i(1) &= \mathbb{E}_{Q_{\hat{i}}}[\log P(H_1, \ldots, H_i = 1, \ldots, H_N, X)] + K \\
&= \sum_{j \in \mathcal{N}(i) \cap H} [\theta_{ij}(2\pi_j - 1)] + \sum_{j \in \mathcal{N}(i) \cap X} [\theta_{ij} X_j] + K \\
\text{and} & \\
\log q_i(-1) &= \mathbb{E}_{Q_{\hat{i}}}[\log P(H_1, \ldots, H_i = -1, \ldots, H_N, X)] + K \\
&= \sum_{j \in \mathcal{N}(i) \cap H} [-\theta_{ij}(2\pi_j - 1)] + \sum_{j \in \mathcal{N}(i) \cap X} [-\theta_{ij} X_j] + K
\end{aligned}$$

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**.

PART SIX

DEEP NETWORKS

CHAPTER 18

Classification with Neural Networks

18.1 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.

18.1.1 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

$$p(\text{class} = i|\mathbf{x}, \theta) = s_i(\mathbf{o}(\mathbf{x}, \theta)).$$

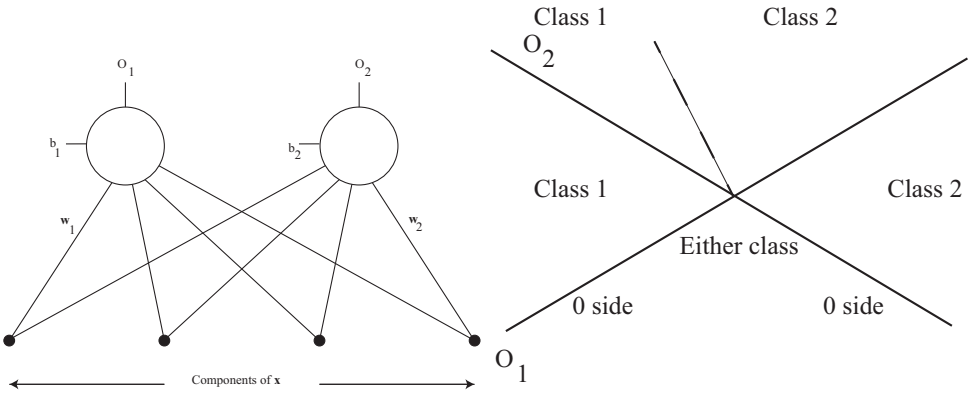


FIGURE 18.1: *On the* **left**, *two units observing an input vector, and providing outputs. On the* **right**, *the decision boundary for two units classifying a point on the plane into one of two classes. The angle of the dashed line depends on the magnitudes of* $\mathbf{w}_1$ *and* $\mathbf{w}_2$.

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.

18.1.2 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

$$\frac{1}{N} \sum_{i \in \text{data}} \left[\left\{ -\mathbf{y}_i^T \log \mathbf{s}(\mathbf{o}(\mathbf{x}_i, \theta)) \right\} \right].$$

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 57), 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 57), we write λ for a weight applied to the penalty. Our cost function is then

$$S(\theta, \mathbf{x}; \lambda) = \underbrace{\frac{1}{N} \sum_{i \in \text{data}} \left[\left\{ -\mathbf{y}_i^T \log \mathbf{s}(\mathbf{o}(\mathbf{x}_i, \theta)) \right\} \right]}_{\text{(misclassification loss)}} + \underbrace{\frac{\lambda}{2} \sum_{u \in \text{units}} \mathbf{w}_u^T \mathbf{w}_u}_{\text{(penalty)}}$$

18.1.3 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.

18.2 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 18.2 - 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;

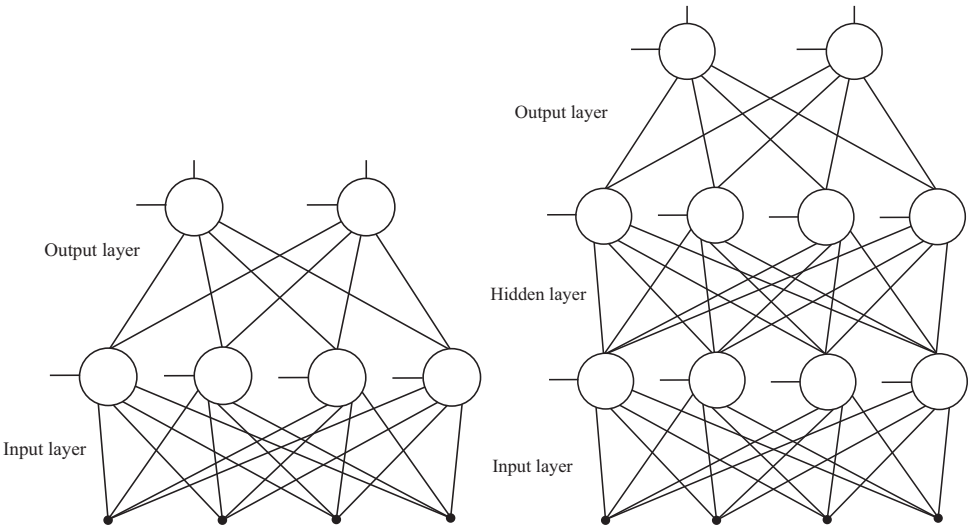


FIGURE 18.2: *On the* **left**, *an input layer connected to an output layer. The units in the input layer take the inputs and compute features; the output layer turns these features into output values that will be turned into class probabilities with the softmax function. On the* **right**, *there is 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.

18.2.1 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

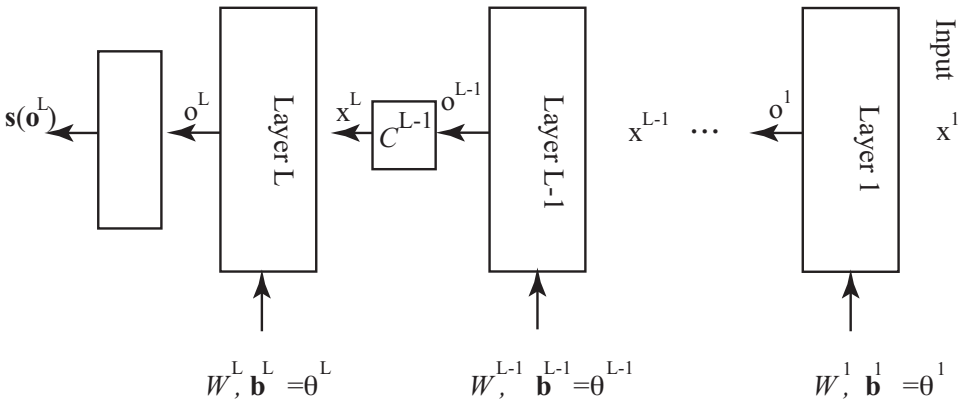


FIGURE 18.3: *Notation for layers, inputs, etc.*

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.

18.2.2 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 18.1.3, very aggressively. We have

$$\nabla_{\theta^L} L(\mathbf{y}_i, \mathbf{s}(\mathbf{o}^L(\mathbf{x}_i, \theta))) = -\mathbf{y}_i^T \mathcal{J}_{\log \mathbf{s}; \mathbf{o}^L} J_{\mathbf{o}^L; \theta^L}$$

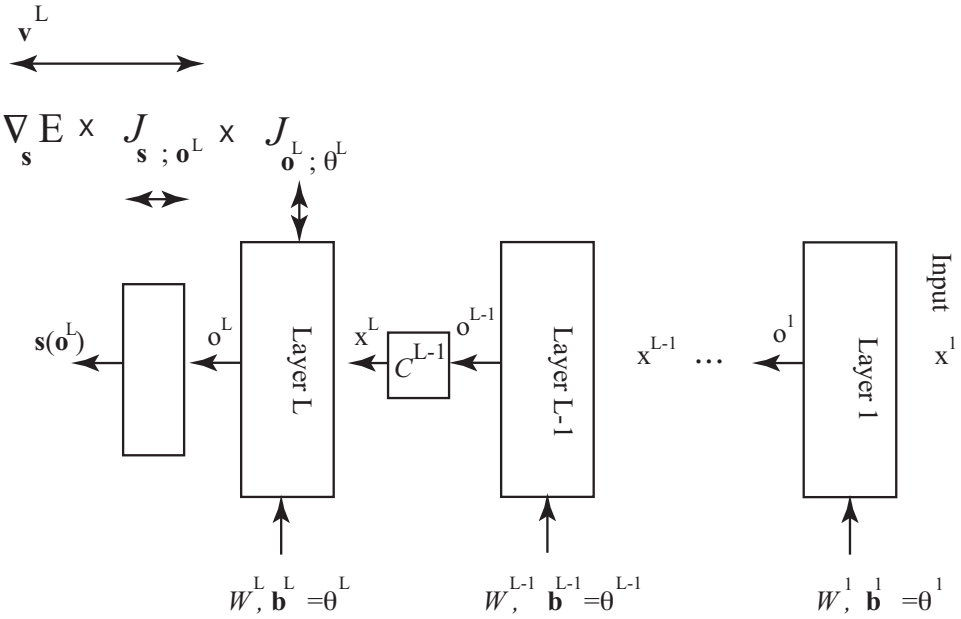


FIGURE 18.4: *Constructing the gradient with respect to θ^L.*

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 18.1.3 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 $J_{\mathbf{o}^{L-1};\theta^{L-1}}$ appears in section 18.1.3. But $\mathbf{o}^L$ depends on θ^{L-2} through $\mathbf{x}^L$ which is a function of $\mathbf{x}^{L-1}$ which is a function of θ^{L-2}, so that

$$\nabla_{\theta^{L-2}} E_i = -\mathbf{y}_i^T \mathcal{J}_{\log \mathbf{s};\mathbf{o}^L} J_{\mathbf{o}^L;\mathbf{x}^L} J_{\mathbf{x}^L;\mathbf{x}^{L-1}} J_{\mathbf{x}^{L-1};\theta^{L-2}}$$

(again, look carefully at the subscripts on each of the Jacobians). Because $\mathbf{x}^L = \mathcal{C}^L \mathbf{o}^{L-1}$, we have that

$$J_{\mathbf{x}^L;\mathbf{x}^{L-1}} = \mathcal{C}^L J_{\mathbf{o}^{L-1};\mathbf{x}^{L-1}}$$

We can now get to the point. We have a recursion, which can be made more

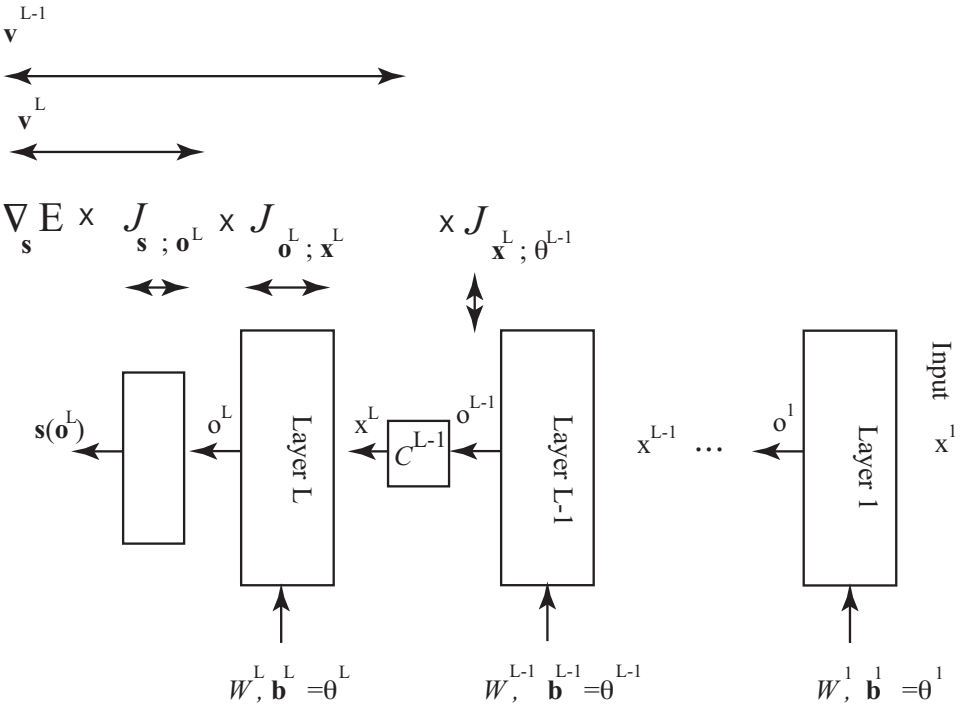


FIGURE 18.5: *Constructing the gradient with respect to θ^{L-1}.*

obvious with some notation. We have

$$\begin{aligned}
\mathbf{v}^L &= \left(\nabla^L_{\mathbf{o}} E_i\right) \\
\nabla_{\theta^L} E_i &= \mathbf{v}^L \mathcal{J}_{\mathbf{o}^L;\theta^L} \\
\nabla_{\theta^{L-1}} E &= \mathbf{v}^L \mathcal{J}_{\mathbf{o}^L;\mathbf{x}^L} \mathcal{J}_{\mathbf{x}^L;\theta^{L-1}} \\
&\cdots \\
\nabla_{\theta^{i-1}} E &= \mathbf{v}^L \mathcal{J}_{\mathbf{o}^L;\mathbf{x}^L} \cdots \mathcal{J}_{\mathbf{x}^{i+1};\mathbf{x}^i} \mathcal{J}_{\mathbf{x}^i;\theta^{i-1}} \\
&\cdots
\end{aligned}$$

But look at the form of the products of the matrices. We don't need to remultiply all those matrices; instead, we are attaching a new term to a product we've already

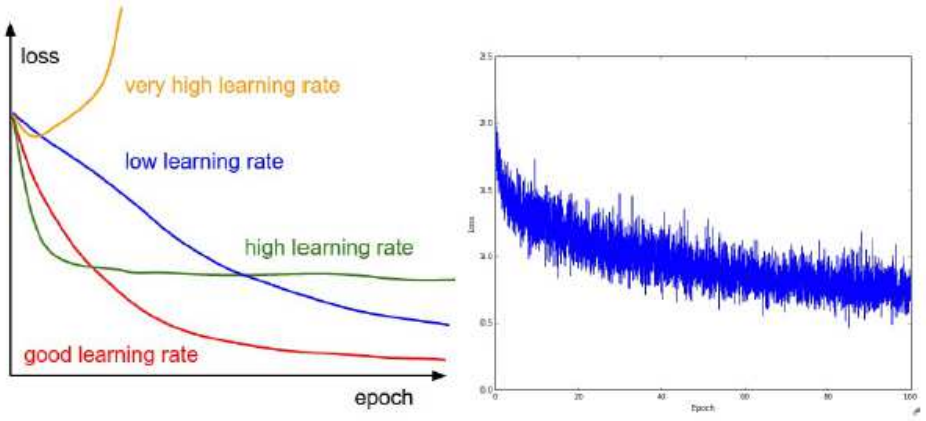


FIGURE 18.6: *On the* **left**, *a cartoon of the error rate encountered during training a multilayer network and the main phenomena you might observe. On the* **right**, *an actual example. Each of these figures is from the excellent course notes for the Stanford class cs231n Convolutional Neural Networks for Visual Recognition, written by Andrej Karpathy. You can find these notes at: http://cs231n.stanford.edu.*

computed. All this is more cleanly written as:

$$\begin{aligned}
\mathbf{v}^L &= (\nabla^L_{\mathbf{o}} E_i) \\
\nabla_{\theta^L} E_i &= \mathbf{v}^L \mathcal{J}_{\mathbf{o}^L;\theta^L} \\
\mathbf{v}^{L-1} &= \mathbf{v}^L \mathcal{J}_{\mathbf{o}^L;\mathbf{x}^L} \\
\nabla_{\theta^{L-1}} E &= \mathbf{v}^{L-1} \mathcal{J}_{\mathbf{x}^L;\theta^{L-1}} \\
&\dots \\
\mathbf{v}^{i-1} &= \mathbf{v}^i \mathcal{J}_{\mathbf{x}^{i+1};\mathbf{x}^i} \\
\nabla_{\theta^{i-1}} E &= \mathbf{v}^{i-1} \mathcal{J}_{\mathbf{x}^i;\theta^{i-1}} \\
&\dots
\end{aligned}$$

Remember here that

$$\begin{aligned}
J_{\mathbf{x}^i;\mathbf{x}^{i-1}} &= \mathcal{C}^i J_{\mathbf{o}^{i-1};\mathbf{x}^{i-1}} \\
J_{\mathbf{x}^i;\theta^{i-1}} &= \mathcal{C}^L J_{\mathbf{o}^{i-1};\theta^{i-1}}
\end{aligned}$$

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**.

18.2.3 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 18.2.4, 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 18.6). 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.

18.2.4 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 $\left[x^{(r)}, y^{(r)}\right] = \left[(1-\epsilon\eta)x^{(r-1)}, (1-\eta)y^{(r-1)}\right]$. 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 18.7 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 18.8).

You may have learned that Newton's method resolves this problem. This is 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

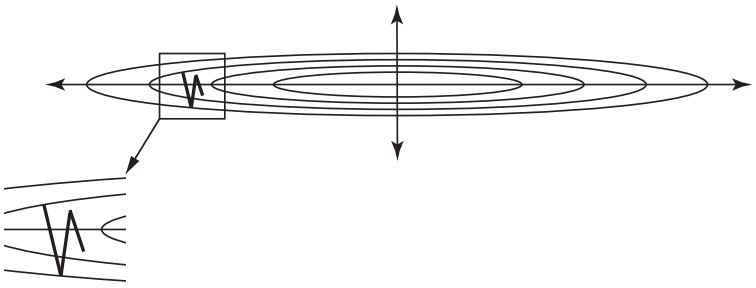


FIGURE 18.7: *A plot of the level curves (curves of constant value) of the function* $f(x, y) = (1/2)(\epsilon x^2 + y^2)$. *Notice that the value changes slowly with large changes in* x, *and quickly with small changes in* y. *The gradient points mostly toward the* x*-axis; this means that gradient 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.*

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

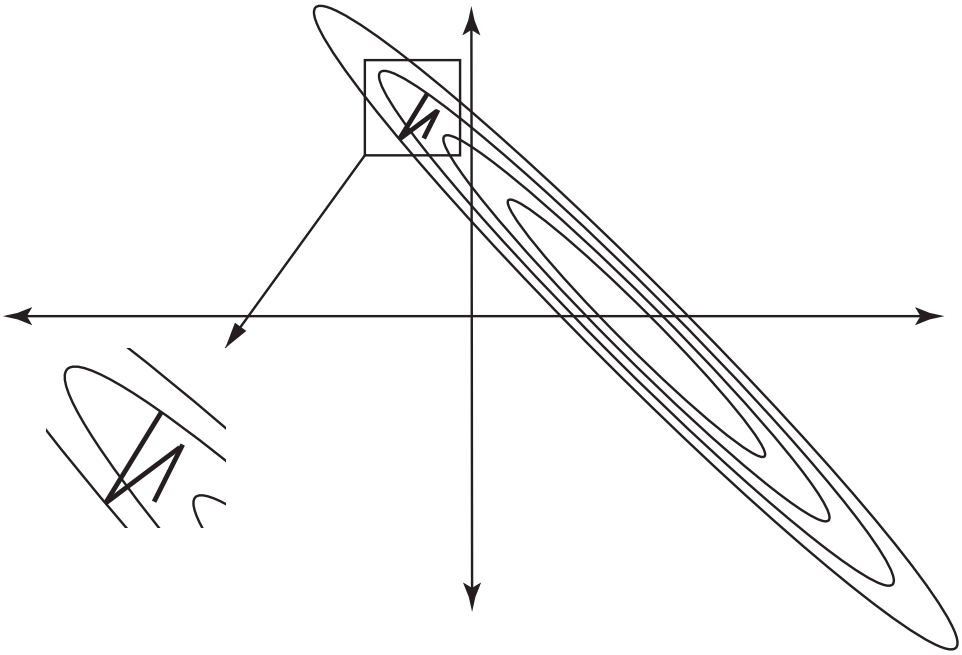


FIGURE 18.8: *Rotating and translating a function rotates and translates the gradient; this is a picture of the function of figure 18.7, but now rotated and translated. The problem of zig-zagging remains. This is important, because it means that we may have serious difficulty choosing a good change of coordinates.*

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.

18.2.5 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

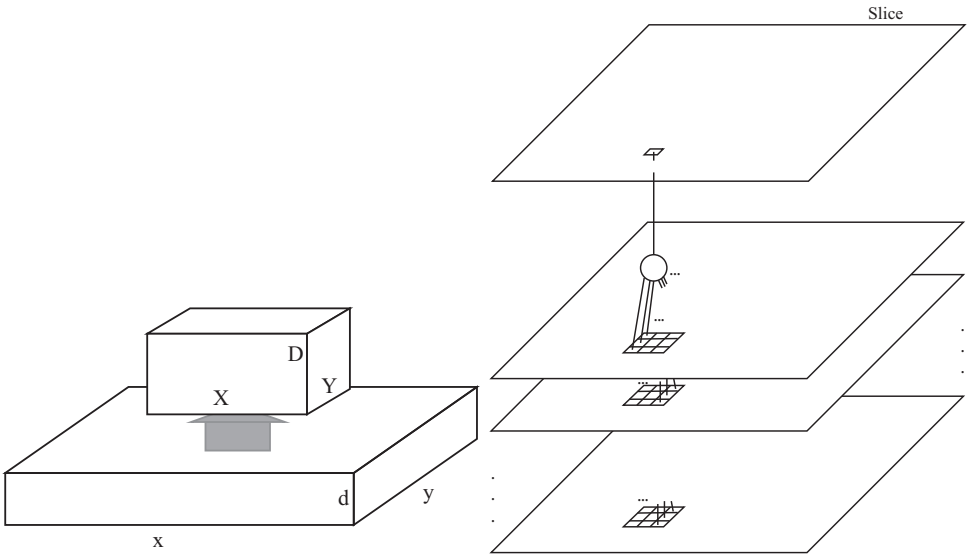


FIGURE 18.9: *Terminology for building a convolutional layer. On the* **left**, *a layer turns one block of data into another. The x and y coordinates index the position of a location in the block, and the d coordinate identifies the data item at that point. A natural example of a block is a color image, 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.*

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.

18.2.6 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 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 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 57) to allow inputs to bypass layers, so that poorly trained layers create fewer difficulties.

18.3 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.

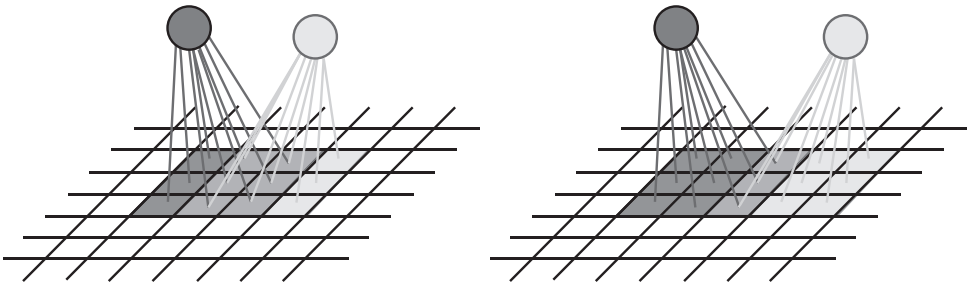


FIGURE 18.10: *Figure 18.9 shows units taking an input block and creating an output block. Each unit is fed by a window on the spatial dimensions of the input block. The window is advanced by the* stride *to feed the next unit. On the* **left***, two units fed by 3x3 windows with a stride of 1. I have 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.*

18.3.1 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 18.9; 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 18.9 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 18.10)

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.

18.3.2 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.

18.3.3 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

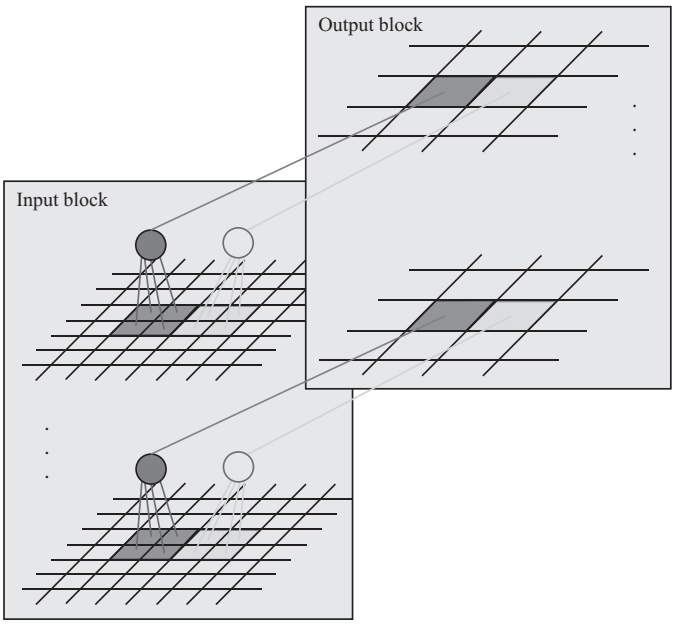


FIGURE 18.11: *In a pooling layer, pooling units compute the largest value of their inputs, then pass it on. The most common case is 2x2, illustrated here. We tile each slice with 2x2 windows that do not overlap. Pooling units compute the max, then pass that on to the corresponding location in the corresponding 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.*

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 18.11). 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.

18.4 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 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.

18.4.1 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.

18.4.2 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: 18.1 *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

$$\mathbf{z}_i = \Lambda^{(-1/2)}\mathcal{U}^T(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\})).$$

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 18.1. 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 57, but I reproduce the useful facts box here as a reminder. Notice that, by using the rule

$$\mathbf{z}_i = \Lambda^{(-1/2)}\mathcal{U}^T(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\})),$$

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

$$\mathbf{z}_i = \mathcal{U}(\Lambda + \epsilon\mathcal{I})^{(-1/2)}\mathcal{U}^T(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\}))$$

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.

18.4.3 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.

18.4.4 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 $\beta = \mathbf{m}$. 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

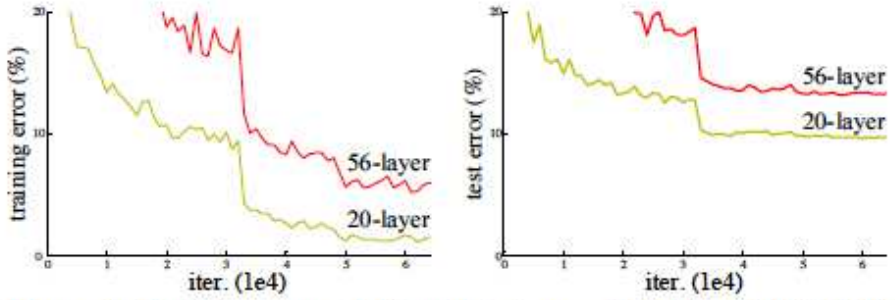


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

FIGURE 18.12: *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.*

$\mathbf{m} = \mathbf{0}$ 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.

18.4.5 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 18.12). One recent strategy for avoiding this difficulty is to build a **residual layer**. Figure 18.13 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

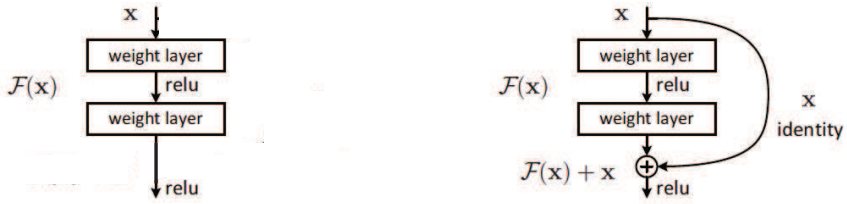


Figure 2. Residual learning: a building block.

FIGURE 18.13: *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.*

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 18.13), 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 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

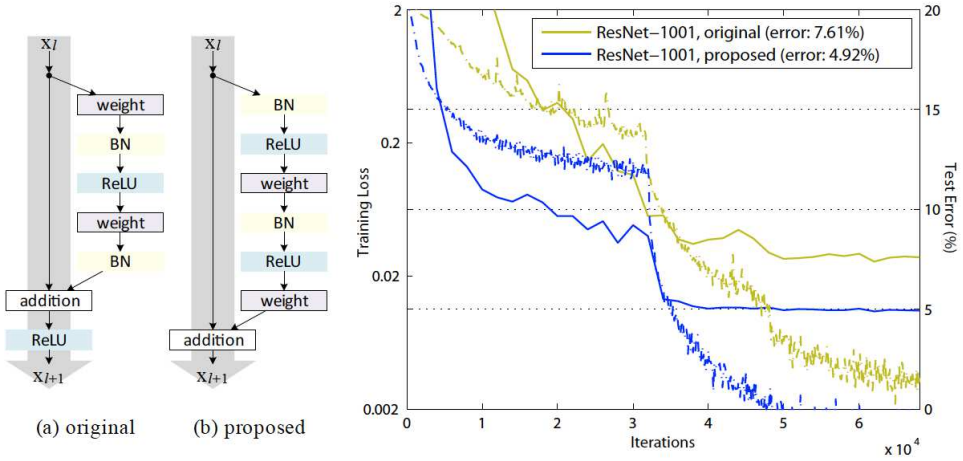


FIGURE 18.14: *This figure is revised from* Identity Mappings in Deep Residual Networks *by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (which you can find on ArXiV using a search engine). On the* **left**, *(a) shows the original residual network architecture, also described in Figure 18.13; (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.*

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.

18.5 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

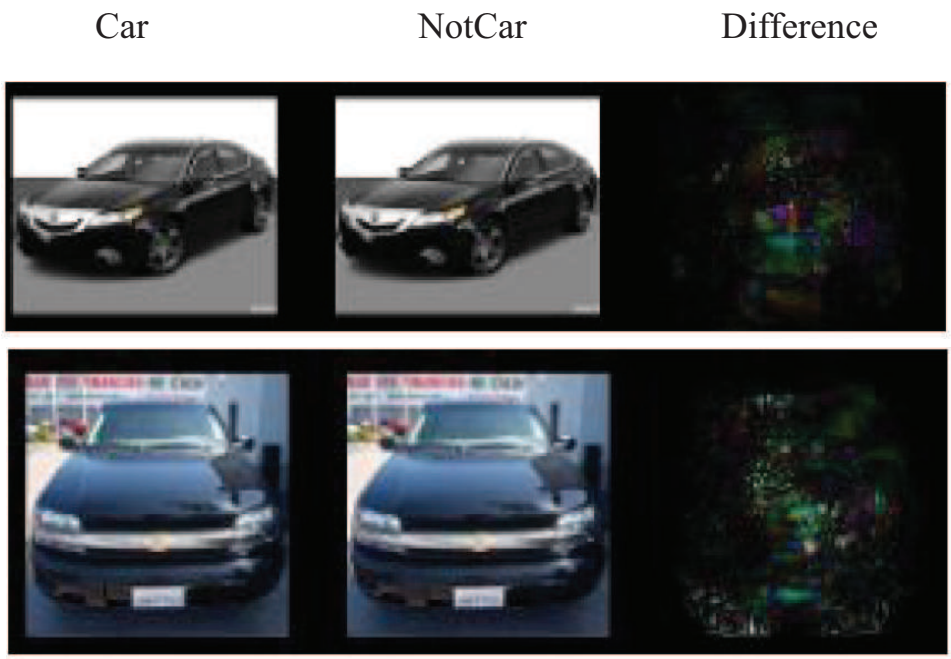


FIGURE 18.15: *This figure is from "Intriguing properties of neural networks", by Christian Szegedy* et al.*. The* **left** *column shows images classified as* `car`*; the* **middle** *column shows modified versions of those images classified as* `not car`*; and the* **right** *column shows the difference.*

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 18.15 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 18.16 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 18.17). The existence of adversarial examples raises the following, rather 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

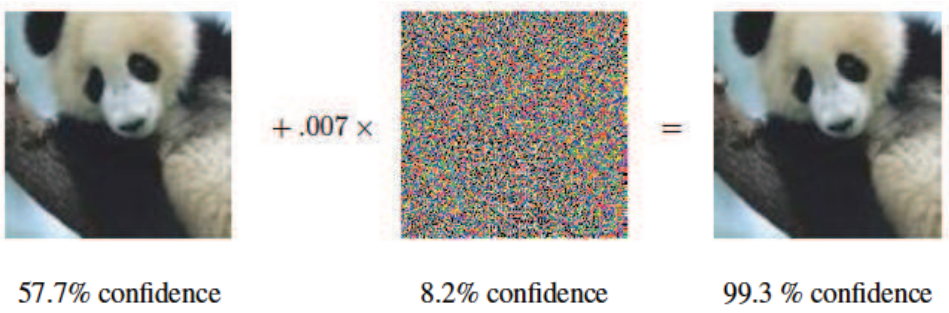


FIGURE 18.16: *This figure is from "EXPLAINING AND HARNESSING ADVERSARIAL EXAMPLES", by Ian Goodfellow* et al.. *The* **left** *column shows a* `panda`*; the* **middle** *column shows a modification (which has been exaggerated to make it non-grey; note the multiplier); and the* **right** *column shows a* `nematode`*.*

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.

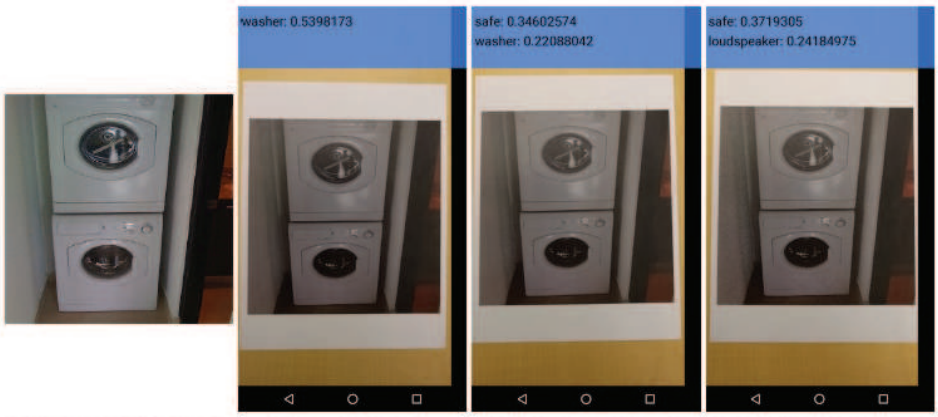


FIGURE 18.17: *This figure is from "ADVERSARIAL EXAMPLES IN THE PHYSICAL WORLD", by Alexey Kurakin* et al.*. The authors photographed the washing machine on the* **left**. *They then prepared clean and adversarial versions of that image and printed them. Finally, they photographed them with a cell-phone 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`*).*

18.6 YOU SHOULD

18.6.1 remember these definitions:

18.6.2 remember these terms:

18.6.3 remember these facts:

18.6.4 remember these procedures:

18.6.5 be able to:

-

CHAPTER 19

More Neural Networks

19.1 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 57). 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).

19.1.1 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.

19.1.2 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

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}.$$

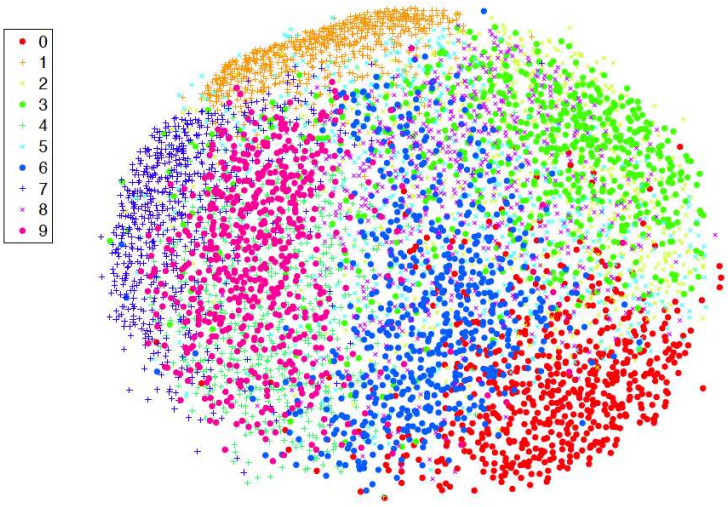


FIGURE 19.1: *A Sammon mapping of 6,000 samples of a 1,024 dimensional data set. The data was reduced to 30 dimensions using PCA, then subjected to a Sammon mapping. This data is a set of 6, 000 samples from the MNIST dataset, consisting of a collection of handwritten digits which are divided into 10 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 57. 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 57 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.

19.2 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

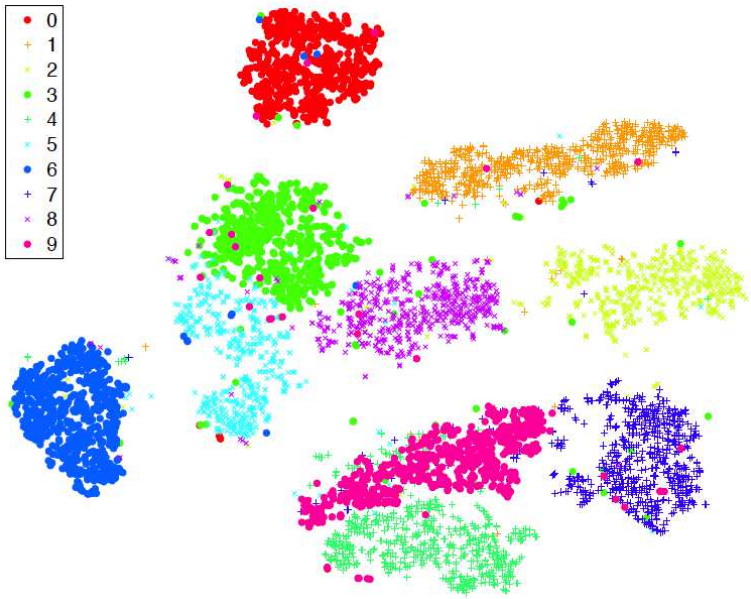


FIGURE 19.2: *A t-sne mapping of 6,000 samples of a 1,024 dimensional data set. The data was reduced to 30 dimensions using PCA, then subjected to a t-sne mapping. This data is a set of 6, 000 samples from the MNIST dataset, consisting of a collection of handwritten digits which are divided into 10 classes* $(0, \ldots 9)$*. 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.

19.2.1 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.

19.2.2 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

$$\left\|\mathcal{D}(Z_i, \theta_d) - \mathcal{I}_i\right\|^2 \text{ where } Z_i = \mathcal{E}(\text{noise}(\mathcal{I}_i), \theta_e)$$

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 57. 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.

19.2.3 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

$$\begin{aligned} \left\|\mathcal{D}(Z_{i2}, \theta_{d2}) - Z_{i1}\right\|^2 \quad & \text{where} \quad Z_{i2} = \mathcal{E}(\text{noise}(Z_{i1}), \theta_{e2}) \\ & \text{and} \quad Z_{i1} = \mathcal{E}(\mathcal{I}_1, \theta_{e1}) \end{aligned}$$

Again, training is by stochastic gradient descent using one of the tricks of section 57.

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

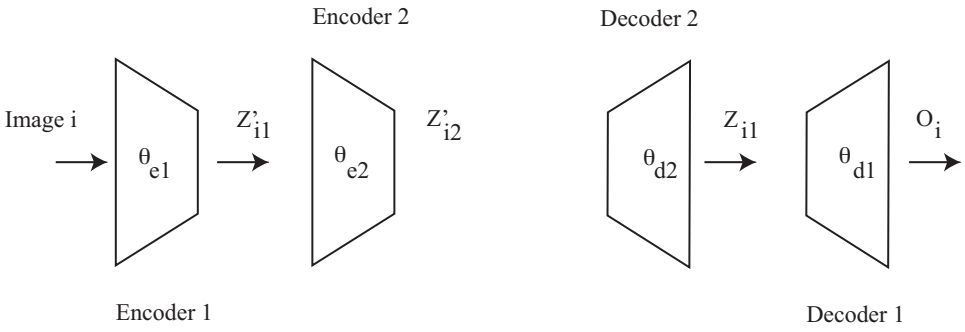


FIGURE 19.3: *Two layers of denoising auto-encoder, ready for fine tuning. This figure should help with the notation in the text.*

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

$$\begin{aligned} \|\mathcal{D}(Z_{i1}, \theta_{d1}) - I_i\|^2 \quad & \text{where} \quad Z_{i1} = \mathcal{D}((Z'_{i2}), \theta_{d2}) \\ & \text{and} \quad Z'_{i2} = \mathcal{E}(Z'_{i1}, \theta_{e2}) \\ & \text{and} \quad Z'_{i1} = \mathcal{E}(\mathcal{I}_i, \theta_{e1}) \end{aligned}$$

(Figure 19.3 might be helpful here).

19.2.4 Current practice with autoencoders

As training methods have become better, it is less usual to use the stacking procedure described above. One can build quite effective autoencoders with a set of convolutional layers, using stride to make the blocks smaller. This is the encoder, and its output is the code. The decoder is a set of convolutional layers using stride to make the blocks larger (Figure 57). This structure is often referred to as an **hourglass network** (by gross visual analogy!).

It remains tricky to get really nice images from the decoder. We will discuss some of the tricks later, but using sum of squared errors as a reconstruction loss tends to produce somewhat blurry images (eg Figure 19.4). This is because the square of a small number is very small. As a result, the sum of squared error loss tends to prefer errors that are small, but somewhat widely distributed. At an edge in an image, an error like this will tend to result in a smoothed edge (Figure **??**).

Some extreme training tricks are possible, and sometimes justified. Figure 57 illustrates an autoencoder trained to fill in large blocks of an image (an **inpainting autoencoder**). This can work for, say, faces, because the missing piece of face can be predicted moderately well from the remaining face.

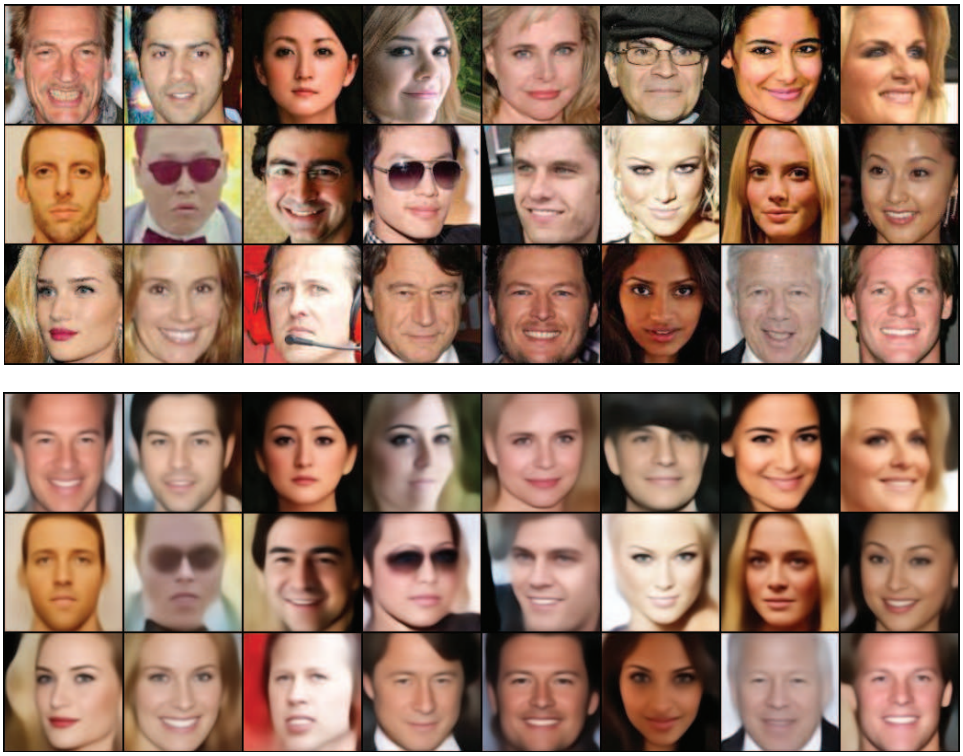


FIGURE 19.4: **Top:** *shows three batches of face images from the widely used Celeb-A dataset, which you can find at* http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html. **Bottom:** *shows the output of a simple autoencoder on these images. You should notice that the autoencoder does not preserve high spatial frequency details (the faces have been slightly blurred), but that it mostly reproduces the faces rather well. It is traditional to evaluate image autoencoders on face datasets, because mild blurring often makes a face look more attractive. Figure courtesy of Anand Bhattad, of UIUC.*

19.2.5 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.

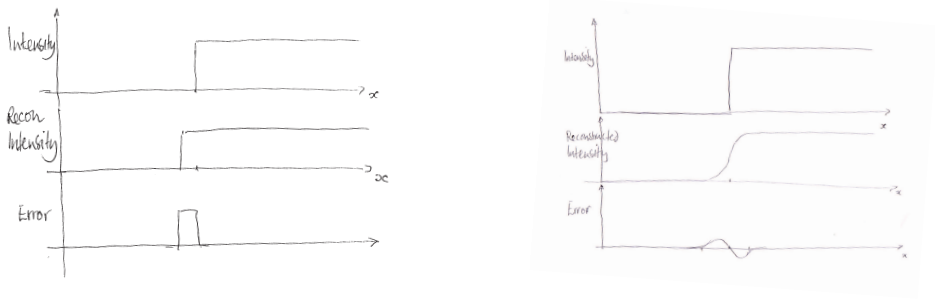


FIGURE 19.5: *On the* **left**, *a stylized image edge with one possible reconstruction error. The top graph shows the image intensity along some line (x is distance along the line); the middle graph shows one possible reconstruction, with the edge correctly reproduced but in the wrong place; and the lower graph shows the resulting error. This error will have a large sum of squares, because it consists of large values. On the* **right**, *a stylized image edge with a different reconstruction error, which makes the edge blurry. Notice how the error is small and spread out; as a result, the sum of squared errors is small. We can safely assume that an autoencoder will make some kind of error. This argument suggests that autoencoders trained with a sum of squared error loss will quite strongly prefer to make errors that result in rather blurry images.*

19.3 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

19.3.1 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

FIGURE 19.6: *Three batches of face images from the widely used Celeb-A dataset, which you can find at* http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html*, with black boxes over the faces where the inpainting autoencoder is required to reconstruct the image without seeing it. Results in figure 19.7. Figure courtesy of Anand Bhattad, of UIUC.*

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.

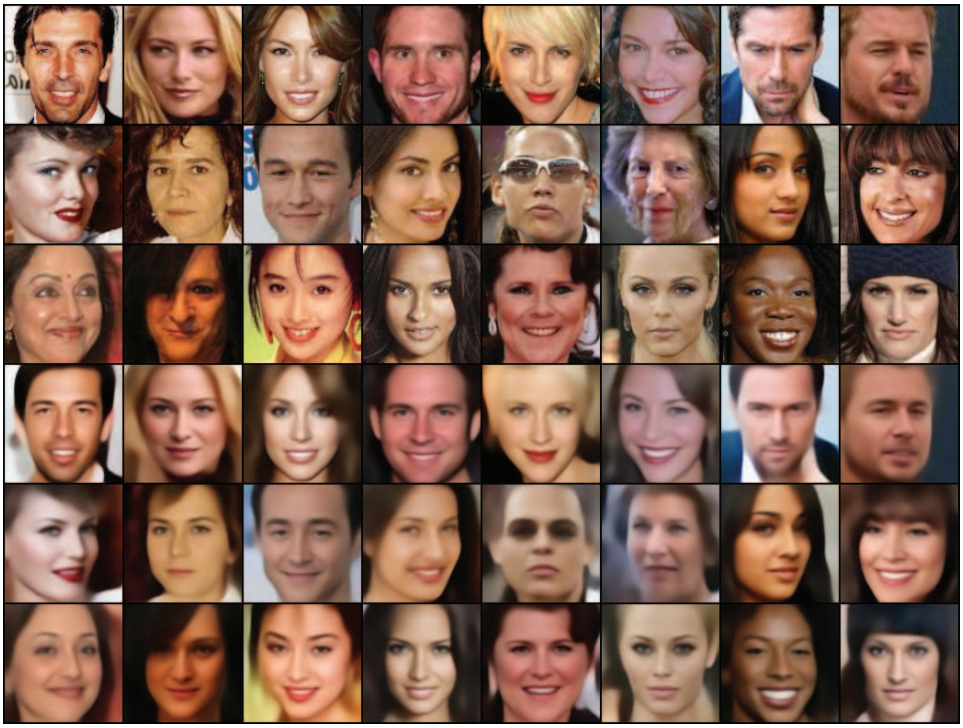


FIGURE 19.7: **Top:** *shows three batches of face images from the widely used Celeb-A dataset, which you can find at* http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html. **Bottom:** *shows the output of an inpainting autoencoder on these images. You should notice that the autoencoder does not preserve high spatial frequency details (the faces have been slightly blurred), but that it mostly reproduces the faces rather well. The inputs are in figure 19.6; notice there are large blocks of face missing, which the autoencoder is perfectly capable of supplying. Figure courtesy of Anand Bhattad, of UIUC.*

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.

19.3.2 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 57 and 57. 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

$$\begin{aligned}\mathbb{D}(Q(z|X) \| P(z|X)) &= \mathbb{E}_Q[\log Q(z|X) - \log P(z|X)] \\ &= \mathbb{E}_Q[\log Q(z|X)] - \mathbb{E}_Q[\log P(X|z) + \log P(z) - \log P(X)] \\ &= \mathbb{E}_Q[\log Q(z|X)] - \mathbb{E}_Q[\log P(X|z) + \log P(z)] + \log P(X))\end{aligned}$$

where the last line works because $\log P(X)$ doesn't depend on z. Recall the definition of the variational free energy from chapter 57. Write

$$\mathsf{F}_Q = \mathbb{E}_Q[\log Q] - \mathbb{E}_Q[\log P(X|z) + \log P(z)]$$

and so we have

$$\log P(X) - \mathbb{D}(Q(z|X) \| P(z|X)) = -\mathsf{F}_Q.$$

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 $\mathbb{D}(Q(z|X) \| P(z|X)) \geq 0$. 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 $-\mathsf{F}_Q$.

19.3.3 Turning the VFE into a Loss

The best case occurs when $Q(z|X) = P(z|X)$ (because then $\mathbb{D}(Q(z|X) \| P(z|X)) = 0$, 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

$$\begin{aligned} \log P(X) &= \sum_i \log P(X_i) \\ &= \mathbb{D}(Q(z|X) \| P(z|X)) - \mathsf{E}_{Q(z|X)} \\ &= \sum_i \left[\mathbb{D}(Q(z|X_i) \| P(z|X_i)) - \mathsf{E}_{Q(z|X_i)}\right]. \end{aligned}$$

It's usual to train networks to minimize losses. We can write the loss as

$$\begin{aligned} \mathsf{E}_Q &= -\mathbb{E}_Q[\log Q] + \mathbb{E}_Q[\log P(X|z) + \log P(z)] \\ &= \mathbb{D}(Q(z|X) \| P(z)) - \mathbb{E}_Q[\log P(X|z)] \\ &= \sum_i \left[\mathbb{D}(Q(z|X_i) \| P(z)) - \mathbb{E}_{Q(z|X_i)}[\log P(X_i|z)]\right]. \end{aligned}$$

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

$$\begin{aligned}
\mathsf{E}_Q &= \sum_i \left[\mathbb{D}(Q(z|X_i) \| P(z)) - \mathbb{E}_{Q(z|X_i)}[\log P(X_i|z)]\right] \\
&= \sum_i \left[\begin{array}{c} \frac{1}{2}\left(\begin{array}{c} \mathsf{Tr}\left(\Sigma(X_i;\theta_\Sigma)\right) + \mu(X_i;\theta_\mu)^T\mu(X_i;\theta_\mu) \\ -k - \log\left(\mathsf{Det}\left(\Sigma(X_i;\theta_\Sigma)\right)\right) \end{array} \right) \\ -\mathbb{E}_{Q(z|X_i)}\left[\frac{-\left(\|X_i - \mathrm{D}(z;\theta_D)\|^2\right)}{2}\right] \end{array} \right].
\end{aligned}$$

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

$$\begin{aligned}
\mathsf{E}_Q &= \sum_i \left[\mathbb{D}(Q(z|X_i) \| P(z)) - \mathbb{E}_{Q(z|X_i)}[\log P(X_i|z)]\right] \\
&\approx \sum_i \left[\begin{array}{c} \frac{1}{2}\left(\begin{array}{c} \mathsf{Tr}\left(\Sigma(X_i;\theta_\Sigma)\right) + \mu(X_i;\theta_\mu)^T\mu(X_i;\theta_\mu) - k \\ -\log\left(\mathsf{Det}\left(\Sigma(X_i;\theta_\Sigma)\right)\right) \end{array} \right) \\ -\frac{-\left(\|X_i - \mathrm{D}(\mu(X_i;\theta_\mu)+\Sigma(X_i;\theta_\Sigma)^{1/2}\mathbf{u}_i;\theta_D)\|^2\right)}{2} \end{array} \right].
\end{aligned}$$

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.

19.3.4 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...

19.4 GENERATIVE ADVERSARIAL NETWORKS (GANS)

Here is a strategy for generating specialized images, for example, images of faces. Construct a decoder. Feed it with a stream of random codes, drawn as IID samples from some convenient distribution. Now train the decoder by requiring that (a) the outputs are "like" images of faces and (b) any image of a face "could be" an output. In this scheme, the decoder is usually called a **generator**, and networks trained with these requirements are usually called **generative adversarial networks** or **GAN**s. Actually imposing these requirements involves important technical difficulties, and I describe two strategies below.

19.4.1 Using a Discriminator

We know how to build classifiers, so we could require that any image made by the generator fools a classifier that is trained to tell the difference between synthetic face images and real ones. In this scheme, the classifier is usually called a **discriminator**.

Write $G(\mathbf{z})$ for an image generated from a code $\mathbf{z}$; write $D(\mathbf{x})$ for the discriminator applied to some image $\mathbf{x}$. We assume the discriminator produces a number between 0 and 1, and we would like it to produce a 1 for any real image, and a 0

for any synthetic image. Now consider the cost function

$$\mathcal{C}(D,G) = \frac{1}{N_r} \sum_{\mathbf{x}_i \in \text{real images}} \log\left(D(\mathbf{x}_i)\right) + \frac{1}{N_s} \sum_{\mathbf{z}_j \in \text{codes}} \log\left(1 - D(G(\mathbf{z}_j))\right).$$

If the discriminator works very well (i.e. can tell the difference between real and synthetic images) this will be large. If the generator works very well (i.e. can fool the discriminator), the cost will be small. So we could try and find $\hat{D}$ and $\hat{G}$ that are obtained as

$$\underset{G}{\text{argmin}} \quad \underset{D}{\text{argmax}} \; \mathcal{C}(D,G).$$

Here G would be some form of decoder, and D would be some form of classifier. It seems natural to try using stochastic gradient descent/ascent on this problem. One scheme is to repeatedly: fix the G, and take some uphill steps in D; now fix D, and take some downhill steps in G. This is where the term *adversarial* comes from: the generator and the discriminator are adversaries, trying to beat one another in a game.

This apparently simple scheme is fraught with practical and technical difficulties. Here is one important difficulty. Imagine G isn't quite right, but D is perfect, and so reports 1 for every possible true image and 0 for every possible synthetic image. This means that we may want a D that isn't very good. Then there is no gradient to train G, because any small update of G will still produce images that aren't quite right. In fact, we are requiring that D has an important property: if you make an image "more real", then D will produce a larger value, and if you make it "less real", D will produce a smaller value. This is a much more demanding requirement than requiring D is a classifier.

Here is a second difficulty. Imagine there are two clusters of faces that are quite different. I will use "glasses" and "no glasses" as an example. In principle, if the generator does not produce "glasses" faces, then the discriminator has an easier job (any face with "glasses" can be classified as real). But there may be no way for the generator to use this information to produce "glasses" faces, because there aren't easy intermediates between "glasses" faces and "no glasses" faces.

Despite these caveats, it has been possible to train networks like this. There is good evidence that they are capable of producing rather good images (Figure 19.8), if the contents are specialized (i.e. one can produce images of faces, of rooms, or of lungs as below, but not some generic image of anything). There is also good evidence that the general idea of an adversarial loss can be used to tune other generators rather well. For example, efforts to improve VAE-like networks or autoencoders by imposing an adversarial loss are often successful. The discriminator can easily spot that real images aren't fuzzy; and the caveats above are mitigated by the use of other losses to ensure the generator starts in about the right place.

19.4.2 Comparing Distributions

Here is an alternative view of our training requirements. You can see the generated images as samples of a probability distribution $P(R)$ (R for reconstruct). The true images are samples of another probability distribution, $P(X)$. We would like to adjust the generator so that $P(R)$ is "the same" as $P(X)$.

FIGURE 19.8: *Three batches of face images generated by a variant of the GAN strategy described in section* **??**, *using the Celeb-A dataset (which you can find at http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html) as training data. You should notice that these images really look like faces "at a glance", but if you attend you'll see various slightly creepy eyes, small global distortions of the face shape, odd mouth shapes, and the like. Figure courtesy of Anand Bhattad, of UIUC.*

One way to test whether two distributions are "the same" is to look at expectations. For example, think about two probability distributions $P(x)$ and $Q(x)$ on the closed interval from 0 to 1. Choose a sufficiently large set of functions ϕ_k, indexed by k. As a concrete example, you could think of the monomials, where $\phi_0 = 1$, $\phi_1 = x$, $\phi_2 = x^2$, and so on. Now assume $\mathbb{E}_{\phi_k}[P(x)] = \mathbb{E}_{\phi_k}[Q(x)]$ for all of these functions. This implies that, for any other function $f(x)$, $\mathbb{E}_f[P(x)]$ must be arbitrarily close to $\mathbb{E}_f[Q(x)]$. This is because you can represent $f(x)$ with a series to arbitrary precision, so that

$$f(x) = a_o\phi_o(x) + a_1\phi_1(x) + \ldots + \text{arbitrarily small error.}$$

In turn, $P(x)$ and $Q(x)$ are "the same" for all practical purposes. If you've seen a lot of formal analysis and probability, you'll notice that I've fudged on some details here, but you'll be able to fill them in.

This all suggests the following strategy. Come up with a collection of test functions ϕ_k. Choose the generator to force

$$\sum_k \left[\frac{1}{N_r} \sum_{\mathbf{x}_i \in \text{real images}} \phi_k(\mathbf{x}_i) - \frac{1}{N_s} \sum_{\mathbf{z}_j \in \text{codes}} \phi_k(G(\mathbf{z}_j)) \right]^2$$

to be small. There are difficulties here, too. First, the collection of test functions might need to be very large, creating problems with gradients and the like. Second, these test functions will need to be "useful" in some reasonable way. So, for example, a test function that extracts the value of a single pixel is unlikely to be much help. Again, it is possible to overcome these problems, but one must use kernel methods, which are outside our scope.

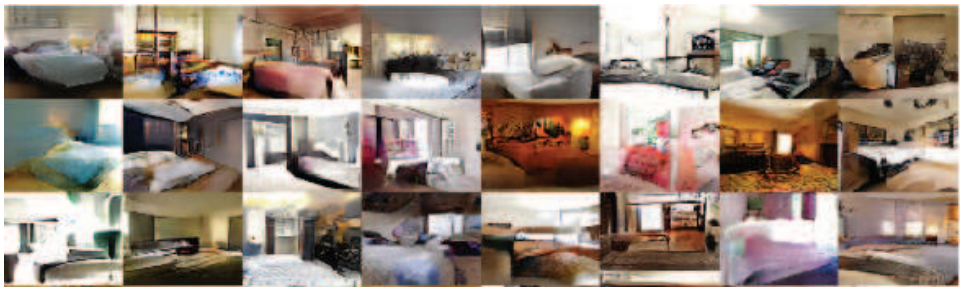


FIGURE 19.9: *Images of rooms generated by a GAN using the optimal transportation theory method sketched in the text. Figure courtesy of Ishan Deshpande and Alex Schwing, of UIUC.*

An alternative method for comparing distributions reasons about nearby points. We will think about two sets of samples $\{R_i\}$ and $\{X_j\}$ in one dimension. For simplicity, we will reason about sample sets that have the same size. If these samples come from the same distribution, there should be an X close to any R, and an R close to any X. In particular, a reasonable measure of similarity is to pair X's with R's, then sum the distance between pairs. We choose pairs so that: each X (resp. R) has exactly one R (resp. X); and the sum of the distances is minimized. It turns out that one can evaluate this particular distance in an easy way. Sort the R_i in descending order; sort the X_j in descending order; then obtain the pairs by pairing the first R_i with the first X_j, the second with the second, and so on. We now sum the squared distances between pairs.

This trick extends to multiple dimensions in a simple way. Assume we have high dimensional R_i (resp. X_j). Now choose some random direction in this high dimensional space, and project the R_i (resp. X_j) onto that direction. If the distributions are the same, the projected distributions are the same. So we should obtain a "small" value of the sum for that – and any – projection. In turn, this justifies averaging the distances over many random projections.

One can use this (sketched!) line of reasoning to construct a distance between sample sets that have different numbers of samples in them, and in multiple dimensions. The general theory is known as **optimal transportation theory** or **Monge-Kantorovich theory** depending on taste, and is rather off our path. However, the line of reasoning leads to rather good generative models, as figures 57 suggest.

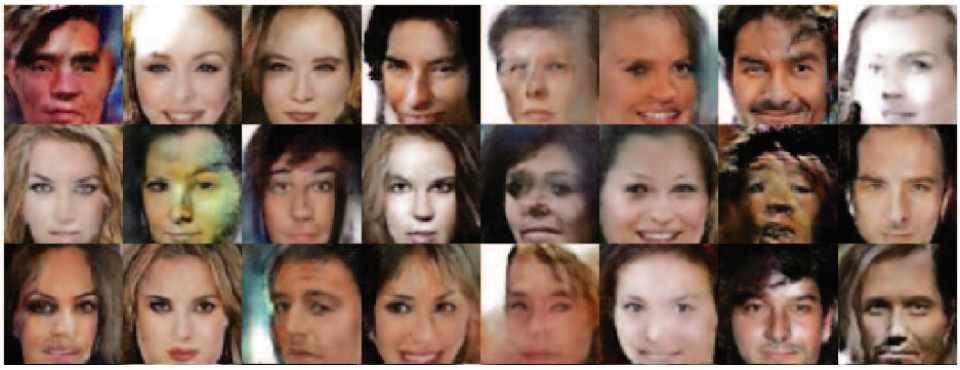


FIGURE 19.10: *Images of faces generated by a GAN using the optimal transportation theory method sketched in the text. Figure courtesy of Ishan Deshpande and Alex Schwing, of UIUC.*

19.5 YOU SHOULD

19.5.1 remember these definitions:

19.5.2 remember these terms:

19.5.3 remember these facts:

19.5.4 remember these procedures:

19.5.5 be able to:

-

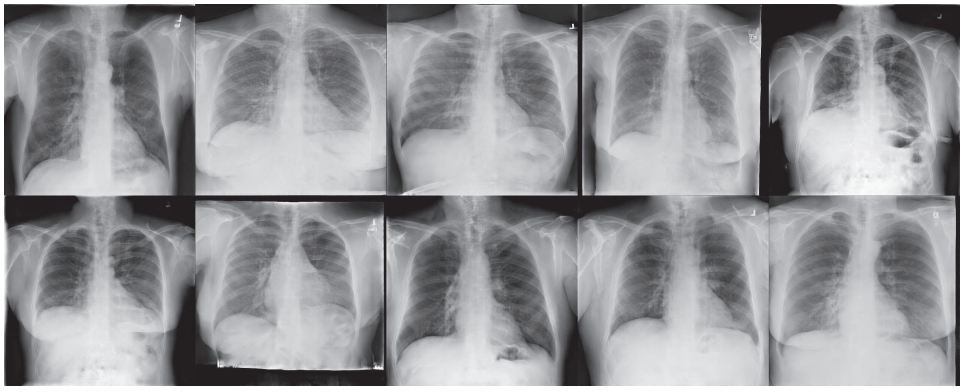


FIGURE 19.11: *Images of chest X-Rays generated by a GAN using the optimal transportation theory method sketched in the text. Figure courtesy of Ishan Deshpande and Alex Schwing, of UIUC. I can show these images* because they're not medical images of real humans - *they were made by a computer program!*

PART SEVEN

BOOSTING

CHAPTER 20

Boosting

The following idea may have occurred to you after reading the chapter on classification. Imagine you have trained a classifier. You could try to 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 to 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.

20.1 GREEDY AND STAGEWISE METHODS

20.1.1 Example: Greedy Stagewise Linear Regression

We wish to build a linear regression of y against some high dimensional vector $\mathbf{x}$. Using the notation of chapter 57, we will need to solve

$$\mathcal{X}^T\mathcal{X}\beta = \mathcal{X}^T\mathbf{y}$$

but this might be hard to do if $\mathcal{X}$ was really big. You're unlikely to see many problems where this really occurs, because modern software and hardware are very efficient at dealing with even enormous linear algebra problems.

However, thinking about this case is very helpful. What we could do is choose some subset of the features to work with, to obtain a smaller problem. We will do

this repeatedly, so we need some notation. Write $\mathbf{x}^{(i)}$ for the i'th subset of features. For the moment, we will assume this is a small set of features and worry about how to choose the set later. Write $\mathcal{X}^{(i)}$ for the matrix constructed out of these features, etc. Now we regress $\mathbf{y}$ against $\mathcal{X}^{(1)}$. This chooses the $\hat{\beta}^{(1)}$ that minimizes the squared length of the residual vector

$$\mathbf{e}^{(1)} = \mathbf{y} - \mathcal{X}^{(1)}\hat{\beta}^{(1)}.$$

We obtain this $\hat{\beta}^{(1)}$ by solving

$$\left(\mathcal{X}^{(1)}\right)^T \mathcal{X}^{(1)}\hat{\beta}^{(1)} = \left(\mathcal{X}^{(1)}\right)^T \mathbf{y}.$$

Now assume we regress the residual vector $\mathbf{e}^{(1)}$ against a new set of features, $\mathcal{X}^{(2)}$. Doing so will choose the $\hat{\beta}^{(2)}$ that minimizes the squared length of the new residual vector

$$\mathbf{e}^{(2)} = \mathbf{e}^{(1)} - \mathcal{X}^{(2)}\hat{\beta}^{(2)}$$

We obtain this $\hat{\beta}^{(1)}$ by solving

$$\left(\mathcal{X}^{(2)}\right)^T \mathcal{X}^{(2)}\hat{\beta}^{(2)} = \left(\mathcal{X}^{(2)}\right)^T \mathbf{e}^{(1)}.$$

So far, this is all pretty routine. But notice that

$$\mathbf{e}^{(2)} = \mathbf{y} - \mathcal{X}^{(1)}\hat{\beta}^{(1)} - \mathcal{X}^{(2)}\hat{\beta}^{(2)}.$$

Because our choice of $\hat{\beta}^{(2)}$ minimizes the squared length of this vector, we have that

$$\mathbf{e}^{(2)^T}\mathbf{e}^{(2)} \leq \mathbf{e}^{(1)^T}\mathbf{e}^{(1)}$$

with equality only if $\mathcal{X}^{(2)}\hat{\beta}^{(2)} = \mathbf{0}$. In turn, the second round did not make the residual worse. If the features in $\mathcal{X}^{(2)}$ aren't all the same as those in $\mathcal{X}^{(1)}$, it is very likely to have made the residual better.

Extending all this to an R'th round is just a matter of notation; you can write an iteration with $\mathbf{e}^{(0)} = \mathbf{y}$. Then you regress $\mathbf{e}^{(i-1)}$ against the features in $\mathcal{X}^{(i)}$ to get $\hat{\beta}^{(i)}$, and

$$\mathbf{e}^{(i)} = \mathbf{e}^{(i-1)} - \mathcal{X}^{(i)}\hat{\beta}^{(i)} = \mathbf{e}^{(0)} - \sum_{u=1}^{i} \mathcal{X}^{(u)}\hat{\beta}^{(u)}.$$

The residual never gets bigger (at least if your arithmetic is exact). This procedure is referred to as **greedy stagewise linear regression**. It's stagewise, because we build up the model in steps. It's greedy, because we do not adjust our estimate of $\hat{\beta}^{(1)}$ when we compute $\hat{\beta}^{(2)}$, etc.

This process won't work for a linear regression when we use all the features in $\mathcal{X}^{(1)}$. 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 57), and this is a global minimizer. So to minimize

$$\left(\left[\mathbf{y} - \mathcal{X}\hat{\beta}\right] - \mathcal{X}\gamma\right)^T \left(\left[\mathbf{y} - \mathcal{X}\hat{\beta}\right] - \mathcal{X}\gamma\right)$$

by choice of γ, we'd have to have $\mathcal{X}\gamma = 0$, meaning that the residual wouldn't improve. At this point, greedy stagewise linear regression may look like a method of getting otherwise unruly linear algebra under control. But it's also a model recipe, exposed in the box below. As we shall see, this recipe admits substantial generalization.

Procedure: 20.1 *Greedy stagewise linear regression*

We choose to minimize the squared length of the residual vector, so write $\mathcal{E}(\mathbf{r}^{(i-1)} - \mathcal{X}^{(i)}\hat{\beta}^{(i)}) = \left|(\mathbf{r}^{(i-1)} - \mathcal{X}^{(i)}\hat{\beta}^{(i)})\right|^2$. Write $\mathbf{r}^{(0)} = \mathbf{y}$. Now iterate:

- choose a set of features to form $\mathcal{X}^{(i)}$;
- construct $\hat{\beta}^{(i)}$ by minimizing $\mathcal{E}(\mathbf{r}^{(i-1)} - \mathcal{X}^{(i)}\hat{\beta}^{(i)})$. Do this by solving the linear system $\left(\mathcal{X}^{(i)}\right)^T \mathcal{X}^{(i)}\hat{\beta}^{(i)} = \left(\mathcal{X}^{(i)}\right)^T \mathbf{r}^{(i-1)}$.
- form $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \mathcal{X}^{(i)}\hat{\beta}^{(i)})$.

20.1.2 Regression Trees

We wish to build a regression of y against some high dimensional vector $\mathbf{x}$, and we believe that a linear regression won't work. 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. For the simplest regression tree, each leaf contains a single value representing the value the predictor takes in that cell (one can place other prediction methods in the leaves; we won't bother). The splitting process parallels the one we used for classification, but now we can use the error in the regression to choose the split instead of the information gain.

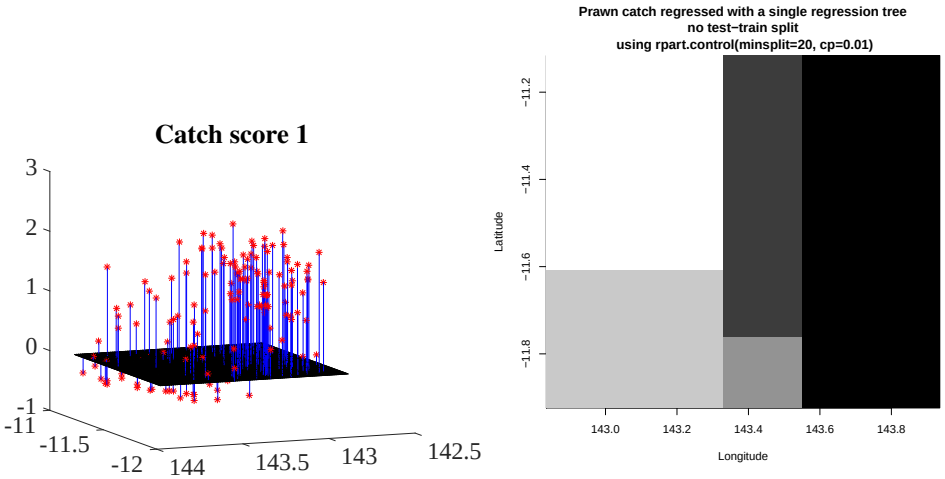


FIGURE 20.1: *On the* **left**, *a 3D scatter plot of score 1 of the prawn trawls data from http://www.statsci.org/data/oz/reef.html, plotted as a function of latitude and longitude. On the* **right**, *a regression using a single regression tree, to help visualize the kind of predictor these trees produce. You can see what the tree does: carve space into boxes, then predict a constant inside each.*

Worked example 20.1 *Regressing prawn scores against location*

Build a regression tree predicting prawn score 1 (whatever that is!) against latitude and longitude using the prawn trawls dataset from http://www.statsci.org/data/oz/reef.html.

Solution: We will use this data set several times, because it is easy to visualize interesting predictors. Figure **??** shows a 3D scatter plot of score 1 against latitude and longitude. There are good packages for building such trees (I used R's `rpart`). Figure **??** shows a regression tree fitted with that package, as an image. This makes it easy to visualize the function. The darkest points are the smallest values, and the lightest points are the largest. You can see what the tree does: carve space into boxes, then predict a constant inside each.

20.1.3 Greedy Stagewise Regression with Trees

The recipe for greedy stagewise linear regression applies to regression trees as well, with very little change. We regress y against $\mathbf{x}$; construct the residuals; and regress the residuals against $\mathbf{x}$. Of course, this could be repeated, perhaps indefinitely. It is helpful to change notation. Write $f(\mathbf{x};\theta_k)$ for a regression tree that accepts $\mathbf{x}$ and produces a prediction (here θ_k are parameters internal to the tree; where to split; what is in the leaves; and so on). Once all trees have been chosen, we can

write the regression as

$$F(\mathbf{x}) = \sum_k f(\mathbf{x}; \theta_k)$$

where there might be quite a lot of trees 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 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 using the recipe for greedy stagewise linear regression. Start with a $F_0 = 0$. Write $r_i^{(n)}$ for the residual at the n'th round and the i'th example. Set $r_i^{(0)} = y_i$.

Now iterate the following step: Choose θ_n to minimize

$$\sum_i \left(r_i^{(n-1)} - f(\mathbf{x}_i; \theta_n) \right)^2$$

and then set

$$r_i^{(n)} = r_i^{(n-1)} - f(\mathbf{x}_i; \theta_n).$$

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.

We estimate θ_n using a regression tree. We regress the residual $r_i^{(n-1)}$ against $\mathbf{x}$, to get a tree $f(\mathbf{x}; \theta_n)$ that minimizes

$$\sum_i \left(r_i^{(n-1)} - f(\mathbf{x}_i; \theta_n) \right)^2.$$

Worked example 20.2 *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 **??**).

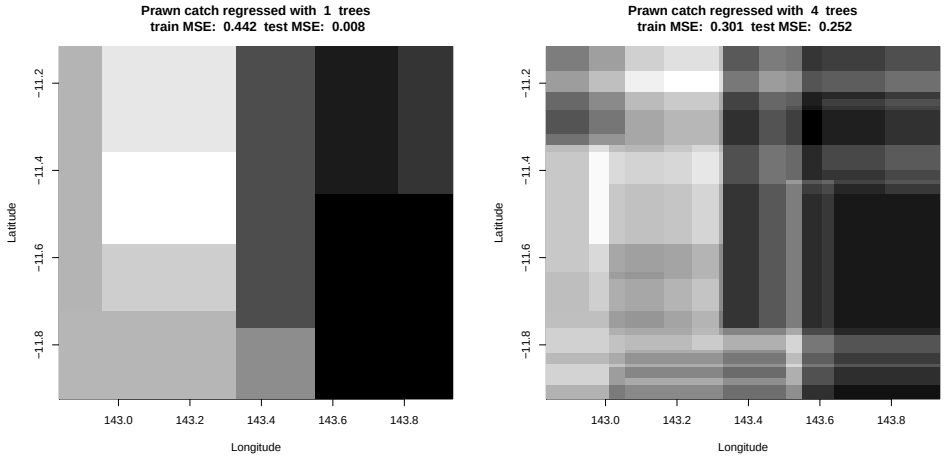


FIGURE 20.2: *Score 1 of the prawn trawls data from http://www.statsci.org/data/oz/reef.html, regressed against latitude and longitude (I did not use depth, also in that dataset; this means I could plot the regression easily). The smallest value in the data is full dark; and the largest value is full light. The figure shows results of a greedy stagewise regression using regression trees using 1 and 4 trees. The figure on the* **left** *is different from the tree of figure 20.1 because I used a test-train split in this case, and different settings for* `rpart`*. Notice that both train and test error go down, and the model gets more complex as we add trees. Further stages appear in figure 20.3, which uses the same intensity scale.*

Procedure: 20.2 *Greedy stagewise regression with regression trees*

We choose to minimize the least square error of the predictions, so write $\mathcal{E}^{(i)}(\theta) = \sum_j \left(r_j^{(i-1)} - f(x_j;\theta) \right)^2$, set $r_i^{(0)} = y_i$, and write $f(\mathbf{x};\theta)$ for a regression tree. Here θ encodes internal parameters (where to split, thresholds, and so on). We will build a regression $F(\mathbf{x};\theta,\mathbf{a}) = \sum_i f(\mathbf{x};\theta_i)$. Now iterate:

- construct $\hat{\theta}^{(i)}$ by minimizing $\mathcal{E}^{(i-1)}(\theta)$. Do this using regression tree software; expect that you will obtain an approximate minimizer.
- form $r_j^{(i)} = r_j^{(i-1)} - f(\mathbf{x}_j;\theta^{(i)})$.

None of this would be helpful if the regression using trees $1 \ldots i-1$ is worse than the regression using trees $1 \ldots i$. Here is an argument that establishes that greedy stagewise regression will make progress in the training error. Assume that,

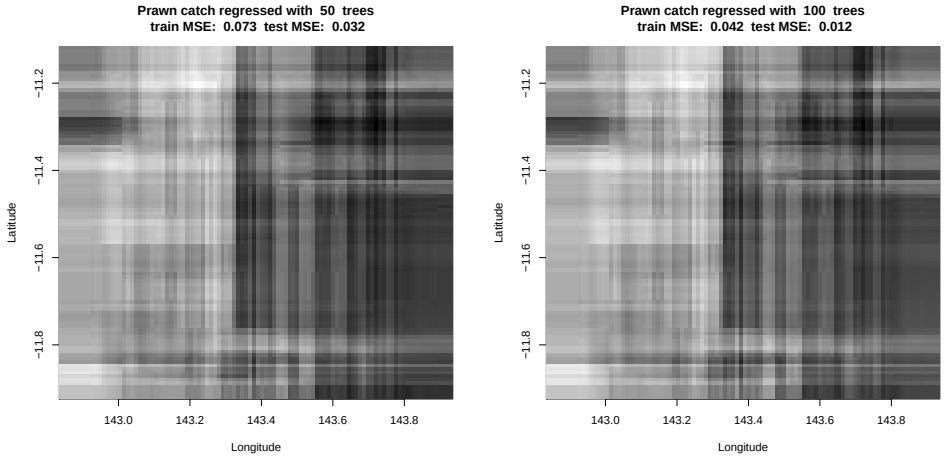


FIGURE 20.3: *Score 1 of the prawn trawls data from* http://www.statsci.org/data/oz/reef.html*, regressed against latitude and longitude (I did not use depth, also in that dataset; this means I could plot the regression easily). The smallest value in the data is full dark; and the largest value is full light. The figure shows results of a greedy stagewise regression using regression trees using 50 and 100 trees. Notice that both train and test error go down, and the model gets more complex as we add trees.*

if there is any tree that reduces the residual, the software will find one such tree; if not, it will return a tree that is a single leaf containing 0. Then $\left\|\mathbf{r}^{(i)}\right\|^2 \leq \left\|\mathbf{r}^{(i-1)}\right\|^2$, because the tree was chosen to minimize $\left\|\mathbf{r}^{(i)}\right\|^2 = \mathcal{E}(\mathbf{r}^{(i-1)} - \mathcal{X}^{(i)}\hat{\beta}^{(i)})$.

In practice, greedy stagewise regression is well-behaved. One could reasonably fear overfitting. Perhaps only the training error goes down as you add trees, but the test error will go up. This can happen, but it tends not to happen (see the examples). We can't go into the reasons here (and they have some component of mystery, anyhow).

Worked example 20.3 *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 20.4). 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.

20.2 BOOSTING A CLASSIFIER

The recipes I have given above are manifestations of a general approach. This approach applies to both regression and classification. The recipes seem more natural in the context of regression (which is why I did those versions first). But in both regression and classification we are trying to build a **predictor** – a function that accepts features and reports either a number (regression) or a label (classification). Notice we can encode the label as a number, meaning we could classify with regression machinery. In particular, we have some function $F(\mathbf{x})$, where $\mathbf{x}$. For both regression and classification, we apply F to example $\mathbf{x}$ to obtain a prediction. The regressor or classifier is learned by choosing a function that gets good behavior on a training set. This notation is at a fairly high level of abstraction (so, for example, the procedure we've used in classification where we take the sign of some function is represented by F).

20.2.1 The Loss

In early chapters, it seemed as though we used different kinds of predictor for classification and regression. But you might have noticed that the predictor used for linear support vector machines bore a strong similarity to the predictor used for linear regression, though we trained these two in quite different ways. There are many kinds of predictor – linear functions; trees; and so on. We now take the view that the kind of predictor you use is just a matter of convenience (what package

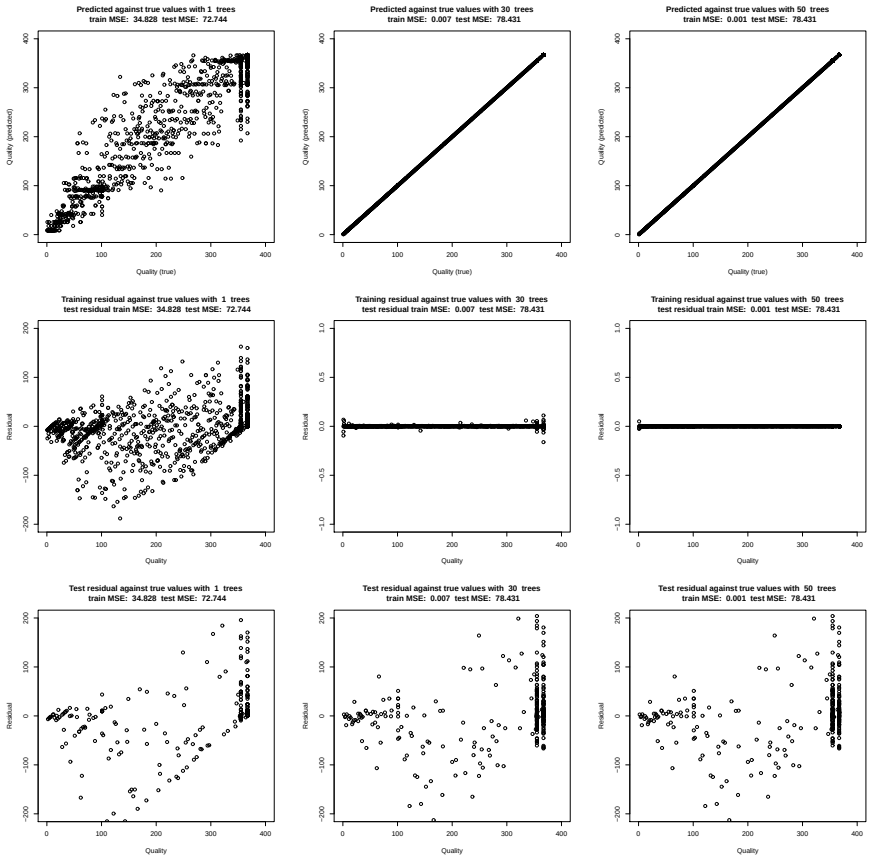


FIGURE 20.4: *Diagnostic data from a stagewise regression of rank of quality of education against all other variables except overall rank and name for the data of https://www.kaggle.com/mylesoneill/world-university-rankings/data.* **Top row:** *Model predictions against true values;* **middle row:** *residual against true values, training examples;* **bottom row:** *residuals against true values, test examples. Details in the worked example.*

you have available; what math you feel like doing; etc.). Once you know what kind of predictor you will use, you must choose the parameters of that predictor. In our view, the really important difference between classification and regression is the **loss** that you use to choose these parameters. The loss is the cost function used to evaluate errors, and so to train the predictor. Training a classifier involves using a loss that penalizes errors in class prediction in some way, and training a regressor means using a loss that penalizes prediction errors.

The **empirical loss** is the average loss on the training set. Different predictors F produce different losses at different examples, so the loss depends on the predictor F. Notice the kind of predictor isn't what's important; instead, the loss scores the difference between what a predictor produced and what it should have produced.

Now write $\mathcal{L}(F)$ for this empirical loss. There are many plausible losses that apply to different prediction problems. Here are some examples:

- For least squares regression, we minimized the least squares error:

$$\mathcal{L}_{ls}(F) = \frac{1}{N}\sum_i (y_i - F(\mathbf{x}_i))^2$$

  (though the $1/N$ term sometimes was dropped as irrelevant; section 57).

- For a linear SVM, we minimized the hinge loss:

$$\mathcal{L}_h(F) = \frac{1}{N}\sum_i \max(0, 1 - y_i F(\mathbf{x}_i))$$

  (assuming that labels are 1 or -1; section 57).

- For logistic regression, we minimized the logistic loss:

$$\mathcal{L}_{lr}(F) = \frac{1}{N}\sum_i \left[\log\left(e^{\frac{-(\hat{y}_i+1)}{2}F(\mathbf{x}_i)} + e^{\frac{1-\hat{y}_i}{2}F(\mathbf{x}_i)}\right)\right]$$

  (again, assuming that labels are 1 or -1; section 12.3.1).

We construct a loss by taking the average over the training data of a **pointwise loss** – a function ℓ that accepts three arguments: a y-value, a vector $\mathbf{x}$, and a prediction $F(\mathbf{x})$. This average is an estimate of the expected value of that pointwise loss over all data.

- For least squares regression,

$$\ell_{ls}(y, \mathbf{x}, F) = (y - F(\mathbf{x}))^2 .$$

- For a linear SVM,

$$\ell_h(y, \mathbf{x}, F) = \max(0, 1 - yF(\mathbf{x})).$$

- For logistic regression,

$$\ell_{lr}(y, \mathbf{x}, F) = \left[\log\left(e^{\frac{-(\hat{y}+1)}{2}F(\mathbf{x})} + e^{\frac{1-\hat{y}}{2}F(\mathbf{x})}\right)\right].$$

We often used a regularizer with these losses; this will not change anything important in what follows.

20.2.2 Recipe: Stagewise Reduction of Loss

Now we peek under the hood of the notation, to apply our earlier stagewise recipe. Write

$$F_m = F_m(\mathbf{x}; \theta) = \sum_{j=1}^{m} a_j f_j(\mathbf{x}; \theta_j)$$

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, yielding F_r. Whatever the particular choice of loss $\mathcal{L}$, we need to minimize

$$\frac{1}{N}\sum_i \ell(y_i, \mathbf{x}_i, F_{r-1}(\mathbf{x}_i) + a_r f_r(\mathbf{x}_i; \theta_r)).$$

For most reasonable choices of loss, we can differentiate ℓ and we write

$$\left.\frac{\partial \ell}{\partial F}\right|_{r-1,i}$$

to mean the partial derivative of that function with respect to the F argument, evaluated at the point $(y_i, \mathbf{x}_i, F_{r-1}(\mathbf{x}_i)$. Then a Taylor series gives us

$$\begin{aligned}\frac{1}{N}\sum_i \ell(y_i, \mathbf{x}_i, F_{r-1}(\mathbf{x}_i) + a_r f_r(\mathbf{x}_i; \theta_r)) \quad &\approx \quad \frac{1}{N}\sum_i \ell(y_i, \mathbf{x}_i, F_{r-1}(\mathbf{x}_i)) + \\ &\quad a_r \frac{1}{N}\sum_i \left[\left(\left.\frac{\partial \ell}{\partial F}\right|_{r-1,i}\right) f_r(\mathbf{x}_i; \theta_r)\right].\end{aligned}$$

In turn, this means that we can minimize by finding a predictor f_r such that

$$\frac{1}{N}\sum_i \left(\left.\frac{\partial \ell}{\partial F}\right|_{r-1,i}\right) f_r(\mathbf{x}_i; \theta_r)$$

is negative. We do this by choosing θ_r. This predictor should cause the loss to go down, at least for small values of a_r. Now assume we have chosen an appropriate predictor, represented by $\hat{\theta}_r$ (the estimate of the predictor's parameters). Then we can obtain a_r by minimizing

$$\Phi(a_r) = \mathcal{L}(F_{r-1}(\mathbf{x}_i) + a_r f_r(\cdot; \hat{\theta}_r))$$

which is a one-dimensional problem (remember, F_{r-1} and $\hat{\theta}_r$ are known, only a_r is unknown). 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**; I have put it in a box, below.

Procedure: 20.3 *Gradient boost*

We wish to choose a predictor F that minimizes a loss

$$\mathcal{L}(F) = \frac{1}{N}\sum_i \ell(y_i, \mathbf{x}_i, F).$$

We will do so iteratively by searching for a predictor of the form $F_r(\mathbf{x};\theta) = \sum_u \alpha_u f_u(\mathbf{x};\theta_u)$. Our search will be greedy. We start with $F_0 = 0$. Now iterate:

- form a set of weights, one per example, where

$$w_{r-1,i} = \left.\frac{\partial \ell}{\partial F}\right|_{r-1,i}$$

- choose θ_r (and so the predictor f_r) so that

$$\sum_i w_{r-1,i} f_r(\mathbf{x}_i;\theta_r)$$

  is negative;

- now form $\Phi(a_r) = \mathcal{L}(F_r + a_r f_r)$ and search for the best value of a_r using a line search method.

The important problem here is finding a predictor f_r such that

$$\frac{1}{N}\sum_i \left(\left.\frac{\partial \ell}{\partial F}\right|_{r-1,i} \right) f_r(\mathbf{x}_i;\theta_r)$$

is negative. For some predictors, this can be done in a straightforward way. For others, this problem can be rearranged into a regression problem. We will do examples of each case.

20.2.3 Weak Learners and Decision Stumps

The predictor used for a boosted classifier is often known as a **weak learner**. This name comes from the considerable body of theory covering when and how boosting should work. An important fact from that theory is that the predictor F_r needs only to be a descent direction for the loss — i.e. we need to ensure that adding some positive amount of F_r to the prediction will result in an improvement in the loss. This is a very weak constraint in the two-class classification case (it boils down to requiring that the learner can do slightly better than a 50% error rate on a weighted version of the dataset), so that it is reasonable to use quite a simple classifier for the predictor.

One very natural classifier is a **decision stump**, which tests one linear projection of the features against a threshold. The name follows, rather grossly, because this is a highly reduced decision tree. There are two common strategies. In one, the stump tests a single feature against a threshold. In the other, the stump projects the features onto some vector chosen during learning, and tests that against a threshold.

Decision stumps are useful because they're easy to learn, though not in themselves a particularly strong classifier. We have examples $(\mathbf{x}_i, y_i)$. We will assume that y_i are 1 or -1. Write $f(\mathbf{x}; \theta)$ for the stump, which will predict -1 or 1. For gradient boost, we will receive a set of weights h_i (one per example), and try to learn a decision stump that *maximizes* the sum $\sum_i h_i f(\mathbf{x}_i)$. We use a straightforward search, looking at each feature and for each, checking a set of thresholds to find the one that maximises the sum. If we seek a stump that projects features, we project the features onto a set of random directions first. The box below gives the details.

Procedure: 20.4 *Learning a decision stump*

We have examples $(\mathbf{x}_i, y_i)$. We will assume that y_i are 1 or -1, and $\mathbf{x}_i$ have dimension d. Write $f(\mathbf{x}; \theta)$ for the stump, which will predict -1 or 1. We receive a set of weights h_i (one per example), and wish to learn a decision stump that *maximizes* the sum $\sum_i h_i f(\mathbf{x}_i; \theta)$. If the dataset is too large (for your computational resources), obtain a subset by sampling uniformly at random without replacement. The parameters will be a projection, a threshold and a sign. Now for $j = 1 : d$

- Set $\mathbf{v}_j$ to be either a random d-dimensional vector *or* the j'th basis vector (i.e. all zeros, except a one in the j'th component).
- Compute $r_i = \mathbf{v}_j^T \mathbf{x}_i$.
- Sort these r's; now construct a collection of thresholds t from the sorted r's where each threshold is halfway between the sorted values.
- For each t, construct two predictors. One reports 1 if $r > t$, and -1 otherwise; the other reports -1 if $r > t$ and 1 otherwise. For each of these predictors, compute the value $\sum_i h_i f(\mathbf{x}_i; \theta)$. If this value is larger than any seen before, keep $\mathbf{v}_j$, t, and the sign of the predictor.

Now report the $\mathbf{v}_j$, t, and sign that obtained the best value.

20.2.4 Gradient Boost with Decision Stumps

We will work with two-class classification (it turns out that boosting multiclass classifiers can be tricky; more below). 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. We will label examples with 1 or -1 (it is easy to derive updates for the case when the labels are 1 or 0 from what follows). Then the exponential loss is

$$\ell_e(y, \mathbf{x}, F(\mathbf{x})) = e^{[-yF(\mathbf{x})]}.$$

Notice if $F(\mathbf{x})$ has the right sign, the loss is small; if it has the wrong sign, the loss is large.

We will use a decision stump. Decision stumps report a label (i.e. 1 or -1). Notice this doesn't mean that F_{r-1} reports only 1 or -1, because F_{r-1} is a weighted sum of predictors. Assume we know F_{r-1}, and seek a_r and f_r. We then form

$$w_{r-1,i} = -y_i e^{[-y_i F_{r-1}(\mathbf{x}_i)]}.$$

Notice there is one weight per example. The weight will have the same sign as the example's label. If F_{r-1} gets the example right, the weight will have small magnitude, and if F_{r-1} gets the example wrong, the weight will have large magnitude. We want to choose f_r so that

$$\sum_i w_{r-1,i} f_r(\mathbf{x}_i).$$

is negative. Notice that the f_r that does that will try to report the same sign as the example's label, but will concentrate on examples that F_{r-1} got very badly wrong (and so have $w_{r-1,i}$ with large magnitude). It is easy to choose a decision stump that minimizes this expression. The weights are fixed, and the stump reports either 1 or -1, so all we need to do is search for a split that achieves a minimum. You should notice that the minimum is always negative (unless all weights are zero, which can't happen). This is because you can multiply the stump's prediction by -1 and so flip the sign of the score.

Worked example 20.4 *Predicting whether a prawn catch will have above median score*

You will find a dataset giving two scores for prawn trawls, as a function of the location at which the trawling occurred, at http://www.statsci.org/data/oz/reef.html. Use a gradient boosted classifier to predict whether the first score will be greater than, or less than, the median over all points as a function of the x and y coordinates. Use a decision stump.

Solution: This isn't a particularly good use of a classifier (you'd really want to regress), but it does allow you to visualize how the predictors behave. In this case, the predictor formed by one stump cuts the region into two halves with a line; on one side of the line, it takes one constant value, and on the other, it takes a different constant value. One predictor isn't much good, but a large number of predictors give quite good results, and reasonably accurate prediction is possible. Figures 57 show the actual maps.

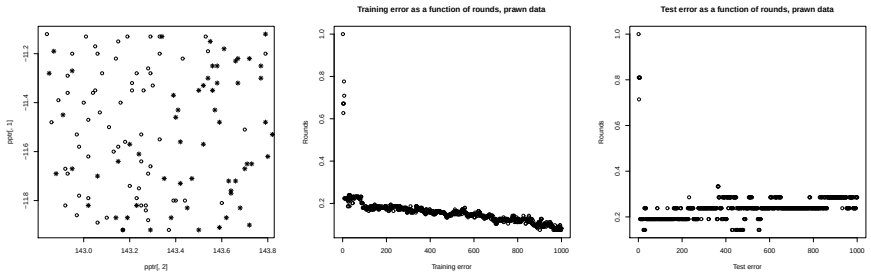


FIGURE 20.5: *On the* **left**, *a scatter plot of the prawn data. The circles are locations where the prawn catch exceeded the median; the stars are locations where the catch was below median.* **Center** *shows the training error for a boosted classifier using decision stumps, as a function of the number of stumps. Notice how adding a stump causes the training error to falls slowly but reliably even when we already have many predictors.* **Right** *shows the test error, which settles close to a fixed value fairly quickly. This isn't typical behavior – it is common to observe the test error falling reliably as well. Figure* **??** *visualizes the predictors.*

20.2.5 Gradient Boost with other Predictors

A decision stump makes it easy to construct a predictor such that

$$\sum_i w_{r-1,i} f_r(\mathbf{x}_i; \theta_r)$$

is negative. For other predictors, it may not be so easy. It turns out that this criterion can be modified, making it straightforward to use other predictors. There

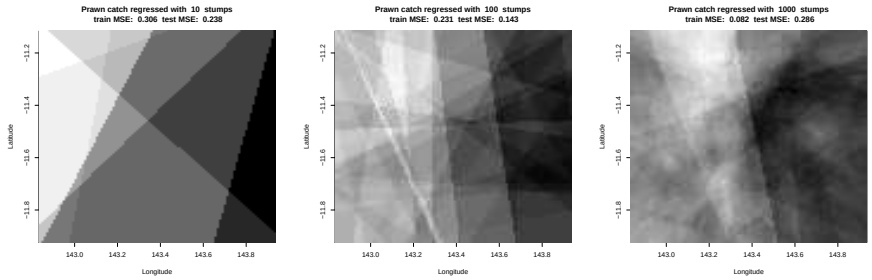


FIGURE 20.6: *Visualizations of the predictor for the prawn data of worked example 20.4. Mid-grey is zero; white is positive (and so any shade in the range mid-grey to white predicts a score greater than the median); black is negative (and so any shade in the range mid-grey to black predicts a score less than the median). The predictor formed by one stump cuts the region into two halves with a line; on one side of the line, it takes one constant value, and on the other, it takes a different constant value. Adding many of these together results in a function that is constant in each cell of an arrangement of lines. Predictors that use more stumps produce more complicated functions of this form, with improvements in training error and some improvement in test error (Figure 20.6).*

are two ways to think about these modifications, which end up in the same place: choosing f_r to minimize

$$\sum_i (w_{r-1,i} - f_r(\mathbf{x}_i; \theta_r))^2$$

is as good (or good enough) for gradient boost to succeed. This is an extremely convenient result, because many different regression procedures can minimize this loss. I will give both derivations, as different people find different lines of reasoning easier to accept.

Reasoning about minimization: Notice that

$$\sum_i (w_{r-1,i} - f_r(\mathbf{x}_i; \theta_r))^2 = \sum_i \begin{bmatrix} w_{r-1,i}^2 \\ +(f_r(\mathbf{x}_i; \theta_r))^2 \\ -2(w_{r-1,i} f_r(\mathbf{x}_i; \theta_r)) \end{bmatrix}.$$

Now assume that $\sum_i (f_r(\mathbf{x}_i; \theta_r))^2$ is not affected by θ_r. For example, f_r could be a decision tree that reports either 1 or -1. In fact, it is usually sufficient that $\sum_i (f_r(\mathbf{x}_i; \theta_r))^2$ is not much affected by θ_r. In this case, one way to obtain a small value of

$$\sum_i w_{r-1,i} f_r(\mathbf{x}_i; \theta_r)$$

is to obtain an f_r that matches the values of $w_{r-1,i}$ as closely as possible at each data point. So we seek f_r that minimizes

$$\sum_i (w_{r-1,i} - f_r(\mathbf{x}_i; \theta_r))^2$$

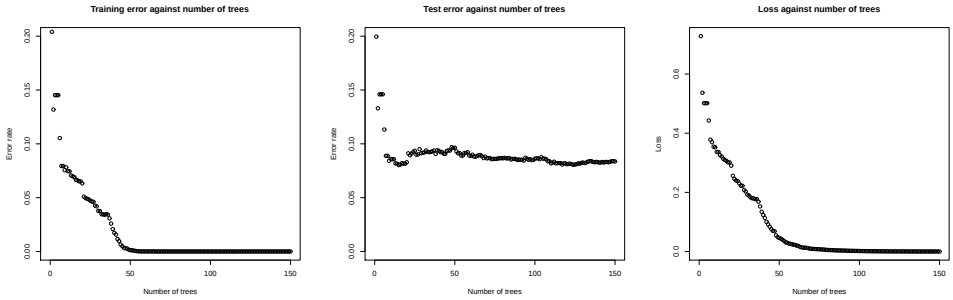


FIGURE 20.7: *Models for a boosted decision tree classifier predicting whether a given prescriber will write more than 10 opioid prescriptions in a year, using the data of https://www.kaggle.com/apryor6/us-opiate-prescriptions.* **Left:** *training error against number of trees;* **center:** *test error 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. Notice also a characteristic property of boosting; continuing to boost after the training error is zero (about 50 trees in this case) still results in improvements in the test error. Note also that lower exponential loss doesn't guarantee lower training error.*

Reasoning about descent directions: You can think of $\mathcal{L}$ as a function that accepts a vector of prediction values, one at each data point. Write $\mathbf{v}$ for this vector. The values are produced by the current predictor. In this model, we have that

$$\nabla_{\mathbf{v}}\mathcal{L} \propto w_{r-1,i}.$$

In turn, this suggests we should minimize $\mathcal{L}$ by obtaining a new predictor f_r which takes values as close as possible to $\nabla_{\mathbf{v}}\mathcal{L}$ – that is, choose f_r that minimizes

$$\sum_i (w_{r-1,i} - f_r(\mathbf{x}_i;\theta_r))^2.$$

20.2.6 Example: 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. The question here is: does a doctors pattern of prescribing predict whether that doctor will predict opiates? We will assume that there are no prescribers engaging in deliberate fraud (e.g. prescribing drugs that aren't necessary, for extra money). You can argue this question either way. For example, it is possible that doctors who see many patients who need opiates also see many patients who need other kinds of drug for similar underlying conditions. This would mean the pattern of drugs prescribed would suggest whether the doctor prescribed opiates. The other possibility is that patients who need opiates attend doctors randomly, so that the pattern of drugs prescribed isn't predictive.

We will predict the 'Opioid.Prescriber' column from the other entries, using a boosted decision tree and the exponential loss function. 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.

This is a fairly complicated classification problem. It is natural to try gradient boost using a regression tree. Doing so produces quite good classification (figure 57). You should notice there is a relatively large number of predictors here, and it's reasonable to wonder if one could get good results with fewer.

This is a good question. When you construct a set of boosted predictors, there is no guarantee they are all necessary to achieve a particular error rate. Each new predictor is constructed to cause the loss to go down. But the loss could go down without causing the error rate to go down. Here is an example. Assume you are using the exponential loss for a classifier. You can force the loss to go down by forcing the predictor to have larger magnitude for many examples that you get right already. This improves the loss, but doesn't improve the error rate. There may be a real benefit to doing this, because you ensure that predictions for points "close" to training points have larger magnitude, and this means there is a good chance the test error will go down. This isn't just an abstract possibility. Boosting classifiers with zero training error — where the loss may go down, but the training error certainly won't — sometimes results in improvements in test error in practice. You can see this effect in Figure 20.7. However, there is a reasonable prospect that some of the predictors are redundant.

Whether this matters depends somewhat on the application. It may be important to evaluate the minimum number of predictors. Furthermore, having many predictors could (but doesn't always) create generalization problems. One strategy to remove redundant predictors is to use the Lasso. For a two-class classifier, one uses a generalized linear model (logistic regression) applied to the values of the predictors at each example. Figure 20.8 shows the result of using a Lasso (from `glmnet`) to the predictors used to make Figure 20.7. Notice that reducing the size of the model seems not to result in significant loss of classification accuracy here.

There is one point to be careful about. You must have a training set to fit the boosted model and obtain the predictors for the Lasso method. It isn't wise to then attach the test set, and compute a cross-validated estimate of error on all data. This is because that estimate of error will be biased low, because you are using some data on which the predictors were trained. There are two options: you could fit a Lasso on the training data, then evaluate on test; or you could use cross-validation to evaluate a fitted Lasso on the test set alone. Neither strategy is perfect. If you fit a Lasso to the training data, you may not make the best estimate of coefficients, because you are not taking into account variations caused by test-train splits. But if you use cross-validation on the test set alone, you will be omitting quite a lot of data. This is a large dataset (25,000 prescribers) so I tried both approaches (Figure 20.8). A better option would be to apply the Lasso *during* the boosting process, but this is beyond our scope.

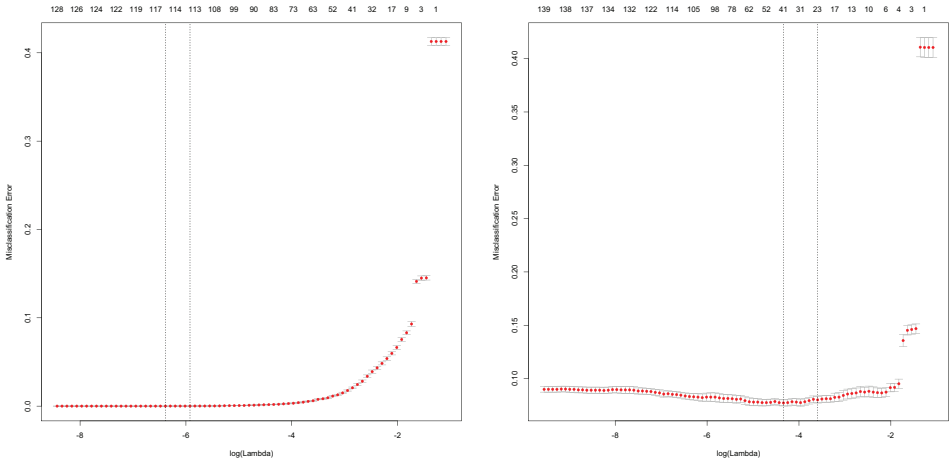


FIGURE 20.8: **Left:** *a cross-validation plot from* `cv.glmnet` *for the lasso applied to predictions on training data obtained from all 150 trees from the boosting of figure 20.7. A model with about 113 trees obtains zero error here. Recall from figure 20.7 that training error can be zero; it is here. But this estimate is biased because the trees have been chosen to fit the training data.* **Right:** *a cross-validation plot from* `cv.glmnet` *using the lasso to choose predictors on data not used to choose the predictors (ie test data). Notice how quite a small model (about 30 trees) achieves the minimum error. This cross-validated estimate of error is not biased, because the trees weren't chosen to fit the data. Notice that this implies a lasso fitted to test must have significantly higher error (it would use about 100 predictors, and you can see what the best such model on training data would do in this curve).*

20.3 YOU SHOULD

20.3.1 remember these definitions:

20.3.2 remember these terms:

20.3.3 remember these facts:

20.3.4 remember these procedures:

20.3.5 be able to:

-

THEORY

CHAPTER 21

A Little Learning Theory

A crucial question in applied machine learning is: how well will this work on test data? There is a body of theory that helps with this question, by providing a variety of bounds.

21.1 HELD-OUT LOSS PREDICTS TEST LOSS

Here is the simplest setup. Assume we have used training data to construct some predictor F. We have a pointwise loss $\ell(y, \mathbf{x}, F)$. We have N pairs $(\mathbf{x}_i, y_i)$ of held-out data items. We assume that none of these were used in constructing the predictor, and we assume that these pairs are IID samples from the distribution of test data, which we write $P(X, Y)$. We now evaluate the held-out error

$$\frac{1}{N}\sum_i \ell(y_i, \mathbf{x}_i, F).$$

Under almost all circumstances, this is a rather good estimate of the true expected loss on all possible test data, which is

$$\mathbb{E}_{P(X,Y)}[\ell].$$

In particular, quite simple methods yield bounds on how often the estimate will be very different from the true value. In turn, this means that the held-out loss is a good estimate of the test loss.

21.1.1 Sample Means and Expectations

Write L for the random variable whose value is obtained by drawing a sample $(\mathbf{x}_i, y_i)$ from $P(X, Y)$ and then evaluating $\ell(y_i, \mathbf{x}_i, F)$. We will study the relationship between the expected value, $\mathbb{E}[L]$, and the approximation obtained by drawing N IID samples from $P(L)$ and computing $\frac{1}{N}\sum_i L_i$. The value of the approximation is a random variable, because we would get a different value if we drew a different set of samples. We write $L^{(N)}$ for this random variable. Now we are interested in the mean of that random variable $\mathbb{E}\left[L^{(N)}\right]$ and its variance $\mathsf{var}\left[L^{(N)}\right] = \mathbb{E}\left[(L^{(N)})^2\right] - \mathbb{E}\left[L^{(N)}\right]^2$. We will assume that the variance is finite.

The mean: Because expectations are linear, we have that

$$\begin{aligned}
\mathbb{E}\left[L^{(N)}\right] &= \frac{1}{N}\mathbb{E}\left[L^{(1)} + \ldots + L^{(1)}\right] \\
&\quad \text{(where there are } N \text{ copies of } L^{(1)}\text{)} \\
&= \mathbb{E}\left[L^{(1)}\right] \\
&= \mathbb{E}[L]
\end{aligned}$$

The variance: Write L_i for the i'th sample used in computing $L^{(N)}$. We have $L^{(N)} = \frac{1}{N}\sum_i L_i$, so

$$\mathbb{E}\left[(L^{(N)})^2\right] = \frac{1}{N^2}\mathbb{E}\left[\sum_i L_i^2 + \sum_i \sum_{j \neq i} L_i L_j\right]$$

but $\mathbb{E}\left[L_i^2\right] = \mathbb{E}\left[L^2\right]$. Furthermore, L_i and L_j are independent, and $\mathbb{E}[L_i] = \mathbb{E}[L]$ so we have

$$\mathbb{E}\left[(L^{(N)})^2\right] = \frac{\left(N\mathbb{E}\left[L^2\right] + N(N-1)\mathbb{E}[L]^2\right)}{N^2}$$

In turn,

$$\mathbb{E}\left[(L^{(N)})^2\right] - \mathbb{E}\left[L^{(N)}\right]^2 = \frac{\left(\mathbb{E}\left[L^2\right] - \mathbb{E}[L]^2\right)}{N}.$$

There is an easier way to remember this. Write $\mathsf{var}[L] = \mathbb{E}\left[L^2\right] - \mathbb{E}[L]^2$ for the variance of the random variable L. Then we have

$$\mathsf{var}\left[L^{(N)}\right] = \frac{\mathsf{var}[L]}{N}.$$

This should be familiar. It is the standard error of the estimate of the mean. If it isn't, it's worth remembering. The more samples you use in computing the estimate of the mean, the better the estimate.

Useful Fact: 21.1 *Expressions for mean and variance of an expectation estimated from samples*

Write X for some random variable. Write $X^{(N)}$ for the mean of N IID samples of that random variable. We have that:

$$\begin{aligned}\mathbb{E}\left[X^{(N)}\right] &= \mathbb{E}[X] \\ \mathsf{var}\left[X^{(N)}\right] &= \frac{\mathsf{var}[X]}{N}\end{aligned}$$

Already, we have two useful facts. First, the held-out loss is the value of a random variable whose expected value is the test loss. Second, the variance of this random variable could be quite small, if we compute the held-out loss on enough held-out examples. If a random variable has small variance, you should expect to see values that are close to the mean with high probability (otherwise the variance would be bigger). The next step is to determine how often the held-out loss will be very different from the test loss.

21.1.2 Using Chebyshev's Inequality

Chebyshev's inequality links the observed value, the expected value and the variance of a random variable. You should have seen this before (if you haven't, look it up, for example, in chapter *** of ****); it appears in a box below.

Definition: 21.1 *Chebyshev's inequality*

For a random variable X with finite variance, **Chebyshev's inequality** states

$$P\left(\{|X - \mathbb{E}[X]| \geq a\}\right) \leq \frac{\mathsf{var}[X]}{a^2}.$$

Combining Chebyshev's inequality and the remarks above about sample mean, we have the result in the box below.

Useful Fact: 21.2 *Held-out error predicts test error, from Chebyshev*

There is some constant C so that

$$P\left(\left\{|L^{(N)} - \mathbb{E}[L]| \geq a\right\}\right) \leq \frac{C}{a^2 N}.$$

21.1.3 A Generalization Bound

Most generalization bounds give a value of $\mathbb{E}[L]$ that will not be exceeded with probability $1 - \delta$. The usual form of bound states that, with probability $1 - \delta$,

$$\mathbb{E}[L] \leq L^{(N)} + g(\delta, N, \ldots).$$

Then one studies how g grows as δ shrinks or N grows. It is straightforward to rearrange the Chebyshev inequality into this form. We have

$$P\left(\left\{|L^{(N)} - \mathbb{E}[L]| \geq a\right\}\right) \leq \frac{C}{a^2 N} = \delta.$$

Now solve for a in terms of δ. This yields that, with probability $1 - \delta$,

$$\mathbb{E}[L] \leq L^{(N)} + \sqrt{\frac{C\left(\frac{1}{\delta}\right)}{N}}.$$

Notice this bound is likely rather weak, because it makes the worst case assumption that all of the probability occurs when $\mathbb{E}[L]$ is larger than $L^{(N)} + w$. It does so

because we don't know where the probability is, and so have to assume it is in the worst place. I have put this bound in a box for convenience.

Useful Fact: 21.3 *Held-out error predicts test error, usual version, from Chebyshev*

There is some constant C depending on the loss and the test data so that with probability $1 - \delta$,

$$\mathbb{E}[L] \leq L^{(N)} + \sqrt{\frac{C\left(\frac{1}{\delta}\right)}{N}}.$$

This result tells us roughly what we'd expect. The held-out error is a good guide to the test error. Evaluating the held-out error on more data leads to a better estimate of the test error. The bound is very general, and applies to almost any form of pointwise loss. The "almost" here is because we must assume that the loss leads to an L with finite variance, but I have never encountered a loss that does not. This means the bound applies to both regression and classification problems, and you can use any kind of classification loss.

There are two problems. First, the bound assumes we have held-out error, but what we'd really like to do is think about test error in terms of training error. Second, the bound is quite weak, and we will do better in the next section. But our better bounds will apply only to a limited range of classification losses.

21.2 TEST AND TRAINING ERROR FOR A CLASSIFIER FROM A FINITE FAMILY

The test error of many classifiers can be bounded using the training error. I have never encountered a practical application for such bounds, but they are reassuring. They suggest that our original approach (choose a classifier with small training error) is legitimate, and they cast some light on the way that families of classifiers behave. However, these bounds are harder to obtain than bounds based on test error, because the predictor you selected minimizes the training error (at least approximately). This means that you should expect the training error to be an estimate of the test error that is too low – the classifier was chosen to achieve low training error. Equivalently, the training error is a biased estimate of the test error.

The predictor you selected came from a family of predictors, and the bias depends very strongly on the family you used. One example of a family of predictors is all linear SVM's. If you're using a linear SVM, you chose a particular set of parameter values that yields the SVM you fitted over all the other possible parameter values. As another example, if you're using a decision tree, you chose a tree from the family of all decision trees that have the depth, etc. limits that you imposed. Rather loosely, if you choose a predictor from a "big" family, then you should expect the bias is large. You are more likely to find a predictor with low training error and high test error when you search a big family.

The problem is to distinguish in a sensible way between "big" and "small" families of predictors. A natural first step is to consider only finite collections of predictors – for example, you might choose one of 10 fixed linear SVM's. Although this isn't a realistic model of learning in practice, it sets us up to deal with more difficult cases. Some rather clever tricks will then allow us to reason about continuous families of predictors, but these families need to have important and delicate properties exposed by the analysis.

From now on, we will consider only a 0-1 loss, because this will allow us to obtain much tighter bounds. We will first construct a bound on the loss of a given predictor, then consider what happens when that predictor is chosen from a finite set.

21.2.1 Hoeffding's Inequality

Chebyshev's inequality is general, and holds for any random variable with finite variance. If we assume stronger properties of the random variable, it is possible to prove very much tighter bounds.

Definition: 21.2 *Hoeffding's inequality for Bernoulli random variables*

Assume that X is a Bernoulli random variable that takes the value 1 with probability θ, and otherwise the value 0. Write $X^{(N)}$ for the random variable obtained by averaging N IID samples of X. Then **Hoeffding's inequality** states

$$P\left(|\theta - X^{(N)}| \geq \epsilon\right) \leq 2e^{-2N\epsilon^2}$$

Notice there is *not* an absolute value in the inequality. The proof is more elaborate than is really tolerable here, and I have isolated it below.

Now assume that our loss is 0-1. This is fairly common for classifiers, where you lose 1 if the answer is right, and 0 otherwise. The loss at any particular example is then a Bernoulli random variable, with probability $\mathbb{E}[L]$ of taking the value 1. This means we can use Hoeffding's inequality to tighten the bound of 414. Doing so yields

Useful Fact: 21.4 *Held-out error predicts test error, usual version, from Hoeffding*

With probability $1 - \delta$,

$$\mathbb{E}[L] \leq L^{(N)} + \sqrt{\frac{\log\left(\frac{1}{\delta}\right)}{2N}}.$$

This bound is tighter than the previous bound, because $\log\left(\frac{1}{\delta}\right) \leq \left(\frac{1}{\delta}\right)$. The difference becomes very important when δ is small, which is the interesting case.

21.2.2 Test from Training for a Finite Family of Predictors

Assume we choose a predictor from a *finite* set $\mathcal{P}$ of M different predictors. We will consider only a 0-1 loss. Write the expected loss of using predictor F as $E_F = \mathbb{E}_{P(X,Y)}[\mathcal{l}(y, \mathbf{x}, F)]$. One useful way of thinking of this loss is that it is the probability that an example will be mislabelled. Write $L_F^{(N)}$ for the estimate of this loss obtained from the training set for predictor F. From the Hoeffding inequality, we have

$$P\left(\left\{E_F - L_F^{(N)} \geq \epsilon\right\}\right) \leq e^{-2N\epsilon^2}$$

for *any* predictor F.

What we'd like to know is the generalization error for the predictor that we pick. This will be difficult to get. Instead, we will consider the worst generalization error in all the predictors — our predictor must be at least as good as this. Now consider the event $\mathcal{G}$ that at least one predictor has generalization error greater than ϵ. We have

$$\begin{aligned}\mathcal{G} = & \left\{E_{F_1} - L_{F_1}^{(N)} \geq \epsilon\right\} \cup \\ & \left\{E_{F_2} - L_{F_2}^{(N)} \geq \epsilon\right\} \cup \\ & \ldots \\ & \left\{E_{F_M} - L_{F_M}^{(N)} \geq \epsilon\right\}.\end{aligned}$$

Recall that, for two events $\mathcal{A}$ and $\mathcal{B}$, we have $P(\mathcal{A} \cup \mathcal{B}) \leq P(\mathcal{A}) + P(\mathcal{B})$ with equality only if the events are disjoint. The events that make up $\mathcal{G}$ may not be disjoint. But

by assuming that they are disjoint, we have an upper bound on $P(\mathcal{G})$. In particular,

$$\begin{aligned} P(\mathcal{G}) &\leq P\left(\left\{E_{F_1} - L_{F_1}^{(N)} \geq \epsilon\right\}\right) + \\ &\quad P\left(\left\{E_{F_2} - L_{F_2}^{(N)} \geq \epsilon\right\}\right) + \\ &\quad \ldots \\ &\quad P\left(\left\{E_{F_M} - L_{F_M}^{(N)} \geq \epsilon\right\}\right) \\ &\leq Me^{-2N\epsilon^2} \\ &\quad \text{by Hoeffding's inequality.} \end{aligned}$$

This is sometimes known as a **union bound**.

Now notice that $P(\mathcal{G})$ is the probability that at least one predictor F in $\mathcal{P}$ has $E_F - L_F^{(N)} \geq \epsilon$. Equivalently, it is the probability that the largest value of $E_F - L_F^{(N)}$ is greater than or equal to ϵ. So we have

$$\begin{aligned} P(\mathcal{G}) &= P\left(\{\text{at least one predictor has generalization error} > \epsilon\}\right) \\ &= P\left(\left\{ \sup_{F \in \mathcal{P}} \left[E_F - L_F^{(N)}\right] \geq \epsilon \right\}\right) \\ &\leq Me^{-2N\epsilon^2}. \end{aligned}$$

It is natural to rearrange this, yielding the bound in the box below. You should notice this bound does not depend on the *way* that the predictor was chosen.

Useful Fact: 21.5 *Training error predicts test error for a finite set of predictors, from Hoeffding*

Assume a 0-1 loss. Choose a predictor F from M different predictors, write the expected loss of using predictor F as $E_F = \mathbb{E}_{P(X,Y)}[\mathcal{l}(y, \mathbf{x}, F)]$, and write $L_F^{(N)}$ for the estimate of this loss obtained from the training set for predictor F. With probability $1 - \delta$,

$$E_F \leq L_F^{(N)} + \sqrt{\frac{\log M + \log\left(\frac{1}{\delta}\right)}{2N}}$$

for *any* predictor F chosen from a set of M predictors.

21.2.3 Number of Examples Required

Generally, we expect it is easier to find a predictor with good training error but bad test error (a bad predictor) when we search a large family of predictors, and the M in the bound reflects this. Similarly, if there are relatively few examples, it should be easy to find such a predictor as well; and if there are many examples, it

should be hard to find one, so there is an N in the bound, too. We can reframe the bound to ask how many examples we need to ensure that the probability of finding a bad predictor is small.

A bad predictor F is one where $E_F - L_F^{(N)} > \epsilon$. The probability that at least one predictor in our collection of M predictors is bad is bounded above by $Me^{-2N\epsilon^2}$. Now assume we wish to bound the failure probability above by δ. We can bound the number of examples we need to use to achieve this, by rearranging expressions, yielding the bound in the box.

Useful Fact: 21.6 *The number of examples required to bound the probability that at least one predictor is bad.*

A predictor F is bad if $E_F - L_F^{(N)} > \epsilon$. Write P_{bad} for the probability that at least one predictor in the collection (and so perhaps the predictor we select) is bad. To ensure that $P_{\text{bad}} \leq \delta$, it is enough to use

$$N \geq \frac{1}{2\epsilon^2}\left(\log M + \log\left(\frac{1}{\delta}\right)\right)$$

examples.

21.3 AN INFINITE COLLECTION OF PREDICTORS

Mostly, we're not that interested in choosing a predictor from a small discrete set. All the predictors we have looked at in previous chapters come from infinite families. The bounds in the previous section are not very helpful in this case. With some mathematical deviousness, we can obtain bounds for infinite sets of predictors, too.

We bounded the generalization error for a finite family of predictors by bounding the worst generalization error in that family. This was straightforward, but it meant the bound had a term in the number of elements in the family. If this is infinite, we have a problem. There is an important trick we can use here. It turns out that the issue to look at is not the number of predictors in the family. Instead, we think about predictors as functions that produce binary strings (section 21.3.1). This is because, at each example, the predictor either gets the example right (0) or wrong (1). Order the examples in some way; then you can think of the predictor as producing an N element binary string of 0's and 1's, where there is one bit for each of the N examples in the training set. Now if you were to use a different predictor in the family, you might get a different string. What turns out to be important is the number s of different possible strings that can appear when you try every predictor in the family. This number must be finite — there are N examples, and each is either right or wrong — but might still be as big as 2^N.

There are two crucial facts that allow us to bound generalization error. First, and suprisingly, there are families of predictors where s is small, and grows slowly

with N (section **??**). This means that, rather than worrying about infinite collections of predictors, we can attend to small finite sets of strings. Second, it turns out that we can bound generalization error using the difference between errors for some predictor given two different training datasets (section **??**). Because there are relatively few strings of errors, it becomes relatively straightforward to reason about this difference. These two facts yield a crucial bound on generalization error (section **??**).

Most of the useful facts in this section are relatively straightforward to prove (no heavy machinery is required), but the proofs take some time and trouble. Relatively few readers will really need them, and I have confined them to the next section.

21.3.1 Predictors and Binary Functions

A predictor is a function that takes an independent variable and produces a prediction. Because we are using a 0-1 loss, choosing a predictor F is equivalent to a choice of a binary function (i.e. a function that produces either 0 or 1). The binary function is obtained by making a prediction using F, then scoring it with the loss function. This means that the family of predictors yields a family of binary functions.

We will study binary functions briefly. Assume we have some binary function b in a family of binary functions $\mathcal{B}$. Take some sample of N points $\mathbf{x}_i$. Our function b will produce a binary string, with one bit for each point. We consider the set $\mathcal{B}_N$ which consists of all the different binary strings that are produced by functions in $\mathcal{B}$ for our chosen set of sample points. Write $\#(\mathcal{B}_N)$ for the number of different elements in this set. We could have $\#(\mathcal{B}_N) = 2^N$, because there are 2^N strings in $\mathcal{B}_N$.

In many cases, $\#(\mathcal{B}_N)$ is much smaller than 2^N. This is a property of the family of binary functions, rather than of (say) an odd choice of data points. The thing to study is the **growth function**

$$s(\mathcal{B}, N) = \sup_{\text{sets of } N \text{ points}} \#(\mathcal{B}_N).$$

This is sometimes called the **shattering number** of $\mathcal{B}$. For some interesting cases, the growth function can be recovered with elementary methods.

Worked example 21.1 *$s(\mathcal{B}, 3)$ for a simple linear classifier on the line*

Assume that we have a 1D independent variable x, and our family of classifiers is $\text{sign}(ax + b)$ for parameters a, b. These classifiers are equivalent to a family of binary functions $\mathcal{B}$. What is $s(\mathcal{B}, 3)$?

Solution: The predictor produces a sign at each point. Now think about the sample. It should be clear that the largest set of strings isn't going to occur unless the three points are distinct. The predictor produces a string of signs (one at each point), and the binary function is obtained by testing if the label is equal to, or different from, the sign the predictor produces. This means that the number of binary strings is the same as the number of distinct strings of signs. In particular, the actual values of the labels don't matter. Now order the three points along the line so $x_1 < x_2 < x_3$. Notice there is only one sign change at $s = -b/a$; we can have $s < x_1$, $x_1 < s < x_2$, $x_2 < s < x_3$, $x_3 < s$ (we will deal with s lying on a point later). All this means the predictor can produce only six sign patterns at most ($---$, $--+$, $-++$, $+++$, $++-$, $+--$). Now imagine that s lies on a data point; the rule is to choose a sign at random. It is straightforward to check that this doesn't increase the set of sign patterns (and you should). So $s(\mathcal{B}, 3) = 6 < 2^3$.

Worked example 21.2 *$s(\mathcal{B}, 4)$ for a simple linear classifier on the plane*

Assume that we have a 2D independent variable x, and our family of classifiers is $\text{sign}(ax + b)$ for parameters a, b. These classifiers are equivalent to a family of binary functions $\mathcal{B}$. What is $s(\mathcal{B}, 4)$?

Solution: The predictor produces a sign at each point. Now think about the sample. The predictor produces a string of signs (one at each point), and the binary function is obtained by testing if the label is equal to, or different from, the sign the predictor produces. This means that the number of binary strings is the same as the number of distinct strings of signs. It should be clear that the largest set of strings isn't going to occur unless the points points are distinct. If they're collinear, we know how to count (example 21.1), and obtain 10. You can check the case where three points are collinear easily, to count 12. There are two remaining cases. Either $\mathbf{x}_4$ is inside the convex hull of the other three, or it is outside. Figure **??** shows the case where it is outside. Notice that a linear predictor cannot predict + for points 1 and 3, and - for points 2 and 4. This means there are 14 strings of signs possible. If $\mathbf{x}_4$ is inside, then you cannot see $+++-$ or $---+$, so there are no more than 14 strings. So $s(\mathcal{B}, 4) = 14$.

The point of these examples is that an infinite family of predictors may yield a small family of binary functions. But $s(\mathcal{B}, N)$ can be quite hard to determine for

arbitrary families of predictors. One strategy is to use what is known as the **VC dimension** of $\mathcal{P}$ (after the inventors, Vapnik and Chervonenkis).

Definition: 21.3 *The VC dimension*

The VC dimension of a class of binary functions $\mathcal{B}$ is

$$\mathsf{VC}(\mathcal{B}) = \sup\left\{N : s(\mathcal{B}, N) = 2^N\right\}$$

Worked example 21.3 *The VC dimension of the binary functions produced by a linear classifier on the line*

Assume that we have a 1D independent variable x, and our family of classifiers is $\text{sign}(ax + b)$ for parameters a, b. These classifiers are equivalent to a family of binary functions $\mathcal{B}$. What is $\mathsf{VC}(\mathcal{B})$?

Solution: From example 21.1 this number is less than three. It's easy to show that $s(\mathcal{B}, 2) = 4$, so $\mathsf{VC}(\mathcal{B}) = 2$.

Worked example 21.4 *The VC dimension of the binary functions produced by a linear classifier on the plane*

Assume that we have a 2D independent variable x, and our family of classifiers is $\text{sign}(\mathbf{a}^T\mathbf{x} + b)$ for parameters a, b. These classifiers are equivalent to a family of binary functions $\mathcal{B}$. What is $\mathsf{VC}(\mathcal{B})$?

Solution: From example 21.1 this number is less than four. It's easy to show that $s(\mathcal{B}, 3) = 8$, so $\mathsf{VC}(\mathcal{B}) = 3$.

Talking about the VC dimension of the binary functions produced by a family of predictors is a bit long-winded. Instead, we will refer to the VC dimension of the family of predictors. So the VC dimension of linear classifiers on the plane is 3, etc.

Useful Fact: 21.7 *VC dimension of linear classifiers*

Write $\mathcal{P}$ for the family of linear classifiers on d-dimensional vectors, $\text{sign}(\mathbf{a}^T\mathbf{x} + b)$. Then

$$\mathsf{VC}(\mathcal{P}) = d + 1.$$

There are $d + 1$ parameters in a linear classifier on d-dimensional vectors, and the VC dimension is $d + 1$. Do not let this coincidence mislead you – you cannot obtain VC dimension by counting parameters. There are examples of one parameter families of predictors with infinite VC dimension (there is one in the exercises). Instead, you should think of VC dimension as measuring some form of "wiggliness". For example, linear predictors aren't wiggly because if you prescribe a sign on a small set of points, you know the sign on many others. But if a family of predictors has high VC dimension, then you can take a large set of points at random, prescribe a sign at each point, and find a member of the family that takes that sign at each point.

Useful Fact: 21.8 *The growth number of a family of finite VC dimension*

Assume $\mathsf{VC}(\mathcal{B}) = d$, which is finite. Then for all $N \geq d$, we have

$$s(\mathcal{B}, N) \leq \left(\frac{N}{d}\right)^d e^d$$

21.3.2 Symmetrization

A bad predictor F is one where $E_F - L_F^{(N)} > \epsilon$. For a finite set of predictors, we used Hoeffding's inequality to bound the probability a particular predictor was bad. We then argued that the probability that at least one predictor in our collection of M predictors is bad is bounded above by M times that bound. This won't work for an infinite set of predictors.

Assume we have a family of predictors which has finite VC dimension. Now draw a sample of N points. Associated with each predictor is a string of N binary variables (whether the predictor is right or wrong on each point). Even though there may be an infinite number of predictors, there is a finite number of distinct strings, and we can bound that number. We need a result that bound the generalization error in terms of the behavior of these strings.

Now assume we have a second IID sample of N points, and compute the average loss over that second sample. Write $\tilde{L}_F^{(N)}$ for this new average loss. This

second sample is purely an abstraction (we won't need a second training set) but it allows us to use an extremely powerful trick called symmetrization to get a bound. The result appears in two forms, in boxes, below.

Useful Fact: 21.9 *The variation of sample means yields a bound*

For any particular predictor F, we have

$$P\left(\left\{L_F^{(N)} - L_F > \epsilon\right\}\right) \leq 2P\left(\left\{L_F^{(N)} - \tilde{L}_F^{(N)} > \frac{\epsilon}{2}\right\}\right)$$

Useful Fact: 21.10 *The largest variation of sample means yields a bound*

$$P\left(\left\{\sup_{F \in \mathcal{P}} \mid L_F^{(N)} - L_F \mid > \epsilon\right\}\right) \leq 2P\left(\left\{\sup_{F \in \mathcal{P}} \left[\mid L_F^{(N)} - \tilde{L}_F^{(N)} \mid\right] > \frac{\epsilon}{2}\right\}\right)$$

The proof of this result isn't particularly difficult, but it'll be easier to swallow with some sense of why the result is important. Notice that the left hand side,

$$P\left(\left\{\sup_{F \in \mathcal{P}} \left[\mid L_F^{(N)} - \tilde{L}_F^{(N)} \mid\right] > \frac{\epsilon}{2}\right\}\right)$$

is expressed entirely in terms of the values that predictors take on data. To see why this is important, remember the example of the family of linear predictors for 1D independent variables. For $N = 3$ data points, an infinite family of predictors could make only six distinct predictions. This means that the event

$$\left\{\sup_{F \in \mathcal{P}} \left[\mid L_F^{(N)} - \tilde{L}_F^{(N)} \mid\right] > \frac{\epsilon}{2}\right\}$$

is quite easy to handle. Rather than worry about the supremum over an infinite family of predictors, we can attend to a supremum over only 36 predictions (which is six for $L_F^{(N)}$ and another six for $\tilde{L}_F^{(N)}$).

21.3.3 Bounding the Generalization Error

Useful Fact: 21.11 *Generalization bound in terms of VC dimension*

Let $\mathcal{P}$ be a family of predictors with VC dimension d. With probability at least $1 - \epsilon$, we have

$$\mid L^{(N)} - L \mid \leq \sqrt{\frac{8}{N}\left(\log\left(\frac{4}{\epsilon}\right) + d\log\left(\frac{Ne}{d}\right)\right)}$$

Proving this fact is straightforward with the tools at hand. We start by proving

Quick statement: *Generalization bound for an infinite family of predictors*

Formal Proposition: *Let $\mathcal{P}$ be a family of predictors, and $t \geq \sqrt{\frac{2}{N}}$. We have*

$$P\left(\left\{ \begin{array}{c} sup \\ F \in \mathcal{P} \end{array} \mid L_F^{(N)} - L_F \mid > \epsilon \right\}\right) \leq 4s(\mathcal{F}, 2N)e^{-N\epsilon^2/8}$$

Proof: Write b for a binary string obtained by computing the error for a predictor $p \in \mathcal{P}$ at $2N$ sample points, and b_i for it's i'th element. Write $\mathcal{B}$ for the set of all such strings. For a string b, write $L_b^{(N)} = \frac{1}{N}\sum_{i=1}^{N} b_i$ and $\tilde{L}_b^{(N)} = \frac{1}{N}\sum_{i=N+1}^{2N} b_i$. Now we have

$$\begin{aligned}
P\left(\left\{ \begin{array}{c} \sup \\ F \in \mathcal{P} \end{array} \mid L_F^{(N)} - L_F \mid > \epsilon \right\}\right) &\leq 2P\left(\left\{ \begin{array}{c} \sup \\ F \in \mathcal{P} \end{array} \mid L_F^{(N)} - \tilde{L}_F^{(N)} \mid > \epsilon/2 \right\}\right) \\
&\quad \text{using the symmetrization idea} \\
&= 2P\left(\left\{ \mathsf{max}\,(b \in \mathcal{B}, |) L_b^{(N)} - \tilde{L}_b^{(N)} \mid > \epsilon/2 \right\}\right) \\
&\quad \text{which is why symmetrization is useful} \\
&\leq 2\sum_{b \in \mathcal{B}} P\left(\left\{ \mid L_b^{(N)} - \tilde{L}_b^{(N)} \mid > \epsilon/2 \right\}\right) \\
&\quad \text{union bound} \\
&\leq 2\sum_{b \in \mathcal{B}} 2e^{-N\epsilon^2/8} \\
&\quad \text{Hoeffding} \\
&\leq 4s(\mathcal{B}, 2N)e^{-N\epsilon^2/8}
\end{aligned}$$

Quick statement: *Generalization bound for family of predictors with finite VC dimension*

Formal Proposition: *Let $\mathcal{P}$ be a family of predictors with VC dimension d. With probability at least $1-\epsilon$, we have*

$$\underset{F \in \mathcal{P}}{sup} \mid L_F^{(N)} - L_F \mid \leq \sqrt{\frac{8}{N}\left(\log\left(\frac{4}{\epsilon}\right) + d\log\left(\frac{Ne}{d}\right)\right)}$$

Proof: From above, we have

$$P\left(\left\{ \sup_{F \in \mathcal{P}} \mid L_F^{(N)} - L_F \mid > \epsilon \right\}\right) \leq 4s(\mathcal{B}, 2N)e^{-N\epsilon^2/8}$$

so with probability $1-\epsilon$,

$$L_F \leq L^N + \sqrt{\frac{8}{N}\left(\log\left(\frac{4s(\mathcal{B},N)}{\epsilon}\right)\right)}.$$

But we have that $s(\mathcal{B}, N) \leq \left(\frac{N}{d}\right)^d e^d$, so

$$\log\frac{4s(\mathcal{B},N)}{\epsilon} \leq \log\frac{4}{\epsilon} + d\log\left(\frac{Ne}{d}\right).$$

21.4 YOU SHOULD

21.4.1 remember these definitions:

21.4.2 remember these terms:

21.4.3 remember these facts:

21.4.4 use these procedures:

21.4.5 be able to:

- ?